Adaptation automatique et semi-automatique des optimisations de programmes

Lénaïc Bagnères

NNT : **2016SACLS295**

Thèse de Doctorat

de

l'Université Paris-Saclay

préparée à

l'Université Paris-Sud

École doctorale n° 580

Sciences et technologies de l'information et de la communication

Spécialité de doctorat : Informatique

Adaptation automatique et semi-automatique des optimisations de programmes

Par

M. **Lénaïc** Bagnères

Thèse présentée et soutenue à Gif-Sur-Yvette, le 30 septembre 2016 :

Composition du Jury :

Yannis Manoussakis, Professeur, Université Paris-Sud ..... Président du Jury
Erven Rohou, Directeur de recherche, Inria (Rennes - Bretagne Atlantique) ..... Rapporteur
Corinne Ancourt, Enseignant-chercheur, École nationale supérieure des mines de Paris ..... Rapporteur
Patrick Carribault, Ingénieur-chercheur, CEA ..... Examinateur
Christine Eisenbeis, Directrice de recherche, Inria (Saclay - Île-de-France) ..... Directrice de thèse
Cédric Bastoul, Professeur, Université de Strasbourg, ... Co-Directeur de thèse

Remerciements

Après quelques mois, je prends enfin le temps d'écrire quelques traditionnels mots de remerciement.

J'ai découvert l'informatique avec Jean-Paul Gélade, voisin, grand ami de la famille et autodidacte intéressé par fonctionnement des ordinateurs et des différentes technologies. Au fil de nos discussions, j'ai compris l'utilité des différents éléments matériels et logiciels d'un ordinateur. Les interactions entre carte mère - processeur - mémoires - disques dur, les différents types de stockage et la compression de données sont des exemples des sujets abordés. Celui des multi-processeurs avait été abordé avant la sortie des Pentium 4 dotés d'hyperthreading.

J'ai commencé à bidouiller avec David Guinehut [1]. En quelques années on avait monté un serveur de jeu, griffonné du PHP et du SQL et mis en ligne un site Internet. Un peu de programmation, un peu de GNU/Linux et un peu de réseau ont été suffisant pour me lancer dans des études en informatique.

J'ai commencé ma première semaine à l'IUT d'Orsay avec Cédric Bastoul [2] captivant dès l'amphi de présentation jusqu'aux derniers cours et TP de système et de Java. Durant ces deux années, j'ai également rencontré Sylvain Chevalier qui nous a enseigné le C++ ainsi que Guillaume Baron binôme jusqu'au master.

En licence, j'ai rencontré Daniel Etiemble [3], Cécile Germain [4] et Joël Falcou [5] qui enseignaient brillamment l'architecture des ordinateurs et les systèmes parallèles.

Durant mon stage en fin de master avec Joël Falcou, j'ai recroisé Cédric Bastoul.

J'ai commencé cette thèse sous la direction de Cédric Bastoul et de Christine Eisenbeis, dans l'équipe « Architectures parallèles », menée par Daniel Etiemble, Cédric Bastoul et Joël Falcou. Dans l'équipe, j'ai également pu longuement discuter avec Rémi Lacroix, Antoine Tran-Tan, Ian Masliah, Marie Laveau, Lionel Lacassagne [6] et Sylvain Jubertie [7]. Cette thèse n'aurait pas pu se faire sans Taj Muhammad Khan et Oleksandr Zinenko [8]. Mes plus grands remerciements vont à Oleksandr.

J'ai croisé de nombreuses personnes très intéressantes aux différentes conférences ; notamment à l'« École polyédrique » et aux « Journées de la compilation française ». La communauté de la compilation polyédrique est très accueillante. Durant ces événements, j'ai rencontré Erven Rohou [9] et Corinne Ancourt [10] que je remercie pour leur rapport. De façon générale, les *reviews* faites pour nos papiers ont été de qualité.

Je tiens également à remercier ma famille (Jean-Louis Bagnères, Agnès Broustaut, Maïlys Bagnères et Stéphanie Tortigues) qui a eu des retours plus que positifs sur mon pot de thèse ; Marine Landrieux [11] et sa famille pour me supporter quotidiennement ; et Émilie Marquebielle pour avoir relu intégralement ma thèse à la recherche de fautes et autres typos.

Depuis octobre 2016, je fais parti de NUMSCALE [12], jeune entreprise qui a des produits et des projets intéressants liés à la performance et aux optimisations logiciel.

1. http://dvdght.toile-libre.org/
2. http://icps.u-strasbg.fr/~bastoul/
3. https://www.lri.fr/~de/
4. https://www.lri.fr/~cecile/communications-english.html
5. https://www.lri.fr/~falcou/
6. http://www-soc.lip6.fr/~lacas/
7. http://warppipe.net/blog/pages/
8. https://ozinenko.com/
9. https://team.inria.fr/pacap/fr/members/rohou/
10. http://www.cri.mines-paristech.fr/people/ancourt/
11. http://marine-landrieux.appspot.com/
12. https://www.numscale.com/

université PARIS-SACLAY | ÉCOLE DOCTORALE Sciences et technologies de l'information et de la communication (STIC)

Titre : Adaptation automatique et semi-automatique des optimisations de programmes

Mots clés : **compilation polyédrique, optimisations de programmes**

Les compilateurs offrent un excellent compromis entre le temps de développement et les performances de l'application. Actuellement l'efficacité de leurs optimisations reste limitée lorsque les architectures cibles sont des multi-cœurs ou si les applications demandent des calculs intensifs. Il est difficile de prendre en compte les nombreuses configurations existantes et les nombreux paramètres inconnus à la compilation et donc disponibles uniquement pendant l'exécution. En se basant sur les techniques de compilation polyédrique, nous proposons deux solutions complémentaires pour contribuer au traitement de ces problèmes.

Dans une première partie, nous présentons une technique automatique à la fois statique et dynamique permettant d'optimiser les boucles des programmes en utilisant les possibilités offertes par l'*auto-tuning* dynamique. Cette solution entièrement automatique explore de nombreuses versions et sélectionne les plus pertinentes à la compilation. Le choix de la version à exécuter se fait dynamiquement avec un faible surcoût grâce à la génération de versions interchangeables : un ensemble de transformations autorisant le passage d'une version à une autre du programme tout en faisant du calcul utile.

Dans une seconde partie, nous offrons à l'utilisateur une nouvelle façon d'interagir avec le compilateur polyédrique. Afin de comprendre et de modifier les transformations faites par l'optimiseur, nous traduisons depuis la représentation polyédrique utilisée en interne n'importe quelle transformation de boucles impactant l'ordonnancement des itérations en une séquence de transformations syntaxiques équivalente. Celle-ci est compréhensible et modifiable par les programmeurs. En offrant la possibilité au développeur d'examiner, modifier, améliorer, rejouer et de construire des optimisations complexes avec ces outils de compilation semi-automatiques, nous ouvrons une boîte noire du compilateur : celle de la plateforme de compilation polyédrique.

Title : Automatic and Semi-Automatic Adaptation of Program Optimizations

Keywords : **polyhedral compilation, program optimizations**

Compilers usually offer a good trade-off between productivity and single thread performance thanks to a wide range of available automatic optimizations. However, they are still fragile when addressing computation intensive parts of applications in the context of parallel architectures with deep memory hierarchies that are now omnipresent. The recent shift to multicore architectures for desktop and embedded systems as well as the emergence of cloud computing is raising the problem of the impact of the execution context on performance.

Firstly, we present a static-dynamic compiler optimization technique that generates loop-based programs with dynamic auto-tuning capabilities with very low overhead. Our strategy introduces switchable scheduling, a family of program transformations that allows to switch between optimized versions while always processing useful computation. We present both the technique to generate self-adaptive programs based on switchable scheduling and experimental evidence of their ability to sustain high-performance in a dynamic environment.

Secondly, we propose a novel approach which aims at opening the polyhedral compilation engines that are powering loop-level optimization and parallelization frameworks of several modern compilers. Building on the state-of-the-art polyhedral representation of programs, we present ways to translate comprehensible syntactic transformation sequences to and from the internal polyhedral compiler abstractions. This new way to interact with high-level optimization frameworks provides an invaluable feedback to programmers with the ability to design, replay or refine loop-level compiler optimizations.

Résumé

Cette thèse porte sur l'adaptation automatique et semi-automatique des optimisations de programmes en utilisant le modèle polyédrique, un modèle mathématique permettant de représenter une certaine classe de programme. Ces optimisations peuvent être statiques, c'est-à-dire faites à la compilation, et dynamiques, c'est-à-dire faites à l'exécution. Une première approche automatique, statique et dynamique se concentre sur l'exploration, la sélection, la combinaison et l'évaluation de différentes versions issues du programme. Celles-ci sont générées par le compilateur polyédrique à partir du programme à optimiser afin d'exécuter la version la plus pertinente au bon moment. Une seconde approche semi-automatique et statique présente les optimisations faites par le compilateur polyédrique sous une forme compréhensible, rejouable et modifiable par n'importe quel développeur qui pourra alors raffiner l'optimisation afin de l'adapter à son cas.

Les architectures matérielles offrent une puissance de calcul de plus en plus élevée pour une consommation d'énergie relative de plus en plus faible mais deviennent de plus en plus complexes. C'est le rôle du programmeur expert de prendre en compte les différentes hiérarchies de mémoire et les multiples niveaux de parallélisme afin d'obtenir de bonnes performances. Malheureusement, pour la grande majorité des développeurs, il est difficile voire impossible, d'atteindre ces performances. Si la rapidité d'exécution est vue comme une fonctionnalité alors, dans de nombreux cas, celle-ci n'est pas toujours nécessaire. Généralement, l'utilisateur a besoin d'une certaine performance qui peut être éloignée des performances maximales. Il est raisonnable de laisser cette tâche aux outils automatiques ou semi-automatiques accessibles par n'importe quel programmeur non expert en techniques d'optimisation.

Les compilateurs offrent un excellent compromis entre le temps de développement et les performances de l'application. Actuellement l'efficacité de leurs optimisations reste limitée lorsque les architectures cibles sont des multi-cœurs ou si les applications demandent des calculs intensifs. Il est difficile de prendre en compte les nombreuses configurations existantes et les nombreux paramètres inconnus à la compilation et donc disponibles uniquement pendant l'exécution. En se basant sur les techniques de compilation polyédrique, nous proposons deux solutions complémentaires pour contribuer au traitement de ces problèmes.

Dans une première partie, nous présentons une technique automatique à la fois statique et dynamique permettant d'optimiser les boucles des programmes en utilisant les possibilités offertes par l'*auto-tuning* dynamique. Cette solution entièrement automatique explore de nombreuses versions et sélectionne les plus pertinentes à la compilation. Le choix de la version à exécuter se fait dynamiquement avec un faible surcoût grâce à la génération de versions interchangeables : un ensemble de transformations autorisant le passage d'une version à une autre du programme tout en faisant du calcul utile.

Dans une seconde partie, nous offrons à l'utilisateur une nouvelle façon d'interagir avec le compilateur polyédrique. Afin de comprendre et de modifier les transformations faites par l'optimiseur, nous traduisons depuis la représentation polyédrique utilisée en interne n'importe quelle transformation de boucles impactant l'ordonnancement des itérations en une séquence de transformations syntaxiques équivalente. Celle-ci est compréhensible et modifiable par les programmeurs. En offrant la possibilité au développeur d'examiner, modifier, améliorer, rejouer et de construire des optimisations complexes avec ces outils de compilation semi-automatiques, nous ouvrons une boîte noire du compilateur : celle de la plateforme de compilation polyédrique.

Ces deux approches complémentaires contribuent à l'élaboration d'optimisations automatiques et semi-automatiques plus robustes. L'utilisation des versions interchangeables permet d'exécuter la meilleure optimisation dynamiquement et automatiquement. Pour les cas où le développeur désire plus de contrôle, on offre un moyen simple d'interagir avec le compilateur polyédrique. Le développeur peut alors profiter de l'analyse précise et des transformations de code agressives du compilateur.

Table des matières

Chapitre 1

Introduction

Sommaire

Dans le monde d'aujourd'hui, la puissance de calcul est primordiale. De nombreux domaines ont besoin d'effectuer de nombreux calculs rapidement pour répondre à leur problèmatique. Les super-ordinateurs sont nécessaires afin de, par exemple, simuler le réel, étudier le vivant, créer des mondes virtuels, concevoir des interfaces homme-machine, prendre des décisions en traitant de nombreuses informations. Cette puissance de cacul permet de résoudre les défis industriels, scientifiques et militaires.

Simuler le réel. L'exemple le plus parlant est la prévision météorologique. De nombreux calculs sont nécessaires lors de la simulation et du post-traitement. Même si de plus en plus de données sont acquises et que, les complexes équations qui régissent l'atmosphère sont de plus en plus précises, de petites variations peuvent introduire de grands changements. Plusieurs simulations, à différentes échelles, sont faites. Le post-traitement s'occupe de sélectionner la simulation qui parait la plus réaliste. Depuis 2014, Météo-France dispose d'une puissance de calcul d'un *petaflops* (soit 10^{15} *flops*[1])[2]. Cette puissance de calcul est comparable à celle détenue par la 41^{e} machine (*non distribuée*) la plus puissante, en juin 2014, parmi celles qui participent à la liste du TOP500[3].

Étudier le vivant. Des projets comme BOINC[4] permettent d'utiliser les ressources matérielles que les utilisateurs laissent à leur disposition. Le problème à résoudre est envoyé sur un ordinateur connecté à Internet. Une fois le calcul fini, le résultat peut être retourné. Différents projets sont disponibles. Ils sont utiles pour la recherche scientifique et permettent, par exemple, d'étudier le réchauffement climatique, vérifier des hypothèses sur les maladies et mieux comprendre l'organisme humain. Début 2016, la puissance de calcul moyenne est de $11,16 \times 10^{15}$ flops. Cette puissance est répartie sur environ 240000 ordinateurs.

Créer des mondes virtuels. Les jeux vidéos de plus en plus réalistes, les effets spéciaux des films et les long métrages d'animation réalisés intégralement en images de synthèse font partie de notre quotidien. Ces applications demandent de nombreux calculs. Par exemple, le film d'animation *Kung Fu Panda 3*, sorti en 2016, a demandé 60 millions d'heures de calcul[5] cumulées. De plus, ces applications demandent

1. *floating-point operation per second*
2. http://www.meteofrance.fr/nous-connaitre/activites-et-metiers/les-supercalculateurs-de-meteo-france
3. https://www.top500.org/
4. http://boinc.berkeley.edu/
5. http://variety.com/2015/film/asia/jeffrey-katzenberg-kung-fu-panda-1201475355/

de plus en plus de calculs pour obtenir de plus en plus en plus de réalisme et de précision. Le rendu en trois dimensions, qui requiert deux images au lieu d'une, et l'augmentation de la résolution, contribuent significativement à cette demande de calculs supplémentaires.

Traiter de nombreuses informations. Sur un Internet de plus en plus centralisé, certaines entreprises comme celles derrières Google, Facebook ou Amazon récoltent et doivent traiter de plus en plus de données. Avec l'arrivée du *cloud computing*, les serveurs ont besoin de plus en plus de puissance de calcul pour assurer leurs services. Par exemple, les itinéraires pour la plupart des GPS disponibles sur téléphones mobiles, ne sont pas calculés sur le téléphone mais sur un serveur en ligne qui communique son résultat, les itinéraires, via le réseau.

Résoudre les défis industriels, scientifiques et militaires. Grâce à une modélisation des problèmes de plus en plus correcte et à une grande puissance de calcul disponible, de nombreux défis, comme la prévision météorologique, ont pu être relevés. Dans de nombreux cas, ces défis ne sont résolus que partiellement et l'amélioration du résultat passe par la nécessité d'une puissance de calcul de plus en plus importante. Les nouvelles technologies et les nouveaux usages sont de plus en plus gourmands en calcul.

Ce chapitre présente les ordinateurs d'aujourd'hui en commençant par leur origine, et explique comment on peut profiter, notamment grâce aux *langages de programmation* et aux *compilateurs*, de ces architectures matérielles de plus en plus difficiles à exploiter.

1.1 Ordinateurs & Processeurs

En 1834, Charles Babbage imagine la *machine analytique* [6]. Il s'agit de la première machine à calculer *programmable*. Bien que le prototype de cette machine ne fut jamais achevé, cette machine est considérée comme l'ancêtre des ordinateurs. La machine analytique était entièrement mécanique et comportait 50000 pièces pour une masse de trois tonnes (3000 kilogrammes). La précision de calcul était de 50 décimales et on pouvait mémoriser 1000 mots. Ada Lovelace participe à la machine analytique en étroite collaboration avec Charles Babbage. En formalisant les idées de ce dernier, elle écrit un algorithme très détaillé, aujourd'hui considéré comme le tout premier programme informatique [7].

Le Harvard Mark I [9] d'IBM peut être vu comme l'aboutissement de cette machine analytique. Cette machine a été conçue par Howard H. Aiken et livrée à Harvard en 1944. Pour programmer Harvard Mark I, les programmeurs utilisaient des *cartes perforées*. Grace Hopper faisait partie de l'équipe qui s'occupait de développer les programmes. Elle est considérée comme l'une des premières programmeuses en informatique. Grace Hopper est aussi connue pour avoir créé le premier compilateur en 1952 [10] (en fait, il s'agissait plus d'un *éditeur des liens* qu'un compilateur moderne). C'est elle qui choisira le terme de *compilateur*.

Si on définit un ordinateur comme étant une machine électronique, numérique, programmable, qui exécute des programmes (enregistrés en mémoire), afin d'effectuer des calculs ; alors la machine analytique de Charles Babbage n'est pas entièrement considéré comme un ordinateur. L'ENIAC [11] (*Electronic Numerical Integrator Analyser and Computer*) est le premier ordinateur entièrement électronique. Il fonctionnait avec 17468 *tubes à vide*. Cette machine de 30 tonnes (30000 kilogrammes) reparties sur 167 m^2 a été opérationnelle en 1946. Question performance, il pouvait effectuer, en une seconde, 5000 additions ou soustractions simples, ou 385 multiplications, ou 40 divisions, ou 3 racines carrées [12]. Pour programmer une telle machine et obtenir un résultat, il fallait manipuler des commutateurs et relier des câbles entre les différentes unités de l'ordinateur.

6. https://fr.wikipedia.org/wiki/Machine_analytique
7. Son prénom est repris pour nommer le langage de programmation Ada [8]
9. https://fr.wikipedia.org/wiki/Harvard_Mark_I
10. https://en.wikipedia.org/wiki/History_of_compiler_construction
11. https://fr.wikipedia.org/wiki/Electronic_Numerical_Integrator_Analyser_and_Computer
12. La machine fonctionnait environ la moitié du temps car plusieurs tubes à vide brûlaient presque tous les jours. Cela était dû au stress thermique lors de l'allumage ou l'arrêt de la machine, mais également suite à la présence d'insectes (*bugs* en anglais) sur les tubes chauds (cela explique l'étymologie du mot *bug informatique*).

Le successeur de l'ENIAC est l'EDVAC [13] (Electronic Discrete Variable Automatic Computer). Même si sa conception a commencé en 1944, il est sorti en 1949. Il utilisait le mode *binaire*, contrairement à l'ENIAC qui était en décimal. Il fonctionnait avec 6000 tubes à vide, faisait près de 8 tonnes (7850 kilogrammes) reparties sur $45,5$ m^2. Il s'agit du premier ordinateur à programme mémorisé.

John von Neumann a participé à la conception de l'EDVAC. Les données et les instructions du programme sont stockées dans la même structure. L'ordinateur est composé de quatre parties. L'unité arithmétique et logique (abrégé UAL, *Arithmetic Logic Unit* en anglais, abrégé ALU) effectue les calculs. L'unité de contrôle s'occupe de contrôler les autres parties en fonction des instructions. La mémoire contient le programme et les données. Les entrées-sorties permettent d'utiliser l'ordinateur. Cette architecture dite de von Neumann est encore celle utilisée pour les ordinateurs actuels. La programmation de ces architectures se faisait soit directement en langage machine (binaire), soit en langage assembleur lorsque ce dernier était disponible.

Le Manchester Mark 1 [14], opérationnel en 1949, utilise aussi l'architecture de von Neumann. Elle est la première à utiliser des *registres*, une mémoire à l'intérieur même du processeur, présente dans tous les processeurs modernes. C'est sur cette machine, en 1952, qu'Alick Glennie développe le premier *système de codage simplifié* [15] (*autocode* en anglais) ainsi que le premier compilateur (fonctionnant avec les mêmes étapes que les compilateurs actuels). On peut considérer cet autocode comme le premier langage compilé. On se posait déjà la question des performances atteintes par le compilateur : dans le manuel d'utilisation, on peut lire : « l'efficacité perdue ne dépasse pas 10% » [16].

En 1947, John Bardeen, William Shockley et Walter Brattain inventent le *transistor* [17]. Le transistor permet de remplacer le tube à vide. Il a beaucoup d'avantages. Il est plus petit, plus robuste, plus rapide et moins gourmand en énergie. Le premier processeur grand public utilisant des transistors est l'Intel 4004 [18] qui est commercialisé en 1971. Il comporte 2300 transistors pour des performances comparables à celles de l'ENIAC. Sa *fréquence d'horloge* est cadencée à 740 kHz. L'écriture d'un programme se faisait en assembleur [19].

En 1957, John Backus est à la tête d'une équipe au sein d'IBM qui développe, durant plus de deux ans et pour un coût équivalent au travail de 18 hommes sur un an, le premier compilateur pour Fortran, le premier langage de programmation dit de *haut-niveau*. Ce compilateur fonctionnait sur l'IBM 704 [20] sorti en 1954. Il s'agit du premier ordinateur de série pouvant effectuer des calculs en virgule flottante. Dans son article, « *The FORTRAN Automatic Coding System* » [21], dès l'introduction, on se pose la question de « produire des programmes aussi bons que ceux écrits par les humains ». Il est aussi indiqué que pour la plupart des programmes, plus de 90% du temps était passé à concevoir, écrire et débugger le programme alors que son exécution ne prenait que peu de temps. Dans le manuel de référence du programmeur de 1956, il est également annoncé que les « programmes produits par FORTRAN seront proches d'être aussi efficaces que ceux écrits par de bons programmeurs » [22].

À chaque nouveau processeur, la puissance de calcul des processeurs a grandement évoluée. Un des derniers, sorti en 2015, est l'Intel Core i7-6950X [23] [24]. Ce processeur contient $3,4$ milliards de transistors pour une surface de 246 mm^2. Ces transistors sont répartis en 10 *cœurs* de calcul *hyper-threadés* donnant un total de 20 *threads*. Leur fréquence est comprise entre 3 et 4 GHz. La performance *crête* (maximale) de ce processeur est de 1120 gflops.

13. https://fr.wikipedia.org/wiki/Electronic_Discrete_Variable_Automatic_Computer
14. https://en.wikipedia.org/wiki/Harvard_Mark_I
15. https://fr.wikipedia.org/wiki/Systèmes_de_codage_simplifiés
16. https://en.wikipedia.org/wiki/Autocode#Glennie.27s_Autocode
17. https://fr.wikipedia.org/wiki/Transistor
18. https://fr.wikipedia.org/wiki/Intel_4004
19. http://www.e4004.szyc.org/
20. https://en.wikipedia.org/wiki/IBM_704
21. http://archive.computerhistory.org/resources/text/Fortran/102663113.05.01.acc.pdf
22. http://archive.computerhistory.org/resources/text/Fortran/102649787.05.01.acc.pdf
23. http://ark.intel.com/fr/products/94456/Intel-Core-i7-6950X-Processor-Extreme-Edition-25M-Cache-up-to-3_50-GHz
24. http://www.hardware.fr/news/14643/intel-lance-i7-bdw-e-i7-6950x-tete.html

La section suivante revient sur l'évolution des ordinateurs et les principales avancées qui ont eu un impact significatif sur les performances et la difficulté d'en tirer parti.

1.1.1 Architecture des ordinateurs

Cette section présente deux modifications majeures permettant d'obtenir des ordinateurs de plus en plus rapides. La première est l'amélioration des mémoires et plus particulièrement celle de la mémoire *cache*. La seconde est la duplication des *cœurs de calcul* (unité physique permettant à elle seule d'exécuter un programme). Ces deux améliorations interviennent dans le processeur de l'ordinateur.

La *loi de* Moore réévaluée [25] est une prédiction basée sur des observations empiriques faites par Gordon E. Moore, cofondateur d'Intel. La loi de Moore prédit que la densité des transistors sur un processeur double tous les deux ans. Cette extrapolation empirique se révèle assez exacte. De ce fait, la puissance crête des ordinateurs est devenue de plus en plus élevée. On peut découper l'évolution des processeurs en deux phases. La première s'étale de 1971 à 2004. Durant cette période, la fréquence des processeurs passe de $0,000740$ GHz (Intel 4004) à $3,2$ GHz (Pentium 4). La deuxième période commence en 2005 avec l'arrivée du Pentium D d'Intel et du Athlon 64 X2 d'AMD qui marquent l'ère des processeurs *multi-cœurs* grand public.

De 1971 à 2004, le nombre de transistors et la fréquence des processeurs ont augmenté exponentiellement. Pour les applications limitées par la puissance de calcul qui ne s'exécutaient pas assez rapidement, il suffisait d'attendre la génération suivante de processeurs pour que ces mêmes applications aillent plus vite. En effet, l'augmentation des performances pour ce type d'application était automatique avec les nouvelles machines. Pour les autres applications, celles limitées par les accès à la mémoire, elles s'exécutaient aussi plus rapidement grâce à l'apparition et l'amélioration des mémoires cache.

L'évolution des mémoires cache est, elle-aussi, exponentielle même si cette dernière est moins spéculaire que celle du nombre de transistors. Au lieu d'utiliser directement la mémoire vive qui a un temps d'accès très long par rapport à la vitesse des calculs à l'intérieur du processeur ; les données transitent par une mémoire plus petite, mais plus proche du processeur, et plus rapide (mais plus coûteuse). Cette mémoire est la mémoire cache. Le processeur Intel 80386 [26] introduit en 1985 supporte l'utilisation d'un cache externe pour certains modèles. Le processeur suivant, le 80486 [27] sorti en 1989, inclut un cache interne (de niveau 1, dit *L1* pour *Level* 1 en anglais) unifié pour les données et les instructions. Sa taille est de 8 kio. Un second cache L2 peut être présent, il s'agit d'un cache externe dont la taille peut monter à 256 kio. Comme la miniaturisation des transistors continue et laisse plus de place dans les processeurs ; en 1997, le Pentium II [28] intègre le cache L2 dans le processeur et le cache L1 passe à 32 kio (16 kio pour les données, 16 kio pour les instructions). En 2005, les Pentium 4 intègrent jusqu'à 2 Mio de cache L2. En 2010, l'Intel i5-760 [29] dispose des caches internes L1 et L2 et d'un cache L3 externe. Comme ce processeur est un *multi-cœurs* avec 4 cœurs et que seul le cache L3 est partagé, chaque cœur contient un cache L1 de 32 kio pour les données, un cache L1 de 32 kio pour les instructions et un cache L2 de 256 kio. Le cache L3 est externe et fait 8 Mio. Pour finir, le i7-6950X vu précédemment a toujours les mêmes tailles de caches L1 et L2 par cœur. Son cache L3 fait 25 Mio.

L'évolution du nombre de transistors suit la loi de Moore. Au fil des différentes générations de processeurs, de plus en plus de transistors les composent. En parallèle, la mémoire cache qui était externe au processeur peut être intégrée à l'intérieur de ce dernier afin d'augmenter significativement la vitesse des transferts. Différents niveaux de caches sont apparus. Actuellement les caches L1 et L2 sont internes alors que le cache L3, lorsqu'il est présent, est externe. Souvent, il y a deux caches L1 : un pour les données et un pour les instructions. Les caches L1 ne sont pas partagés entre les différents cœurs : chaque cœur a ses propres caches L1. Si le cache L3 est présent, il est partagé entre les différents cœurs et chaque cœur dispose de son propre cache L2. Si le cache L3 n'est pas présent, c'est le cache L2 qui est partagé entre les différents cœurs. Jusqu'en 2004, l'augmentation des fréquences des processeurs était aussi exponentielle mais celle-ci a dû s'arrêter à cause d'un problème physique : la dissipation de la chaleur.

25. https://fr.wikipedia.org/wiki/Loi_de_Moore
26. https://en.wikipedia.org/wiki/Intel_80386
27. https://en.wikipedia.org/wiki/Intel_80486
28. https://en.wikipedia.org/wiki/Pentium_II
29. http://ark.intel.com/products/48496

Dans son article « *The Free Lunch Is Over* » [30], Herb Sutter explique pourquoi l'amélioration des fréquences des processeurs ne peut se poursuivre. Dans les plans initiaux d'Intel au milieu de l'année 2004, un processeur cadencé à 4 GHz était planifié mais ce dernier fut abandonné fin 2004. La raison principale est due au problème de dissipation de la chaleur générée par le processeur. En effet, l'augmentation de la fréquence n'est pas un problème en soi mais le système de refroidissement à installer sur de tels processeurs n'est plus réaliste pour les machines grand public. Par exemple, en 2006, un processeur est monté à 500 GHz grâce à un système de refroidissement avec de l'hélium liquide ($-269°C$) [31]. Comme la fréquence des processeurs ne suit pas la loi de Moore (et que le nombre de transistors continue de diminuer), les concepteurs de processeurs ont dû réfléchir à de nouvelles architectures. Ces dernières, présentes aujourd'hui dans toutes les machines grand public, y compris les téléphones portables, sont les *architectures parallèles*.

Depuis 2005, les processeurs grand public sont dotés de plusieurs cœurs de calculs. C'est un peu comme si les processeurs étaient composés de plusieurs processeurs en une seule puce. Contrairement à l'augmentation de la fréquence qui avait un impact direct sur les performances, l'augmentation du nombre de cœurs n'influence pas les performances des programmes *séquentiels* (qui s'exécutent sur un seul cœur). Il faut prendre en compte ces nouvelles architectures de façon logicielle en utilisant la programmation *parallèle*.

1.1.2 Adéquation Matériel - Logiciel & Programmation parallèle

L'adéquation matériel - logiciel consiste à concevoir et développer des programmes adaptés aux nouvelles architectures afin de tirer parti de leurs améliorations. Dans cette section, on s'intéressera à deux aspects des architectures : les mémoires cache et le *parallélisme*.

Les mémoires cache permettent d'améliorer significativement les performances en accédant plus rapidement aux données. Lorsqu'on accède à ces dernières, au lieu de lire et écrire dans la mémoire principale (mémoire vive) qui est lente, on passe par les différents niveaux de caches. Par exemple, dans le cas où il y a un seul cache, lorsqu'on lit une donnée, on charge depuis la mémoire principale tout le *bloc mémoire* où était présent la donnée demandée, dans une *ligne de cache*. Les données contiguës à la donnée lue sont également chargées dans la ligne de cache. Si la donnée suivante est lue juste après, son temps d'accès est très rapide car celle-ci est déjà dans le cache. Il n'est donc pas nécessaire de faire un accès à la mémoire principale. Pour profiter des mémoires cache, il faut prendre en compte la *localité*. Il y a deux sortes de localités mémoire : la *localité spatiale* et la *localité temporelle*.

Localité spatiale. Lorsque le programme accède à une donnée, il doit accéder aux données suivantes contiguës en mémoire. Ces données seront déjà dans le cache et les accès seront donc rapides. Ces chargements sont effectués par le matériel. En revanche c'est au logiciel d'accéder aux données de façon contiguë. La plupart des parcours de tableaux peuvent s'effectuer " dans le bon sens " mais avec certains algorithmes ou certaines structures de données comme les listes chaînées ou les graphes, il est parfois difficile d'appliquer le principe de localité spatiale.

Localité temporelle. Lorsque le programme accède à une donnée, il re-accède à cette donnée plusieurs fois et de façon suffisamment rapprochée de manière à ce que la donnée soit encore présente dans le cache. Ici aussi, c'est au logiciel de rapprocher les différentes utilisations des mêmes données et de restructurer le programme si nécessaire. Le *tuilage* (*tiling* en anglais) est une transformation classique de code permettant d'améliorer la localité temporelle des données et ainsi de profiter des effets bénéfiques de la mémoire cache.

En 2003, certains Pentium 4 d'Intel sont dotés de la technologie *hyper-threading* [32]. Un processeur est vu comme **deux** cœurs logiques. Cela permet, selon les applications, de gérer plus efficacement plusieurs programmes simultanément, et donc d'obtenir de meilleures performances (avec un ordre de grandeur compris entre 15 et 30 %). En interne, les différentes ressources du processeur sont mieux utilisées : le nombre de *défauts de cache* est réduit et il est plus facile de remplir le *pipeline* avec des *instructions* indépendantes. Malheureusement, notamment lorsque les performances de l'application sont limitées par les

30. http://www.gotw.ca/publications/concurrency-ddj.htm
31. http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1644829
32. https://en.wikipedia.org/wiki/Hyper-threading

accès mémoire, l'utilisation de l'hyper-threading peut dégrader les performances. La génération suivante de processeurs, sortis en 2005, introduit les processeurs multi-cœurs. Dans cette architecture, il y a bien plusieurs cœurs physiques qui ont chacun leur propre unité de calcul et leur propre cache L1. Ces processeurs multi-cœurs sont présents dans toutes nos machines, y compris dans nos téléphones portables. Il est donc essentiel de savoir programmer de telles architectures afin de profiter pleinement de leurs améliorations et de leurs performances. Cette tâche reste compliquée pour la plupart des développeurs, d'autant plus que plusieurs niveaux de parallélisme existent.

Parallélisme. Plusieurs formes de parallélisme existent. Certaines sont principalement gérées par le matériel alors que d'autres nécessitent des modifications profondes dans le logiciel. Le parallélisme permet d'exploiter au mieux possible l'ensemble des transistors.

Parallélisme au niveau bit. Même lorsque le processeur calcule une simple addition, il peut découper cette opération en différentes étapes et exécuter ces étapes en parallèle. Les techniques comme la *carry-lookahead adder*[33] combinée avec la *carry-select adder*[34] permettent cela malgré la dépendance générée par la propagation de la retenue. Cette technique était connue par Charles Babbage et a été implémentée par IBM en 1957. Ce parallélisme *élémentaire* permet d'utiliser l'ensemble des circuits afin d'accélérer les calculs simples. Même si ce parallélisme est entièrement géré par le matériel, le logiciel peut choisir, réorganiser et décomposer les instructions pour que le matériel puisse exécuter les instructions plus efficacement.

Parallélisme d'instructions. Un second parallélisme, aussi géré automatiquement par le matériel, est le parallélisme *d'instructions*, appelé aussi *pipeline*. Le pipeline est inspiré du fonctionnement des lignes de montage et a été utilisé pour la première fois sur l'IBM Strech[35] en 1961. Les instructions sont découpées en plusieurs étapes (par exemple : lecture instruction, décodage instruction, exécution, accès mémoire et écriture du résultat dans un registre). Pendant qu'une étape pour une instruction est exécutée, une autre étape pour une autre instruction peut être exécutée en parallèle, s'il ne s'agit pas de la même étape. Généralement, une étape prend un cycle. Une fois que le pipeline est plein, une instruction est exécutée à chaque cycle. Il s'agit d'un cas idéal. En pratique des soucis peuvent apparaître. Par exemple, lors d'un branchement, on ne sait pas quelle instruction lire et on ne peut donc pas commencer l'étape de la lecture de l'instruction suivante tant que l'instruction du branchement n'a pas été résolue. Ici aussi, l'ordre des instructions donné par le logiciel est important pour obtenir de bonnes performances.

Parallélisme de tâches. Avec l'arrivée des systèmes multi-cœurs, pour la plupart des applications, l'utilisation de ces nouvelles architectures n'apportait pas le gain de performance naïvement espéré. En effet, l'utilisation des différents cœurs disponibles dans le processeur n'est pas automatique pour une application. Il faut les prendre en compte de façon logicielle. C'est au développeur ou au compilateur de prendre en compte ce genre d'architecture. Pour utiliser efficacement les différents cœurs, il faut donc faire de la *programmation parallèle*, c'est-à-dire utiliser plusieurs *threads* (*processus légers*). Plusieurs solutions sont disponibles. On peut programmer des threads avec la *bibliothèque* C `pThread`[36], la bibliothèque standard C de la norme de 2011 (fonctionnalité optionnelle), `std::thread` venant de la bibliothèque stardard C++, *OpenMP*[37] ou avec MPI[38]. Les trois premiers, pThread, OpenMP et `std::thread` permettent de créer et manipuler des threads. MPI permet de créer et manipuler des processus. Comme les trois premiers, MPI permet donc de programmer les architectures multi-cœurs mais également les systèmes multi-processeurs ou multi-ordinateurs. On parle de *parallélisme de tâches*.

Parallélisme de données. En 2007, NVidia lance un GPGPU[39] (*General-Purpose computing on Graphics Processing Units*) programmable via leur kit de développement CUDA[40]. Ces architectures peuvent être utilisées comme cartes accélératrices. Si l'on compare les cœurs des GPGPU avec ceux des processeurs classiques, on remarque qu'ils ont été grandement simplifiés. Des unités comme la gestion de la cohérence des caches, de la traduction et la réorganisation des instructions ou encore celles permet-

33. https://en.wikipedia.org/wiki/Carry-lookahead_adder
34. https://en.wikipedia.org/wiki/Carry-select_adder
35. https://fr.wikipedia.org/wiki/IBM_Stretch
36. https://fr.wikipedia.org/wiki/Threads_POSIX
37. https://fr.wikipedia.org/wiki/OpenMP
38. https://fr.wikipedia.org/wiki/Message_Passing_Interface
39. https://fr.wikipedia.org/wiki/General-purpose_processing_on_graphics_processing_units
40. https://fr.wikipedia.org/wiki/Compute_Unified_Device_Architecture

tant d'effectuer des branchements efficaces, ont été enlevées. Ces unités, qui ne faisaient pas de calculs arithmetiques et logiques, prennent plus de 90% de l'espace du processeur. En libérant la place qu'elles occupaient, il est alors possible de dupliquer de nombreuses fois les unités qui prennaient moins de 5% de la place : celles qui effectuent les calculs. Par exemple, la carte NVidia Titan X [41] sortie en 2015, dispose de 3072 cœurs CUDA. En contrepartie, programmer ces cartes est très compliqué. OpenCL [42] est le standard multi-plateformes pour programmer ces architectures mais c'est souvent CUDA qui est utilisé même s'il n'est compatible qu'avec les cartes NVidia. Chaque cœur exécute le même code. Mais de par la simplification des cœurs, ces architectures ciblent principalement les applications effectuant du calcul régulier. Par exemple, il n'est pas possible pour deux cœurs d'exécuter des instructions différentes : si le code contient un branchement, les cœurs qui ont évalué la condition à vrai exécutent le code correspondant pendant que les autres attendent ; puis ensuite, c'est à eux d'attendre pendant que les autres exécutent le code correspondant à la condition évaluée à faux. Des cartes accélératrices, autres que les GPGPU, disposent d'un grand nombre de cœurs simplifiés (mais pas autant).

En 2010, Intel lance le premier Xeon Phi [43]. Cette carte contient 32 processeurs comprennant 4 cœurs chacun (soit un total de 128 *cœurs*). Ces cœurs sont cadencés à 1, 2 GHz et ont une hierarchie de mémoires cache contrairement à la plupart GPGPU. En 2013, le Xeon Phi « *Knights Landing* » [44] intègre 72×4 cœurs (288 cœurs). Pour programmer les différents cœurs de cette architecture, on utilise les threads (parallélisme de tâches). Chaque cœur supporte les *instructions vectorielles* AVX-512 permettant de traiter les flottants simple précision 32 bits par paquet de 16 (parallélisme de données).

Le parallélisme de données existe bien avant les GPGPU et est présent dans les processeurs classiques depuis longtemps. Le jeu d'instructions vectorielles AVX-512 est une évolution de MMX [45]. Ce jeu d'instructions vectorielles permettant de traiter les flottants simple précision 32 bits par paquet de 2 est sorti en 1997. Entre MMX et AVX-512, il y a eu les jeux d'instructions SSE [46] (Streaming SIMD Extensions). SIMD signifie *Single Instruction Multiple Data*. Par exemple, SSE2 sort en 2003 et permet de traiter les flottants simple précision 32 bits par paquet de 4. Un dernier exemple, AVX sort en 2011 et permet de traiter les flottants simple précision 32 bits par paquet de 8. Tous les processeurs récents disposent d'un jeu d'instructions vectorielles, y compris les processeurs ARM équipant les téléphones portables. Pour profiter de ce parallélisme, on peut soit utiliser directement le jeu d'instructions vectorielles de la machine donnant des programmes non portables, soit utiliser une bibliothèque haut-niveau comme Boost.SIMD [47], soit appeler un auto-vectoriseur comme celui embarqué dans le compilateur.

Parallélisme spécifique. Par définition, ce parallélisme est spécifique. Il ne s'agit pas d'une unité de calcul programmable mais d'une puce conçue pour une tâche spécifique/précise. Il s'agit d'une solution matérielle spécialisée dans le traitement d'un problème particulier. Ces puces sont très répandues dans les cartes graphiques et les téléphones portables. Elles permettent, par exemple, de décoder un flux vidéo ou d'appliquer des filtres de traitement d'images directement après la capture de l'image. Ce parallélisme est très performant et peu gourmand en énergie car c'est le matériel conçut pour, qui se charge d'effectuer les calculs. Les deux grandes familles de puces sont les *FPGA* [48] (*Field-Programmable Gate Array*) et les *ASIC* [49] (*Application-Specific Integrated Circuit*). Les ASIC sont figés alors que les FPGA peuvent être reconfigurées.

Les deux points clés pour obtenir de bonnes performances sont la localité des données et le parallélisme. Appliquer le principe de localité spatiale et temporelle permet d'exploiter pleinement les bénéfices apportés par les mémoires cache. Ces améliorations peuvent être automatiques mais peuvent aussi parfois demander de restructurer les accès mémoire du programme. Depuis l'introduction des architectures parallèles dans toutes les machines grand public, les programmes doivent être souvent réécrits pour tirer partie de la puissance de calcul disponible. Il faut donc modifier les applications et introduire de nouveaux paradigmes de la programmation parallèle sans oublier le fonctionnement des mémoires cache. On parle d'adéquation matériel - logiciel.

41. http://www.tomshardware.fr/articles/nvidia-geforce-gtx-titan-x-gm200,2-2370.html
42. https://fr.wikipedia.org/wiki/OpenCL
43. https://en.wikipedia.org/wiki/Xeon_Phi
44. http://wccftech.com/intel-xeon-phi-knights-landing-features-integrated-memory-500-gbs-bandwidth-ddr4-memory-suppor
45. https://fr.wikipedia.org/wiki/MMX_%28jeu_d'instructions%29
46. https://fr.wikipedia.org/wiki/Streaming_SIMD_Extensions
47. https://isocpp.org/blog/2014/02/nt2-boost.simd
48. https://fr.wikipedia.org/wiki/Circuit_logique_programmable
49. https://fr.wikipedia.org/wiki/Application-specific_integrated_circuit

1.1.3 *The Wall*

Si le nombre de transistors dans un processeur a pu suivre la loi de Moore, c'est grâce à la miniaturisation des transistors. En effet, la *finesse de gravure* des transistors est de plus en plus avancée. En 1971, la taille d'un transistor sur l'Intel 4004 était de 10 µm. La taille évolue pour arriver à un µm en 1985 pour l'Intel 80386. Sur les Pentium 4 de 2004, la finesse de gravure passe à 90 nm. Elle passe à 45 nm en 2008, 32 nm en 2010, 22 nm en 2012 et 14 nm en 2014. Le 10 nm est prévu pour 2017 alors que les dates de sortie des tailles plus petites restent floues. 7 nm pourrait sortir en 2020 et 5 nm en 2023.

Avec une telle finesse de gravure, les transistors se rapprochent de la taille d'une molécule. En se rapprochant des limites physiques, les transistors sont de plus en plus difficiles à fabriquer. Comme la taille d'un atome de silicium est d'environ 0,11 nm, pour les transistors 14 nm, la longueur de la *porte* du transistor est d'environ 90 atomes. Avec si peu d'atomes, les effets physiques jusque là ignorés se manifestent. Par exemple, il est difficile d'obtenir une *résistance* et un *rayonnement ionisant*[50], nécessaires au fonctionnement d'un transistor. *Actuellement*, la limite physique théorique est estimée à 5 nm par les experts. Et même si des prototypes 5 nm, 4 nm et 3 nm ont déjà été créés, il y a une limite à la miniaturisation des transistors. Cette limite est appelé « *The Wall* » (*le mur* en français) par l'industrie.

Pour dépasser ce mur, les multi-cœurs ne suffisent plus. Plusieurs pistes sont envisagées[51]. Il est possible d'utiliser du graphène à la place du silicium pour la fabrication des transistors. En 2008, des chercheurs ont créé le plus petit transistor. Il fait 1 nm : plus exactement, son épaisseur est de 1 atome et sa largeur est de 10 atomes. Il est également possible d'empiler les transistors pour fabriquer une puce 3D. Cette structure pose de nouveau des problèmes de refroidissement pour les transistors mais pas pour la mémoire vive. Il serait donc possible de rapprocher ces deux éléments. Les autres pistes, plus éloignées, sont les *transistors molécules* et l'*informatique quantique*.

1.1.4 *Outside the Wall*

Les processeurs disposent d'un ensemble de fonctionnalités permettant d'exécuter relativement efficacement de nombreux programmes. Par exemple, la gestion du cache, le pipeline d'instructions et la prédiction de branchement sont automatiques pour le logiciel car ils sont entièrement gérés par le matériel. En contrepartie, leur puissance de calcul crête est faible si on la compare à celle offerte par les GPGPU. À l'inverse des processeurs, les GPGPU se concentrent sur leur puissance de calcul brute sans offrir les fonctionnalités habituellement effectuées par le matériel. C'est au programmeur de pallier ces manques et de gérer le matériel lui-même.

Le fonctionnement, l'utilisation et le but des processeurs et des GPGPU sont différents. À long terme, on ne sait pas si une technologie l'emportera sur une autre ou si elles co-existeront en symbiose. Ces dernières années les processeurs sont devenus de plus en plus complexes. Mais avec l'arrivée des processeurs ARM présents dans les téléphones portables, les objectifs ont changés. En général, les architectures se sont simplifiées afin d'être moins gourmandes en énergie.

En 2011, AMD lance son premier APU[52] (*Accelerated Processing Unit*). Il s'agit d'un processeur couplé avec une GPGPU, sur la même puce. En 2014, les APU d'AMD sont utilisés dans la PlayStation 4 de Sony, l'une des consoles de jeux les plus répandues. En 2010, Intel a une technologie similaire même si le terme APU n'est pas utilisé. On parle d'IGP[53] (*Integrated Graphics Processors*). La programmation des APU rappelle celle des machines hétérogènes qui sont composées de processeurs et de GPGPU. Même si dans le cas des APU, la mémoire entre le processeur et le GPGPU est unifiée, programmer ces nouvelles architectures efficacement posent les mêmes problématiques.

Les cartes *embarquées* comme la NVidia Tegra X1[54] disposent d'un processeur et d'un GPGPU dont la mémoire est unifiée. Sur certaines de ces cartes, il y a plusieurs processeurs. Le premier permet d'obtenir des bonnes performances. Le deuxième permet de consommer peu d'énergie. Si le problème est adapté

50. https://fr.wikipedia.org/wiki/Rayonnement_ionisant
51. http://www.zdnet.fr/actualites/processeurs-la-loi-de-moore-c-est-termine-quid-de-la-suite-39832776.htm
52. https://en.wikipedia.org/wiki/AMD_Accelerated_Processing_Unit
53. https://en.wikipedia.org/wiki/Intel_HD_and_Iris_Graphics#Generations
54. http://www.nvidia.fr/object/tegra-x1-processor-fr.html

à une exécution efficace sur GPGPU, le premier processeur est désactivé et le deuxième n'est pas utilisé pour faire des calculs mais pour orchestrer les calculs du GPGPU. Une fois de plus, ces architectures hétérogènes demandent des connaissances techniques très avancées pour être programmées efficacement.

Malgré une puissance crête de plus en plus élevée, il est difficile d'utiliser toute cette puissance de calcul. Le logiciel doit être adapté pour tirer parti des performances offertes par le matériel. C'est une tâche complexe et compliquée qui a un coût. Le titre de l'article « *Welcome to the Jungle* »[55] d'Herb Sutter résume bien la situation en parlant des multi-cœurs, des systèmes hétérogènes, du cloud computing et d'une possible fin de la loi de Moore.

1.1.5 Coût des développements logiciels

Avec les systèmes multi-processus et multi-utilisateurs ainsi que les interfaces graphiques, le développement de logiciels est déjà devenu complexe et compliqué, et semble demander une expertise de plus en plus forte.

Complexe, car, par nature, programmer demande une certaine rigueur et une compréhension suffisamment claire de l'algorithme pour " l'expliquer " à l'ordinateur.

Compliqué, car, indépendemment de l'algorithme à implémenter, de nombreux facteurs sont à prendre en compte et cela demande un niveau technique avancé ou très avancé ainsi qu'une forte expertise.

Tout au long de l'histoire de l'informatique, les techniques de développement ont évolué afin d'obtenir plus rapidement des applications plus fiables, plus robustes, plus rapides, plus maintenables et qui peuvent évoluer plus facilement. Le *génie logiciel* s'intéresse à ces problématiques et définit plusieurs méthodes pour y arriver.

Même avec ces méthodes, l'adéquation matériel - logiciel et l'optimisation restent compliqués. Les langages dits de *haut-niveau* permettent d'abstraire une partie du fonctionnement interne de la machine. Par exemple, le développeur C++ n'a généralement pas à se soucier de la gestion de le mémoire grâce au *RAII*[56]. Mais dans certains cas, il est nécessaire d'utiliser directement les interfaces dites *bas-niveau*, proches de la machine, pour arriver aux performances désirées. Par exemple, en C++ comme en C, on peut utiliser directement les instructions vectorielles disponibles sur les processeurs si les phases de vectorisation automatique du compilateur échouent.

Les langages de programmation, les différentes *bibliothèques* et *plateformes*, les compilateurs et autres outils comme ceux qui composent les environnements de développement intégrés (*Integrated Development Environment* en anglais, abrégé IDE), permettent de simplifier grandement le travail du développeur.

Les langages de programmation peuvent être bas-niveau ou haut-niveau ou les deux à fois. Certains offrent plusieurs paradigmes de programmation. Le choix du langage pour les développements n'est pas toujours basé uniquement sur les détails techniques mais intègre souvent une part de préférences personnelles. Si le langage est adapté au problème à traiter, le développement sera alors moins coûteux.

Beaucoup de bibliothèques et plateformes existent. Utiliser du code existant permet de développer plus vite et d'obtenir rapidement un programme qui est souvent plus robuste, plus fiable, plus rapide et plus maintenable.

Pour les langages compilés comme C et C++, le compilateur est d'une importante capitale. Son rôle est de vérifier la syntaxe et la cohérence du programme. Par exemple, grâce à un système de type avancé, un programme C++ qui compile (avec les bonnes options et sans *warning*) fait généralement ce que le développeur a demandé (indépendemment de ce que le développeur a voulu demander). Actuellement les *optimiseurs* des compilateurs de production offrent de bonnes performances lorsque l'exécution du programme se fait sur un seul cœur. Les optimisations permettant de cibler automatiquement les multi-cœurs sont jugées expérimentales lorsqu'elles sont présentes.

Les environnements de développement intégrés proposent une interface pour, au minimum, éditer le code source, compiler et exécuter le programme en mode *release* ou *debug*. D'autres fonctionnalités peuvent être proposées. Par exemple, la génération de code idiomatique (comme les *getters* et les *setters*),

55. https://herbsutter.com/welcome-to-the-jungle/
56. http://en.cppreference.com/w/cpp/language/raii

la refactorisation de code, l'analyse de code statique ou le *profilage dynamique* peuvent être disponibles selon les IDE.

Malgré l'ensemble de ces outils, programmer les architectures d'aujourd'hui et celles de demain dans le but de profiter de leur puissance de calcul, demande des connaissances avancées en adéquation matériel - logiciel. Comme peu de développeurs sont intéressés par ces problématiques, une solution est de se tourner vers les approches automatiques. Par exemple, certains compilateurs savent générer des transformations agressives pour améliorer la localité des données et savent extraire le parallélisme. Plusieurs optimisations sont possibles et il est difficile pour le compilateur de choisir celle qui sera la plus adaptée pour l'utilisation faite du programme. Il est alors nécessaire d'adapter ces optimisations de programmes au besoin de l'utilisateur.

1.1.6 Conclusion

Le chapitre 1 vient de présenter l'histoire et l'évolution des ordinateurs : le début des ordinateurs aux architectures modernes, en passant par les premiers compilateurs. Comme ces architectures deviennent de plus en plus complexes et de plus en plus hétérogènes, il est de plus en plus difficile d'obtenir des performances proches des performances crêtes disponibles. Le but de l'adéquation matériel - logiciel et des compilateurs est de répondre aux problématiques de performance, d'énergie et de coût et temps de développement. L'approche manuelle est efficace pour les performances mais coûteuse. L'approche automatique est efficace pour la productivité mais peut donner des optimisations non adaptées. Dans cette thèse, nous présentons une approche automatique et une approche semi-automatique afin d'adapter les optimisations faites par le compilateur. Nos deux approches se basent sur le *modèle polyédrique* : une abstraction permettant de représenter une sous-partie des programmes.

1.2 Organisation du document

Le chapitre 2 présente le modèle polyédrique utilisé dans ce document et reposant sur l'union de relations basiques. Dans ce chapitre sont présentées les différentes parties de la relation basique : le domaine d'itération, l'ordonnancement des instances de l'instruction et les accès mémoire. Nous définissons un ensemble de contraintes limitant la structure et les valeurs des relations d'ordonnancement. Ces contraintes sont essentielles à notre approche semi-automatique.
Différents outils sont également présentés : le format OpenScop, l'extracteur de la représentation polyédrique depuis un programme, l'analyseur de dépendances, l'optimiseur polyédrique automatique et manuel, ainsi que le générateur de code. L'ensemble de ces outils, en les combinant, permet de créer une plateforme de compilation polyédrique. La suite de nos travaux se base sur une telle plateforme.

Le chapitre 3 présente rapidement les limites des optimiseurs polyédriques automatiques qui créent des optimisations trop génériques ou trop spécialisées. Différents exemples sont présentés pour pointer la fragilité de la portabilité des optimisations.

Le chapitre 4 présente un premier moyen d'adapter l'optimisation d'un programme sans l'intervention du programmeur. Il s'agit d'une nouvelle technique hybride associant optimisations statiques et dynamiques. L'idée est de créer un programme multi-versions qui sait s'adapter durant son exécution en tirant profit de la meilleure version disponible. Cette solution répond à la problématique posée au chapitre 3 grâce à l'introduction des *versions interchangeables* : un ensemble de transformations autorisant le passage d'une version du code optimisé à une autre.

Le chapitre 5 présente une plateforme de transformations polyédriques basées sur les transformations classiques de code. Nos transformations interviennent uniquement sur la relation d'ordonnancement et dans le respect des conditions présentées dans le chapitre 2. Cet ensemble de transformations, comme ses prédécesseurs, permet à n'importe quel développeur d'utiliser la puissance du modèle polyédrique sans connaissance de ce modèle. En revanche, notre plateforme de transformations est la première à être complète : elle permet d'exprimer *n'importe quel ordonnancement polyédrique.*

Le chapitre 6 présente une seconde manière d'adapter les optimisations d'un programme en permettant l'intervention du programmeur grâce à la plateforme présentée au chapitre 5. Nous présentons le premier algorithme permettant d'exprimer l'ordonnancement polyédrique d'un optimiseur automatique sous la

forme d'une séquence de transformations définies dans le chapitre 5. De par le fonctionnement de l'optimiseur polyédrique automatique, comprendre son résultat est quasiment impossible sans connaissances avancées dans la représentation polyédrique. En traduisant ce résultat en séquence de transformations haut-niveau, le développeur peut comprendre mais également modifier l'optimisation proposée par le compilateur polyédrique sans connaissance de ce modèle.

Le chapitre 7 conclut ce document et présente les perspectives.

L'annexe A donne le vocabulaire de base autour des langages de programmation à travers des exemples en langage C.

L'annexe B montre l'utilisation de différents outils qui, associés, permettent de créer un compilateur polyédrique complet.

1.3 Contributions

Les contributions principales présentées de ce document sont les suivantes :

— Dans la représentation polyédrique, nous formalisons un ensemble de conditions sur les relations d'ordonnancement. Ces conditions sont nécessaires pour nos approches, sans perdre en généricité.

— Notre première approche d'adaptation des optimisations introduit les *versions interchangeables*, une classe de programmes multi-versions qui peuvent passer d'une version à une autre sans retour en arrière dans les calculs, et que nous exploitons pour adapter un programme à son environnement d'exécution.

Cette approche a donné lieu à une publication internationale [1] :
Lénaïc Bagnères et Cédric Bastoul. **Switcheable Scheduling for Runtime Adaptation of Optimization** à *Euro-Par'20 International Euro-Par Conference*, Porto, Portugal, Août 2014

— Nous formalisons un ensemble de transformations haut-niveau basées sur le modèle polyédrique. Cet ensemble de transformations est le premier à être complet étant données nos conditions sur les relations d'ordonnancement.

— Notre seconde approche d'adaptation des optimisations utilise cet ensemble de transformations composable et complet. Nous proposons le premier algorithme qui prend en entrée deux relations d'ordonnancement et qui trouve une séquence de transformations permettant de passer d'un ordonnancement à l'autre.

Cet algorithme et la plateforme de transformations polyédriques utilisée ont également donné lieu à une publication internationale [2] :
Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot et Cédric Bastoul. **Opening Polyhedral Compiler's Black Box** à *CGO 2016 - 14th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Barcelone, Espagne, Mars 2016

Chapitre 2

La compilation polyédrique

Sommaire

Le modèle polyédrique permet de représenter certains nids de boucles (boucles imbriquées) sous la forme de polyèdres. Les points entiers contenus dans ces polyèdres correspondent aux instances des instructions. Dans ce modèle on ne réfléchit plus sur les boucles mais sur les instances des instructions elles-mêmes. L'analyse des dépendances des données est précise et autorise donc des restructurations de code complexes. L'utilisation du modèle polyédrique permet d'appliquer automatiquement des transformations de code de haut niveau sur les nids de boucles afin, par exemple, d'améliorer la localité des données [3, 4, 5, 6, 7], de les vectoriser [8, 9, 10, 11] et/ou de les paralléliser [5, 4, 12].

L'utilisation classique de la plateforme de compilation polyédrique se fait en trois étapes. La première est la conversion des noyaux de calculs du programme dans le modèle polyédrique. Cette opération peut être faite depuis le code source avec des outils comme *Clan* [13] ou *pet* [14], ou depuis la réprésentation interne des compilateurs, le plus souvent un *arbre syntaxique abstrait* (*Abstract Syntax Tree* en anglais, abrégé *AST*) [15]. La seconde étape est la transformation du code dans le modèle polyédrique. Un exemple classique de cette étape est l'analyse des dépendances avec *Candl* [16] et l'appel à un optimiseur polyédrique comme *Pluto* [5]. La troisième et dernière étape est la génération de code depuis la représentation polyédrique avec des outils comme *CLooG* [17] ou *isl codegen* [18, 19]. Ces étapes peuvent être faites automatiquement, le développeur n'a pas nécessairement à intéragir avec la plateforme de compilation polyédrique.

La plupart des compilateurs de production peuvent utiliser en interne le modèle polyédrique ; c'est le cas de GCC [20], LLVM [11] et IBM XL [21]. Dans le cas de GCC, la plateforme de compilation polyédrique embarquée se nomme *Graphite*[1] [22, 23]. Ce dernier utilise isl [24]. Graphite permet de représenter GIMPLE[2] sous forme de polyèdres. GIMPLE est le jeu d'instructions utilisé par GCC comme représentation interne.

Par défaut, GCC n'utilise pas cette plateforme, ni même avec les options `-O3` et `-Ofast` connues pour activer la grande majorité des optimisations les plus agressives. Pour profiter des optimisations polyédriques, GCC 6 demande l'option `-floop-nest-optimize` permettant d'activer les optimisations des

1. https ://gcc.gnu.org/wiki/Graphite
2. https ://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html

nids de boucles. Actuellement, cette option est notée comme expérimentale. Il est également possible de profiter de l'analyse des dépendances offerte par le modèle polyédrique et de paralléliser les boucles identifiées comme parallélisables. Cette opération se fait dans GCC grâce à l'option `-floop-parallelize-all`.

Les techniques utilisant le modèle polyédrique ont souvent une complexité élevée. L'analyse des dépendances, le calcul des transformations et la génération de code peuvent montrer une complexité exponentielle dans le pire cas [16]. En pratique, ces techniques prennent un temps raisonnable pour les codes ciblés par les techniques polyédriques. Ces codes sont des noyaux de calculs de 10 à 20 lignes avec quelques boucles imbriquées dont la profondeur dépasse rarement 3 ou 4. Les PolyBench/C [3] en sont de bons exemples. Ils correspondent à des codes source simples sans optimisation préalable. Malgré cette réalité, l'utilisation du modèle polyédrique par défaut dans les compilateurs de production n'est pas encore effective même si des avancées ont été faites. Par exemple, un algorithme rapide permettant de détecter les portions de code transformables dans le modèle polyédrique a été implémenté dans GCC 6.0 [25]. Ce travail poursuit les améliorations faites sur la plateforme de compilation polyédrique intégrée à GCC : par exemple, en 2013, la version 4.8 de GCC améliorait déjà significativement Graphite en lui offrant un vrai optimiseur polyédrique basé sur l'algorithme de Pluto [4].

Ce chapitre détaille les différents composants de la plateforme de compilation polyédrique : sa représentation interne, l'extraction depuis un code source, une collection d'outils d'analyse, de transformations et d'optimisations et enfin le générateur de code depuis la représentation polyédrique.

2.1 Exemple

Le code source en langage C [5] de la figure 2.1 correspond au noyau de calcul `jacobi-2d-imper` venant des PolyBench/C 3.2. Il y a deux instructions `S1` et `S2` (lignes 7 et 15). Ces instructions sont toutes les deux dans des boucles imbriquées de profondeur trois et partagent la boucle la plus externe.

```
for (t = 0; t < T_STEPS; t++)                                                    1
{                                                                                2
    for (i = 1; i < N - 1; i++)                                                  3
    {                                                                            4
        for (j = 1; j < N - 1; j++)                                              5
        {                                                                        6
/* S1 */    B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j]);  7
        }                                                                        8
    }                                                                            9
                                                                                 10
    for (i = 1; i < N - 1; i++)                                                  11
    {                                                                            12
        for (j = 1; j < N - 1; j++)                                              13
        {                                                                        14
/* S2 */    A[i][j] = B[i][j];                                                   15
        }                                                                        16
    }                                                                            17
}                                                                                18
```

Figure 2.1 – Code source en langage C du noyau de calcul `jacobi-2d-imper` venant des PolyBench/C 3.2

Dans le code séquentiel de la figure 2.1, on peut naïvement paralléliser les deux boucles internes i (lignes 5 et 11) avec OpenMP. Malheureusement, le programme ne s'exécute pas plus rapidement dans le cas considéré : la taille des données utilisée est `T_STEPS = 20` et `N = 2000` ; le processeur est un quadcore d'Intel `Q6600` cadencé à 2.4 GHz. Pour obtenir du gain, il faut améliorer la localité des données, c'est-à-dire, qu'il faut réutiliser les données qui viennent d'être chargées depuis la mémoire dans la mémoire cache. Comme dans la plupart des opérations point à point, le problème est limité par la bande

3. http://web.cs.ucla.edu/~pouchet/software/polybench/
4. https://gcc.gnu.org/wiki/Graphite-4.8
5. Pour le vocabulaire technique sur les codes source (instruction, boucle, branchement, tableaux), voir l'annexe A sur la programmation en langage C

passante mémoire et seule une amélioration des accès mémoire peut apporter de meilleures performances. L'optimiseur polyédrique Pluto permet de paralléliser automatiquement mais aussi d'améliorer la localité temporelle : le code suivant, optimisé par Pluto, est deux fois plus rapide que le code original et quatre fois plus rapide si on utilise deux cœurs ou plus pour la configuration utilisée.

```
    for (t1=N-1; t1<=3*T_STEPS-2; t1++) {                                          1
      if ((2*t1+1)%3 == 0) {                                                       2
        for (t3=ceild(2*t1+1,3); t3<=floord(2*t1+3*N-8,3); t3++) {                 3
          B[1][((-2*t1+3*t3+2)/3)] = 0.2 * (A[1][((-2*t1+3*t3+2)/3)] +             4
    A[1][((-2*t1+3*t3+2)/3)-1] + A[1][1+((-2*t1+3*t3+2)/3)] +
    A[1+1][((-2*t1+3*t3+2)/3)] + A[1 -1][((-2*t1+3*t3+2)/3)]);;
        }                                                                          5
      }                                                                            6
      lbp=ceild(2*t1+2,3);                                                         7
      ubp=floord(2*t1+N-2,3);                                                      8
#pragma omp parallel for private(lbv,ubv,t3)                                       9
      for (t2=lbp; t2<=ubp; t2++) {                                                10
        B[(-2*t1+3*t2)][1] = 0.2 * (A[(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)][1 -1] +   11
    A[(-2*t1+3*t2)][1+1] + A[1+(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)-1][1]);;
        for (t3=2*t1-2*t2+2; t3<=2*t1-2*t2+N-2; t3++) {                            12
          B[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] = 0.2 * (A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)]  13
    + A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)-1] + A[(-2*t1+3*t2)][1+(-2*t1+2*t2+t3)] +
    A[1+(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)-1][(-2*t1+2*t2+t3)]);;
          A[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)] =                                   14
    B[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)];;
        }                                                                          15
        A[(-2*t1+3*t2-1)][(N-2)] = B[(-2*t1+3*t2-1)][(N-2)];;                      16
      }                                                                            17
      if ((2*t1+N+2)%3 == 0) {                                                     18
        for (t3=ceild(2*t1-2*N+8,3); t3<=floord(2*t1+N-1,3); t3++) {               19
          A[(N-2)][((-2*t1+3*t3+2*N-5)/3)] = B[(N-2)][((-2*t1+3*t3+2*N-5)/3)];;    20
        }                                                                          21
      }                                                                            22
    }                                                                              23
```

Figure 2.2 – Extrait du code source en langage C du noyau de calcul `jacobi-2d-imper` venant des PolyBench/C 3.2, généré par CLooG 0.18.1, après optimisation par Pluto 0.11.4 avec l'option `--parallel`

Le code source de la figure 2.2 est un extrait du programme résultat optimisé par Pluto et généré par CLooG. La version complète est disponible dans la figure B.1 de l'annexe B. Si on compte le nombre de caractères de cette version, le code généré est plus de seize fois plus grand : il passe de 285 à 4693 caractères et de 12 lignes à 96.

Pour arriver à un tel résultat, on a récupéré la représentation polyédrique depuis le code source original. Ensuite, on a appliqué l'algorithme d'optimisation de Pluto. Cet algorithme exprime les contraintes nécessaires afin de paralléliser et d'optimiser la localité en utilisant une plateforme de transformations affines. Pour terminer, on génère le code source C depuis la représentation polyédrique avec CLooG. Il s'agit donc d'un exemple complet de l'utilisation d'un compilateur source à source utilisant la plateforme de compilation polyédrique. La section suivante explique la représentation polyédrique.

Si on regarde plus attentivement l'extrait du code généré, on remarque qu'une boucle a été parallélisée avec OpenMP (lignes 9 et 10). Cette boucle exécute une boucle imbriquée (ligne 12) contenant les instructions `S1` et `S2` (lignes 13 et 14). Comme ces instructions sont dans la même boucle interne, Pluto a réussit à fusionner les boucles originales correspondantes (lignes 5 et 13 du code original dans la figure 2.1). Pour effectuer cette fusion sans violer les dépendances, et vu les accès mémoire mémoire des tableaux `A` et `B`, Pluto a appliqué des transformations qui ressemblent à *l'inclinaison et au déplacement des instances des instructions*. Ces deux transformations, issues des transformations classiques de boucles, sont bien connues et sont détaillées dans le chapitre 5. Cependant, de par la complexité de ce code source, il est difficile de comprendre la nature exacte des transformations faites par Pluto.

Cette version correspond à l'optimisation faite par Pluto avec l'option `--parallel` permettant à l'utilisateur de demander un code parallèle. Avec les options `--tile` et `--parallel`, Pluto *tuile* le code et le parallélise. Avec ces options, l'algorithme de Pluto trouve les bonnes façons d'appliquer un tuilage afin de, ici aussi, paralléliser et optimiser la localité. Ce code fait 9695 caractères pour 163 lignes. Il est deux fois plus grand que la première version de Pluto et trente-quatre fois plus grand que la version originale. Le code complet est disponible dans la figure B.2 de l'annexe B.

2.2 Le modèle polyédrique

Les représentations polyédriques les plus avancées utilisent un seul élément mathématique pour encoder l'ensemble des informations : les *unions de relations* [24] [6].

Un noyau de calcul peut être composé de plusieurs instructions. Ces instructions peuvent être dans des nids de boucles. Pour représenter un noyau de calcul dans le format polyédrique, chaque instruction a besoin de plusieurs relations : le *domaine d'itération* (ou *espace d'itération*), la *relation d'ordonnancement*, et les *relations d'accès*. Le domaine d'itération représente l'ensemble des instances d'instructions à exécuter. La relation d'ordonnancement indique quand les exécuter en définissant pour chaque instance de l'instruction une date logique multidimensionnelle. Les relations restantes décrivent les accès mémoire en lecture et écriture ; il y a une relation par accès mémoire.

2.2.1 Relation

Le modèle polyédrique utilise un seul objet mathématique : la *relation* polyédrique : une union de *relations basiques*. Une relation basique est une association d'un ensemble de dimensions d'entrée vers un ensemble de dimensions de sortie. Elle permet de représenter des contraintes affines. Ces expressions affines sont exprimées en fonction : de coefficients sur les dimensions d'entrée, de coefficients sur les dimensions de sortie, de coefficients sur les dimensions locales, de coefficients sur les paramètres et de constantes. Dans ce document, les dimensions locales nous permettent *d'espacer les instances de l'instruction*. Une instance de l'instruction correspond à une exécution de l'instruction. Les paramètres correspondent à des entiers dont la valeur est inconnue mais constante pour le noyau de calcul considéré. Les constantes sont des entiers dont la valeur est connue.

Pour définir un espace d'itération, les contraintes affines peuvent uniquement limiter cet espace. Le résultat est un polyèdre obligatoirement convexe ; c'est-à-dire que pour tout segment joignant deux points quelconques du polyèdre, tous les points sur ce segment appartiennent au polyèdre. Des exemples de polyèdre convexes sont représentés dans la figure 2.6. Pour représenter un domaine d'itération qui correspondrait à un polyèdre non convexe, il faut le découper en plusieurs polyèdres convexes. L'objet mathématique utilisé pour cela est obligatoirement une union de relations basiques.

Une relation $\mathcal{R}(\vec{p})$ est une union finie de relations basiques. Elle est définie telle quelle :

$$\mathcal{R}(\vec{p}) = \bigcup_i \mathcal{R}_i(\vec{p})$$

avec :

- $\mathcal{R}(\vec{p})$ la relation
- $\bigcup_i \mathcal{R}_i(\vec{p})$ l'union finie de relations basiques
- $\mathcal{R}_i(\vec{p})$ une relation basique

6. Bien qu'isl distingue les *set* et *map*, on peut trouver à la ligne 13 du fichier `isl_map_private.h` le code suivant : `#define isl_basic_set isl_basic_map` (isl 0.17.1)

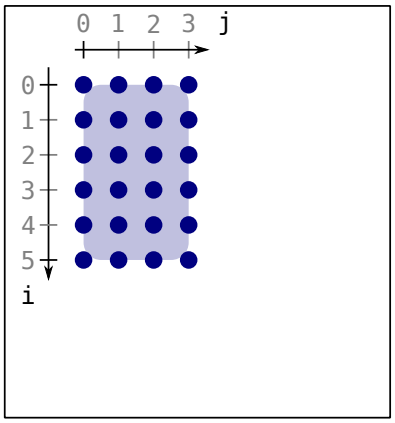


Figure 2.3 – Polyèdre convexe

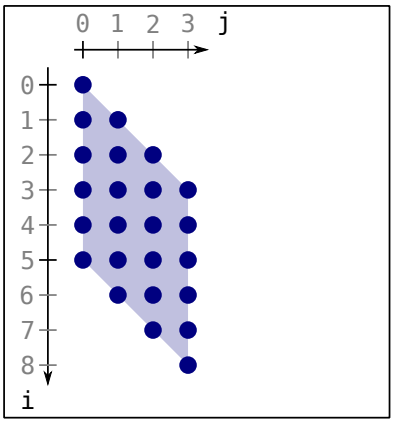


Figure 2.4 – Polyèdre convexe

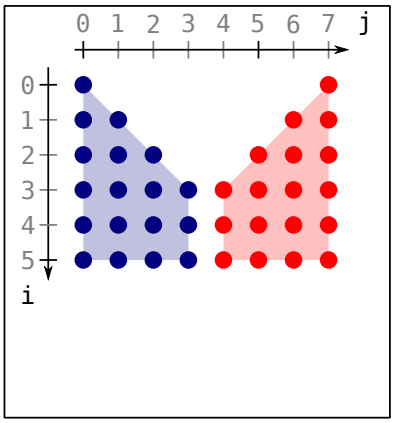


Figure 2.5 – Polyèdres convexes

Figure 2.6 – Polyèdres convexes et union de polyèdres convexes

Une relation basique $\mathcal{R}_i(\vec{p})$ est définie telle quelle :

$$\mathcal{R}_i(\vec{p}) = \left\{ \vec{x}_{in} \rightarrow \vec{x}_{out} \in \mathbb{Z}^{dim(\vec{x}_{in})} \times \mathbb{Z}^{dim(\vec{x}_{out})} \;\middle|\; \exists \vec{l_i} \in \mathbb{Z}^{dim(\vec{l}_i)} : [A_{out,i}, Ain, i, L_i, P_i, \vec{c}_i] \begin{pmatrix} \vec{x}_{out} \\ \vec{x}_{in} \\ \vec{l}_i \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\}$$

avec :

- $\vec{x}_{in} \in \mathbb{Z}^{dim(\vec{x}_{in})}$ les dimensions d'entrée
- $\vec{x}_{out} \in \mathbb{Z}^{dim(\vec{x}_{out})}$ les dimensions de sortie
- $\vec{l}_i \in \mathbb{Z}^{dim(\vec{l}_i)}$ le vecteur de variables locales
- $\vec{p} \in \mathbb{Z}^{dim(\vec{p})}$ le vecteur des paramètres
- $\vec{c}_i \in \mathbb{Z}^{dim(\vec{d}_m)}$ le vecteur de constantes
- $A_{out,i} \in \mathbb{Z}^{m \times dim(\vec{x}_{out})}$, $A_{in,i} \in \mathbb{Z}^{m \times dim(\vec{x}_{in})}$, $L_i \in \mathbb{Z}^{m \times dim(\vec{l}_i)}$, $P_i \in \mathbb{Z}^{m \times dim(\vec{p})}$ sont des matrices d'entiers

L'abstraction polyédrique de chaque instruction contient les informations suivantes : le domaine d'itération, la relation d'ordonnancement et les relations d'accès en lecture et écriture dans chaque variable ou case de tableaux. En plus de cette abstraction, on peut aussi accéder aux informations propres à l'instruction mais non utiles à la représentation polyédrique (le corps de l'instruction par exemple). Pour simplifier la lecture des relations, on utilisera la notation mathématique suivante dans les exemples :

$$\mathcal{R}(\text{Paramètres}) = \left\{ \begin{pmatrix} \textit{vecteur des} \\ \textit{dimensions} \\ \textit{d'entrée} \end{pmatrix} \rightarrow \begin{pmatrix} \textit{vecteur des} \\ \textit{dimensions} \\ \textit{de sortie} \end{pmatrix} \;\middle|\; \begin{matrix} \textit{équations} \\ \textit{ou} \\ \textit{inéquations} \end{matrix} \right\}$$

2.2.2 Domaine d'itération

Grâce aux structures de contrôle comme les boucles, une instruction peut être exécutée plusieurs fois. Les conditions des boucles sont extraites et combinées dans un ensemble d'équations et d'inéquations linéaires qui définissent un polyèdre. Chaque point entier de ce polyèdre correspond à une instance de l'instruction. Cette relation est le domaine d'itération.

Par exemple, la relation suivante correspond au domaine d'itération de l'instruction `S1` du code `jacobi-2d-imper`. Il s'agit d'un polyèdre convexe en trois dimensions :

$$\mathcal{D}_{S1}(\text{T\_STEPS}, \text{N}) = \left\{ () \to \begin{pmatrix} t \\ i \\ j \end{pmatrix} \middle| \begin{array}{l} 0 \leq t < \text{T\_STEPS} \\ 1 \leq i < \text{N} - 1 \\ 1 \leq j < \text{N} - 1 \end{array} \right\}$$

Le domaine d'itération de `S2` est identique à celui de `S1`.

Les relations qui décrivent des domaines d'itération ont une forme particulière. Les dimensions d'entrée sont toujours absentes.

La relation suivante, la relation d'ordonnancement, décrit comment sont ordonnancées les instances de l'instruction. Dans ce document, notre relation d'ordonnancement respecte une forme particulière décrite dans la section suivante. Les autres formes ne sont pas considérées.

2.2.3 Ordonnancement des instances de l'instruction

La relation d'ordonnancement décrit l'ordre d'exécution et le placement des instances de l'instruction. Chaque instance de l'instruction provenant du domaine d'itération est liée à une date logique multidimensionnelle. Il s'agit d'une date composée de plusieurs dimensions qui sont de moins en moins significatives. Par exemple, les horloges définissent une date multidimensionnelle. Les jours correspondent à la première dimension, la plus significative dans notre exemple. Les heures correspondent à la seconde dimension, moins significative que la première. Ensuite viennent les minutes puis les secondes, etc.

Cet ordonnancement est défini par les (in)équations portant sur les dimensions d'entrée et les dimensions de sortie. L'ordre d'exécution des instances de l'instruction suit l'ordre lexicographique des dimensions définissant les dates logiques. Lorsque le nombre de dimensions de sortie du domaine d'itération est égal au nombre de dimensions d'entrée de la relation d'ordonnancement, on dit que le domaine d'itération et l'ordonnancement sont *d-compatible*s.

La relation qui suit l'ordre original donné par le code si l'on ne considère qu'une seule instruction est une association directe entre les dimensions d'entrée et celles de sortie D'autres relations d'ordonnancement sont possibles afin d'effectuer des transformations de code plus ou moins complexes.

Dimensions β. Des nouvelles dimensions de sortie, dites dimensions β, sont introduites pour respecter l'ordre des instructions entre elles. Les autres dimensions de sortie, les dimensions α, correspondent aux boucles d'itération. On alterne une dimension β avec une dimension α en commençant et en finissant par une dimension β. La numérotation des dimensions α commence à 1. Les dimensions impaires sont des dimensions β et les dimensions paires sont des dimensions α. Cette structure particulière a été suggérée par Paul Feautrier [26] et est utilisée dans plusieurs plateformes de compilation polyédriques [27, 28, 16, 2].

Par exemple, la relation suivante correspond à l'ordonnancement des instances de l'instruction `S1` du code `jacobi-2d-imper` qui suit l'ordre original défini par les boucles dans le code source :

$$\theta_{S1}(\text{T\_STEPS}, \text{N}) = \left\{ \begin{pmatrix} t \\ i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \\ \alpha_3 \\ \beta_3 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & t \\ \beta_1 & = & 0 \\ \alpha_2 & = & i \\ \beta_2 & = & 0 \\ \alpha_3 & = & j \\ \beta_3 & = & 0 \end{array} \right\}$$

L'ordonnancement des instances de l'instruction `S2` du code `jacobi-2d-imper` qui suit l'ordre original définit par les boucles est donné dans la relation ci-dessous. La valeur de β_1 de `S2` est supérieure à celle de `S1` ; l'exécution de `S2` se fera après celle de `S1`.

$$\theta_{S2}(\text{T\_STEPS}, \text{N}) = \left\{ \begin{pmatrix} t \\ i \\ j \end{pmatrix} \rightarrow \left(\begin{array}{c|rcl} \beta_0 & \beta_0 & = & 0 \\ \alpha_1 & \alpha_1 & = & t \\ \beta_1 & \beta_1 & = & \mathbf{1} \\ \alpha_2 & \alpha_2 & = & i \\ \beta_2 & \beta_2 & = & 0 \\ \alpha_3 & \alpha_3 & = & j \\ \beta_3 & \beta_3 & = & 0 \end{array} \right) \right\}$$

Le *β-vecteur* d'une relation d'ordonnancement est le vecteur composé des valeurs des dimensions β. Dans notre exemple, le β-vecteur de `S1` est $(0, 0, 0, 0)$ et celui de `S2` est $(0, 1, 0, 0)$. Sous la forme que nous avons choisie pour la relation d'ordonnancement, chaque relation d'ordonnancement a un β-vecteur. Les valeurs des β-vecteurs sont supérieures ou égales à 0 et se suivent. Les β-vecteurs sont uniques, c'est-à-dire qu'il n'existe pas deux β-vecteur de même taille avec toutes les valeurs respectivement égales. Un β-vecteur ne peut pas être égal au préfixe d'un autre. Si deux β-vecteurs ont le même préfixe, les instructions correspondantes peuvent partager une ou plusieurs boucles d'itération. Même si la notion de boucle n'apparaît pas dans le modèle polyédrique, par abus de langage, on parle de *β-boucle* pour désigner le préfixe d'un β-vecteur. Par exemple, le préfixe le plus long du β-vecteur de `S2` est $(0, 1, 0)$. Ce β-boucle correspond à la boucle la plus interne de `S2` (ligne 13). La β-boucle (0) est un préfixe du β-vecteur de `S1` ainsi que du β-vecteur de `S2`. Les instructions `S1` et `S2` partagent donc une boucle : il s'agit de la boucle la plus externe (ligne 1). La β-boucle vide est valide et correspond à la racine du programme.

Modifier la relation d'ordonnancement, correspond à appliquer une transformation polyédrique. Par exemple, l'ordonnancement ci-dessous suit l'ordre inverse de la boucle la plus interne. Cette transformation est très simple, il suffit d'indiquer que la dimension de sortie α_3 décrite en fonction de la dimension d'entrée j correspondante à la boucle la plus interne ne suit plus j mais l'ordre inverse, c'est-à-dire $-j$.

$$\theta_{S1}(\text{T\_STEPS}, \text{N}) = \left\{ \begin{pmatrix} t \\ i \\ j \end{pmatrix} \rightarrow \left(\begin{array}{c|rcl} \beta_0 & \beta_0 & = & 0 \\ \alpha_1 & \alpha_1 & = & t \\ \beta_1 & \beta_1 & = & 0 \\ \alpha_2 & \alpha_2 & = & i \\ \beta_2 & \beta_2 & = & 0 \\ \alpha_3 & \alpha_3 & = & \boldsymbol{-j} \\ \beta_3 & \beta_3 & = & 0 \end{array} \right) \right\}$$

Par definition, *toutes* les transformations polyédriques peuvent s'exprimer en modifiant uniquement les relations d'ordonnancement des instances des instructions. D'autres méthodes sont néanmoins possibles. Traditionnellement, les optimiseurs polyédriques comme Pluto, modifient le domaine d'itération afin d'effectuer des transformations comme le tuilage ou le découpage d'ensembles d'instances de l'instruction qui sépare le domaine d'itération en plusieurs relations basiques comme dans l'exemple de la figure 2.5. Ce découpage permet d'appliquer des transformations spécifiques sur une sous-partie des instances de l'instruction. Ces transformations peuvent donc aussi s'effectuer dans le domaine d'itération. Modifier le domaine d'itération n'est plus nécessaire grâce à l'utilisation des unions de relations. Modifier le domaine d'itération amène un problème de sémantique : le rôle du domaine d'itération, comme son nom l'indique, doit se limiter à définir les instances de l'instruction. De plus, modifier le domaine d'itération rend plus complexes certaines opérations polyédriques comme la vérification des dépendances. Il s'agit d'un choix de conception utilisé tout au long de ce document.

Les dimensions de sortie de la relation d'ordonnancement peuvent être définies *explicitement* ou *implicitement*. On parle de dimensions *explicites* et de dimensions *implicites*. Dans nos exemples précédents, les dimensions sont toutes explicites.

Dimension explicite. Une dimension de sortie est explicite si elle est définie grâce à une seule équation. Cette équation porte généralement sur les dimensions d'entrée, les paramètres et la constante mais peut aussi utiliser les dimensions de sortie.

Dimension implicite. Les dimensions d'entrée définies implicitement sont autorisées si elles sont bornées grâce à des inéquations.

Pour les relations basiques, on définit les termes de *forme d'entrée* et de *forme de sortie*.

Forme d'entrée. La forme d'entrée d'une relation basique est la forme telle que les dimensions d'entrée sont exprimées en fonctions des dimensions de sortie, des paramètres et de la constante. La valeur des coefficients des dimensions d'entrée est de 1 lors de leur définition.

Forme de sortie. Dans la forme de sortie, ce sont les dimensions de sortie qui sont définies explicitement. Les dimensions de sortie sont exprimées en fonctions des dimensions d'entrée, des paramètres et de la constante. Comme il est possible d'avoir un nombre de dimensions de sortie plus grand que le nombre de dimensions d'entrée, certaines dimensions de sortie sont définies implicitement grâce à des inéquations qui les bornent. La valeur des coefficients des dimensions de sortie est de 1 lors de leur définition.

On définit les termes de *validité*, de *validité globale* et de *validité conditionnelle* ainsi que les différentes contraintes pour atteindre la condition de validité globale : « *d-compatibles* », « *Existence de l'ordonnancement* », « *Date logique entière* » et « *Date logique unique* ».

Validité. On dit qu'une relation d'ordonnancement est valide si elle respecte les conditions suivantes :
— la relation d'ordonnancement des instances de l'instruction est *d-compatible* avec son domaine d'itération respectif
— chaque point entier du domaine d'itération est ordonnancé, on parle de l'*existence de l'ordonnancement*
— chaque date logique définie par l'ordonnancement est entière, on parle de *date logique entière*
— chaque date logique définie par l'ordonnancement est unique, on parle de *date logique unique*
Si l'une de ces contraintes n'est pas respectée, la relation d'ordonnancement est *invalide*.

d-compatibles. Pour que la relation d'ordonnancement et le domaine d'itération soient d-compatibles, le nombre de dimensions de sortie du domaine doit être égal au nombre de dimensions d'entrée de la relation d'ordonnancement.

Existence de l'ordonnancement. L'existence de l'ordonnancement permet de s'assurer que chaque point entier du domaine est ordonnancé. Comme la relation d'ordonnancement est en fait une union de relations basiques, il est possible d'appliquer un ordonnancement différent à différentes parties regroupant différentes instances du domaine d'itération. Pour que l'existence de l'ordonnancement soit respectée, il est impératif que les différentes relations basiques de l'union de la relation d'ordonnancement ordonnent tous les points entiers définis par le domaine d'itération.

Date logique entière. L'ordonnancement doit générer une date logique entière pour chaque dimension. Si la relation basique est sous la forme de sortie et ne contient que des coefficients entiers, alors il est garanti que l'ordonnancement génère une date logique entière pour chaque dimension.

Date logique unique. L'ordonnancement doit générer une et une seule date logique unique pour chaque instance de l'instruction. Comme chaque β-vecteur est unique, cela permet de limiter le problème entre les différentes instances d'une même instruction. Pour respecter entièrement cette condition, il faut que chaque dimension d'entrée soit utilisée lorsqu'on définit les dimensions α. Il n'est pas nécessaire de se soucier des autres dimensions ou du domaine d'itération.

Validité globale. Une relation d'ordonnancement valide respecte la condition de validité globale si elle peut ordonnancer n'importe quel domaine d'itération d-compatible tout en restant valide. Par définition, on sait qu'une relation d'ordonnancement respectant la condition de validité globale ordonnance tous les points entiers définis par le domaine d'itération une et une seule fois à une date entière unique quel que soit le domaine d-compatible.

Validité conditionnelle. Dans le cas contraire, si la relation d'ordonnancement est valide mais pour des domaines d'itération particuliers, la relation d'ordonnancement a une validité conditionnelle.

Voici un exemple simple de validité conditonnelle et de validité globale. Soit l'ordonnancement suivant θ_S permettant d'ordonnancer les instances des instructions provenant d'un domaine d'itération avec deux dimensions de sortie. Les dimensions de sortie β sont enlevées pour simplifier la relation.

$$\theta_S(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \middle| \begin{array}{lcl} \alpha_1 & = & i \\ \alpha_2 & = & j \end{array} \right\}$$

Cet ordonnancement ne fait aucune supposition sur le domaine d'itération. Le domaine d'itération suivant $\mathcal{D}_S$ peut donc être ordonnancé par θ_S dans le respect de la validité globale.

$$\mathcal{D}_S(\mathrm{N},\mathrm{M}) = \left\{ () \rightarrow \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{array}{l} 1 \leq i < \mathrm{N} \\ 1 \leq j < \mathrm{M} \end{array} \right\}$$

Il en est de même pour le domaine d'itération $\mathcal{D}'_S$ suivant où j est compris entre 1 et 32 inclus.

$$\mathcal{D}'_S(\mathrm{N},\mathrm{M}) = \left\{ () \rightarrow \begin{pmatrix} i \\ j \end{pmatrix} \middle| \begin{array}{l} 1 \leq i < \mathrm{N} \\ 1 \leq j < 32 \end{array} \right\}$$

Le découpage des instances de l'instruction est une transformation inspirée des transformations classiques de boucle. Elle permet de séparer les instances des instructions en plusieurs parties (deux dans notre exemple). Une fois effectuée, elle permet d'appliquer des transformations différentes sur les différentes parties. Dans la représentation polyédrique, elle se traduit par une relation définie par plusieurs relations basiques. Dans notre exemple, on définit un nouvel ordonnancement θ'_S.

$$\theta'_S(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \middle| \begin{array}{lcl} \alpha_1 & = & i \\ \alpha_2 & = & j \\ j & \leq & 32 \end{array} \right\} \bigcup \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \middle| \begin{array}{lcl} \alpha_1 & = & i \\ \alpha_2 & = & j \\ j & > & 32 \end{array} \right\}$$

Cet ordonnancement est composé de deux relations basiques. La première ordonnance les instances dont la dimension d'entrée j est comprise entre 1 et 32 inclus. La deuxième ordonnance les instances dont la dimension d'entrée j est inclus entre 33 et M. θ'_S respecte la condition de validité globale car, grâce à ces deux relations basiques, elle ordonnance bien l'ensemble des instances définies dans le domaine d'itération, quel que soit ce domaine d'itération.
On définit l'ordonnancement θ''_S qui est uniquement composé de la première relation basique de θ'_S.

$$\theta''_S(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \alpha_1 \\ \alpha_2 \end{pmatrix} \middle| \begin{array}{lcl} \alpha_1 & = & i \\ \alpha_2 & = & j \\ j & \leq & 32 \end{array} \right\}$$

Cette relation ordonnance uniquement les instances qui satisfont la contrainte $j \leq 32$. Cet ordonnancement ne respecte donc pas la condition de validité globale car les instances qui satisfont la contrainte $j > 32$ ne sont pas ordonnancées par θ''_S.
En revanche, θ''_S a une validité conditionnelle. Il ordonnance bien toutes les instances définies par $\mathcal{D}'_S$ car toutes ces instances satisfont la contrainte $j \leq 32$.

Équivalence & Identicité (ou Égalité stricte). Deux ordonnancements sont dits *équivalents* si l'ordre d'exécution relatif des instances de l'instruction est le même. Pour que deux ordonnancements soient *identiques*, les deux relations doivent avoir exactement les mêmes valeurs modulo les transformations mathématiques suivantes : l'élimination de Gauss-Jordan et la simplification des (in)équations. Si deux ordonnancements sont identiques, on peut s'attendre à ce que le générateur de code génère exactement le même code pour ces deux ordonnancements. En pratique, selon les options passées au générateur, il peut générer deux codes différents, mais ces codes doivent exécuter les instances des instructions dans le même ordre. En revanche, s'ils sont uniquement équivalents mais pas identiques, le générateur de code est libre de générer deux codes différents. Ces deux codes sont différents mais font le même calcul, exactement dans le même ordre et ont donc la même sortie, même si le calcul ne se fait pas forcément de la même façon. Pour passer facilement d'un ordonnancement équivalent à un autre, on applique une séquence de transformations qui assure la *validité*, la *légalité* et l'équivalence des ordonnancements. Seule la relation d'ordonnancement est modifiée. Par exemple la *normalisation des β-vecteurs* est une transformation d'équivalence qui minimise les valeurs des β-vecteurs sans affecter les dimensions α. Les valeurs des β-vecteurs normalisées sont celles décrites précédemment. Toute transformation ayant pour résultat un ordonnancement identique est autorisée. Ces transformations peuvent être syntaxiques ; par exemple *stripminer* une dimension (décomposer une boucle en deux boucles, cela correspond à un tuilage sur une seule dimension) ne change pas l'ordre d'exécution des instances de l'instruction. Toute transformation ou séquence de transformations ne changeant pas l'ordre relatif des instances des instructions, entre elles, est donc autorisée dans la construction d'un ordonnancement équivalent.

Informations sémantiques. Certaines transformations ne sont pas représentables dans le modèle polyédrique, par exemple, le déroulage de boucle et les annotations OpenMP permettant de paralléliser les boucles `for`. Pour contourner ce problème, des informations sémantiques peuvent être portées par les dimensions de sortie. Ces ajouts permettent de transporter des informations entre les différentes étapes de la compilation. Dans certains cas, on pourrait encoder les informations dans la représentation polyédrique. Par exemple, si on révoque notre choix sur l'unicité des β-vecteurs, on pourrait exprimer du parallélisme avec deux β-vecteurs identiques. Un autre exemple : en dupliquant les instructions et en changeant le domaine, on pourrait exprimer un déroulage de boucle. Ces solutions restent hasardeuses car elles dépendent fortement du générateur de code. De plus elles complexifient les outils polyédriques. Par exemple, le déroulage de boucle effectué en dupliquant les instructions quitte le modèle polyédrique classique lorsque le facteur de déroulage n'est pas un diviseur entier de la borne supérieure de l'indice de la boucle à dérouler. L'ajout des informations sémantiques sur les dimensions de sortie est donc la bonne solution. Par exemple, on peut qualifier les dimensions de sortie de parallèles, vectorisables, déroulées. Pour ces exemples, ces informations sont destinées au générateur de code qui générera le code demandé.

Dans ce document, on ne s'intéresse qu'aux relations d'ordonnancement qui respectent la condition de validité globale.

2.2.4 Accès mémoire

Les relations d'accès terminent l'abstraction polyédrique des instructions : les accès aux variables et aux tableaux, en lecture et en écriture. Ces accès permettent de calculer précisément les dépendances entre les différentes instances de l'instruction. Plus la description des dépendances est précise, plus le compilateur aura la possibilité d'effectuer des transformations agressives.

Les dimensions de sortie d'une relation d'accès correspondent au nom de la variable ou du tableau auquel s'ajoute, dans le cas où c'est un tableau, aux différentes dimensions de ce tableau. Les dimensions d'entrée correspondent aux dimensions de sortie du domaine d'itération. Le nom de la variable ou du tableau est considéré comme une dimension et les variables sont considérées comme des tableaux sans dimension supplémentaire.

Par exemple, les relations suivantes correspondent aux accès mémoires de l'instruction `S2` du code `jacobi-2d-imper` :

- en lecture :

$$\mathcal{A}_{S2,1}(\text{T\_STEPS}, \text{N}) = \left\{ \begin{pmatrix} B \\ i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} Array \\ dim_1 \\ dim_2 \end{pmatrix} \middle| \begin{array}{lcl} Array & = & B \\ dim_1 & = & i \\ dim_2 & = & j \end{array} \right\}$$

- en écriture :

$$\mathcal{A}_{S2,2}(\text{T\_STEPS}, \text{N}) = \left\{ \begin{pmatrix} A \\ i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} Array \\ dim_1 \\ dim_2 \end{pmatrix} \middle| \begin{array}{lcl} Array & = & A \\ dim_1 & = & i \\ dim_2 & = & j \end{array} \right\}$$

On définit le terme de *légalité*.

Légalité. Un ensemble de relations d'ordonnancement est légal par rapport à un autre ensemble de relations d'ordonnancement si le premier ensemble ordonnance les instances des instructions définies dans les domaines d'itération en respectant les dépendances définies dans le second ensemble. Les domaines des instructions sont les mêmes pour les deux ensembles de relations d'ordonnancement. Les relations d'accès sont nécessaires pour calculer les dépendances. Il ne faut pas confondre la légalité avec la validité.

2.3 Le format OpenScop

Avec ces différentes relations : la relation du domaine d'itération, la relation de l'ordonnancement des instances de l'instruction et les relations d'accès, on peut représenter une instruction dans le modèle polyédrique. La section suivante présente le format *OpenScop*. Ce format d'échange permet d'encoder la représentation polyédrique d'un code source (contenant plusieurs instructions).

La représentation polyédrique présentée peut être encodée au format *OpenScop* [7] [29]. OpenScop permet de stocker les informations minimales mais nécessaires pour construire une chaîne de compilation polyédrique complète.

Les services fournis par OpenScop sont à la fois pérennes, extensibles et universels.

Pérennes. OpenScop utilise uniquement des unions de relations qui permettent d'encoder les informations nécessaires. Il s'agit du cœur du système, il est stable et il peut être facilement exporté vers d'autres formats comme *ScopLib* (dépréciée) et isl.

Extensibles. OpenScop fournit un système d'extensions extensible et donc personnalisable.

Universels. OpenScop est un format mais pour faciliter son intégration, la bibliothèque *OpenScop Library (osl)* est disponible sous licence libre. osl est écrite en C qui est compatible avec de nombreux autres langages de programmation et n'a aucune dépendance obligatoire vers des bibliothèques externes. osl est multiplateforme.

OpenScop est simple. Le format d'échange est pensé pour être très facilement *parsable* (lue par une machine). Pour un humain, c'est un peu verbeux mais la bibliothèque osl ajoute des commentaires pour mettre les équations et inéquations sous une forme facilement lisible.

```
<OpenScop>                                  1
# Language                                  2
C                                           3
# Context                                   4
CONTEXT                                     5
0 4 0 0 0 2                                 6
# Parameters are provided                   7
1                                           8
<strings> T_STEPS N </strings>              9
```

(a) Début

Figure 2.7 – Extraits du SCoP utilisant le format OpenScop, extrait depuis le code source C `jacobi-2d-imper` venant des PolyBench/C 3.2 par Clan 0.7.1

Dans sa configuration minimale, OpenScop est composé de ces différents éléments permettant de décrire le *ScoP* : le contexte, le langage du code source, les noms des paramètres, les instructions et des extensions. Le contexte est une relation décrivant les éventuelles contraintes sur les paramètres, Pour chaque instruction (*statement* en anglais), on a la relation décrivant le domaine d'itération, la relation décrivant l'ordonnancement des instances de l'instruction, les relations décrivant les accès mémoires et des extensions. Des exemples d'extensions du SCoP sont : `scatnames`, qui donne les noms des dimensions de sortie de l'ordonnancement, `arrays`, qui donne les noms des variables et des tableaux et `coordinates`, qui donne les coordonnées du SCoP dans le code source d'origine. Chaque instruction a une extension `body` décrivant le corps de l'instruction. Ces extensions apparaissent par défaut mais, comme toutes les autres, restent optionnelles.

Des extraits du fichier OpenScop correspondant à la représentation polyédrique de `jacobi-2d-imper` se trouve dans la figure 2.7. La version complète se trouve dans la figure B.3 dans l'annexe B. Ce SCoP a été généré par osl. On remarque qu'il n'y a aucune ambiguïté lors du parsage : soit, on sait quel est l'élément à lire, soit, un entier permet de savoir si l'élément est présent. Par exemple, la ligne 42 indique la présence ou non des paramètres et la ligne 48 donne le nombre d'éléments à parser. Les extensions sont dans des balises inspirées de XML contenant le nom de l'extension comme celles des paramètres des lignes 43 à 45. Les commentaires commencent par le caractère `#` et finissent à la fin de la ligne courante. De nombreux commentaires sont présents pour faciliter la lecture du SCoP par un humain.

7. https://github.com/periscop?tab=repositories

```
DOMAIN                                                                                                  1
8 7 3 0 0 2                                                                                             2
# e/i|  t    i    j |T_S.   N |  1                                                                      3
   1    1    0    0    0    0    0    ## t >= 0                                                        4
   1   -1    0    0    1    0   -1    ## -t+T_STEPS-1 >= 0                                             5
   1    0    0    0    1    0   -1    ## T_STEPS-1 >= 0                                                6
   1    0    1    0    0    0   -1    ## i-1 >= 0                                                      7
   1    0   -1    0    0    1   -2    ## -i+N-2 >= 0                                                   8
   1    0    0    0    0    1   -3    ## N-3 >= 0                                                      9
   1    0    0    1    0    0   -1    ## j-1 >= 0                                                      10
   1    0    0   -1    0    1   -2    ## -j+N-2 >= 0                                                   11
SCATTERING                                                                                              12
7 14 7 3 0 2                                                                                            13
# e/i| c1   c2   c3   c4   c5   c6   c7 |  t    i    j |T_S.   N |  1                                  14
   0   -1    0    0    0    0    0    0    0    0    0    0    0    0    ## c1 == 0                    15
   0    0   -1    0    0    0    0    0    1    0    0    0    0    0    ## c2 == t                    16
   0    0    0   -1    0    0    0    0    0    0    0    0    0    0    ## c3 == 0                    17
   0    0    0    0   -1    0    0    0    0    1    0    0    0    0    ## c4 == i                    18
   0    0    0    0    0   -1    0    0    0    0    0    0    0    0    ## c5 == 0                    19
   0    0    0    0    0    0   -1    0    0    0    1    0    0    0    ## c6 == j                    20
   0    0    0    0    0    0    0   -1    0    0    0    0    0    0    ## c7 == 0                    21
WRITE                                                                                                   22
3 10 3 3 0 2                                                                                            23
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1                                                       24
   0   -1    0    0    0    0    0    0    0    6    ## Arr == B                                      25
   0    0   -1    0    0    1    0    0    0    0    ## [1] == i                                      26
   0    0    0   -1    0    0    1    0    0    0    ## [2] == j                                      27
READ                                                                                                    28
3 10 3 3 0 2                                                                                            29
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1                                                       30
   0   -1    0    0    0    0    0    0    0    7    ## Arr == A                                      31
   0    0   -1    0    0    1    0    0    0    0    ## [1] == i                                      32
   0    0    0   -1    0    0    1    0    0    0    ## [2] == j                                      33
# ...                                                                                                   34
# Number of Statement Extensions                                                                        35
1                                                                                                       36
<body>                                                                                                  37
# ...                                                                                                   38
B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j]);                              39
</body>                                                                                                 40
```

(b) Première instruction

Figure 2.7 – Extraits du SCoP utilisant le format OpenScop, extrait depuis le code source C `jacobi-2d-imper` venant des PolyBench/C 3.2 par Clan 0.7.1

Dans ce format, une relation basique est décrite avec 6 entiers suivis par une matrice contenant uniquement des entiers. Les deux premiers entiers donnent le nombre de lignes et le nombre de colonnes de la matrice, les troisièmes et quatrièmes entiers donnent le nombre de dimensions de sortie et le nombre de dimensions de sortie, l'entier suivant donne le nombre de dimensions locales et le dernier entier donne le nombre de paramètres. Chaque ligne correspond à une équation ou une inéquation. Si la première valeur est 0, il s'agit d'une équation (opérateur égal). Si c'est 1, il s'agit d'une inéquation (opérateur supérieur ou égal à 0). Ensuite viennent les coefficients des dimensions de sortie, puis ceux des dimensions d'entrée. On finit avec les coefficients des paramètres et une constante. L'(in)équation est de la forme *coefficients des dimensions de sortie* × *dimensions de sortie* + *coefficients des dimensions d'entrée* × *dimensions d'entrée* + *coefficients des dimensions locales* × *dimensions locales* + *coefficients des paramètres* × *paramètres* + constante *opérateur* 0. Un exemple, quelque peu intéressant, d'inéquation est à la ligne 143 du SCoP, la ligne `1 -1 0 0 1 0 -1` donne l'inéquation $-\mathtt{t} + \mathtt{T\_STEPS} - 1 \geq 0$. Si notre relation est

composée de plusieurs relations basiques, cette structure se répète autant de fois que nécessaire. Ce format permet d'exprimer n'importe quelle relation polyédrique en tenant compte des limitations du modèle.

```
<scatnames> b0 t b1 i b2 j b3 </scatnames>

<arrays>
# ...
1 t
2 T_STEPS
3 i
4 N
5 j
6 B
7 A
</arrays>

<coordinates>
# File name
# ...
</coordinates>

</OpenScop>
```

(c) Extension du SCoP

Figure 2.7 – Extraits du SCoP utilisant le format OpenScop, extrait depuis le code source C `jacobi-2d-imper` venant des PolyBench/C 3.2 par Clan 0.7.1

Le SCoP donné en exemple a été extrait du code source par l'outil *Clan*. Cet outil permet d'extraire la représentation polyédrique depuis un code source. Clan, et les limitations du modèle polyédrique, sont présentés dans la section suivante.

2.4 Extraction depuis un programme

L'extraction de l'abstraction polyédrique d'un programme peut se faire depuis sa représentation directe, souvent haut-niveau, par exemple son code source, ou depuis sa représentation interne, par exemple l'arbre de syntaxe abstraite.

Clan[8] [13] permet d'extraire la représentation polyédrique au format OpenScop d'un ou plusieurs noyaux de calculs d'un code source écrit en syntaxe C. Clan signifie *Chunky Loop ANalyzer*. Comme Clan se base uniquement sur une syntaxe proche du C, le code n'est pas nécessairement valide en C, ni même nécessairement du code C. Clan fonctionnera dans d'autres langages de programmation comme C++, Java, C#, ... si les boucles et autres structures de contrôle ainsi que les accès mémoires respectent la syntaxe de C.

Clan ne permet pas de représenter n'importe quel code écrit en C, il accepte uniquement un sous ensemble avec les restrictions suivantes :

Restrictions syntaxiques. Clan 0.7.0 accepte uniquement les mots clés `for`, `while`, `if` et `else`. Les déclarations ne sont pas (encore) autorisées. Les accès aux tableaux se font avec l'opérateur `[]` et l'utilisation de l'arithmétique des pointeurs n'est pas acceptée.

Pour les restrictions polyédriques, on s'intéresse particulièrement aux accès de tableaux qui suivent la syntaxe suivante : `tab[`*`indice`*`]`, à la structure conditionnelle `if (`*`condition`*`)` et à la boucle `for (`*`initialisation, condition, pas`*`)`.

8. https://github.com/periscop?tab=repositories

Restriction polyédrique : accès aux tableaux. Une expression affine est la somme de constantes et de plusieurs variables qui peuvent être multipliées par un coefficient constant.
Les indices de tableaux doivent être des expressions affines qui opèrent uniquement sur les indices de boucles, les paramètres et les constantes. Ces expressions affines ne peuvent donc pas être exprimées en fonction des valeurs des variables ou des éléments des tableaux.

Restriction polyédrique : initialisation des boucles. L'initialisation des boucles doit être aussi une expression affine en fonction des indices de boucles, des paramètres et de constantes mais peut aussi inclure l'utilisation des fonctions `max`, (min), `floord`, `ceild`. L'utilisation de `max` et `min` mélangée n'est pas autorisée. Si l'expression utilise `max`, le pas de la boucle doit être positif alors que Si l'expression utilise `min`, le pas de la boucle doit être négatif. La fonction `min(a, b)` renvoie la valeur minimale entre `a` et `b`. La fonction `max(a, b)` renvoie la valeur maximale entre `a` et `b`. La fonction `floord(a, b)` renvoie l'entier inférieur du résultat de la division entre `a` et `b`. La fonction `ceild(a, b)` renvoie l'entier supérieur du résultat de la division entre `a` et `b`.

Restriction polyédrique : condition dans les boucles et les `if`. Les différentes parties de la condition doivent être des expressions affines. L'utilisation des opérateurs C `==`, `!=`, `<`, `<=`, `>`, `>=`, `&&` et `||` est autorisée. Les fonction `min` et `max` peuvent être utilisées avec les opérateurs `<`, `<=`, `>`, `>=`. Si on utilise `min`, l'expression doit être la partie à droite de l'opérateur. Si on utilise `max`, l'expression doit être la partie à gauche de l'opérateur. L'opérateur `%` doit avoir la forme suivante : `e % a == b` avec `e` une expression affine, `x` et `y` deux entiers positifs.

Restriction polyédrique : le pas des boucles. Le pas des boucles peut être uniquement incrémenté ou décrémenté. L'indice de la boucle courante ne peut être modifié uniquement ici. L'incrémentation ou la décrémentation ne peut se faire uniquement par une constante entière.

Restrictions sémantiques. En plus de ces limitations, Clan assume que toutes les fonctions appelées n'ont aucun effet de bord, c'est-à-dire qu'elles ne doivent pas modifier de variables globales. Ces fonctions ne sont pas nécessairement *pures* car aucune restriction n'est faite sur le retour de ces fonctions. Cette limitation est présente pour des raisons de temps de développement. Elle pourrait être facilement levée si Clan avait connaissance des variables ou tableaux modifiés. Cela pourrait être automatiquement détecté ou donné par le développeur. Également pour des raisons de développement, Clan suppose que tous les tableaux utilisés ont leur propres zones mémoires et ne se chevauchent pas avec d'autres tableaux utilisés dans le SCoP. Clan régit comme si tous les tableaux avaient été marqués avec le mot clé `restrict` du langage C.

Ces restrictions ont pour origine les limitations du modèle polyédrique. Certaines pourraient être levées par sur-approximations du programme. Par exemple, si un accès n'est pas une expression affine, on peut indiquer que l'accès au tableau concerné se fait sur l'ensemble de ses valeurs au lieu d'une seule case. Cette approche permet de convertir le programme dans la représentation polyédrique et permet donc d'utiliser la plateforme de compilation polyédrique même si la sur-approximation pourrait limiter sa puissance. Un autre exemple, si une condition ou une boucle ou même un ensemble d'instructions ne correspond pas au modèle polyédrique, on peut créer une “ macro-instruction ” qui contient l'ensemble des instructions non polyédrique. On peut extraire les accès aux tableaux de cette macro-instruction. Cet exemple limitera potentiellement la puissance du modèle polyédrique mais permettra de l'utiliser. D'autres restrictions pourraient être levées en étendant le modèle polyédrique lui-même en autorisant les contraintes et les relations polynômiales [30].

Static vs Affine. On utilise le terme SCoP (*Static Control Parts*) pour désigner les morceaux de code qui peuvent être traduits dans une représentation polyédrique. L'utilisation des mots « contrôle statique » n'est pas tout à fait exacte. En réalité, pour être représenté dans le modèle polyédrique, un code source doit avoir des contrôles affines et non statiques. Par exemple, $i < N \times N$ est statique car $N \times N$ est une constante connue à la compilation, mais cette expression n'est pas affine. À l'inverse, $i < N$ avec N une variable qui ne change pas durant l'exécution du SCoP mais qui n'est pas pas une constante de compilation, n'est pas statique mais est affine. Le terme SCoP est largement utilisé même s'il est erroné par habitude et abus de langage.

PET est un autre extracteur polyédrique. Contrairement à Clan, il supporte le langage C11. Il fonctionne grâce au compilateur LLVM. PET (*Polyhedral Extraction Tool*) extrait la représentation polyédrique depuis l'AST de LLVM. PET a l'avantage de supporter les différentes syntaxe de C11 mais cela est aussi un inconvénient car PET se limite au C11 pendant que Clan supporte n'importe quel langage proche syntaxiquement de C. Clan est plus souple dans la mesure où les modifications rapides sont plus faciles. Par exemple, le mot clé `xfor` [31] a été récemment implémenté dans Clan.

Extraction directe vs Extraction interne. L'extraction depuis la représentation interne du compilateur peut profiter des informations et transformations appliquées aux différentes passes déjà effectuées par ce dernier, par exemple, la détection et la propagation des constantes. Ces informations peuvent permettre l'extraction de la représentation polyédrique de programmes qui ne respectaient pas les contraintes des SCoPs dans leur forme haut-niveau. Au contraire, certaines passes peuvent complexifier la représentation interne et empêcher l'extraction de la représentation polyédrique.
À l'inverse, l'extraction depuis la représentation directe comme le code source, ne profite pas des informations inférées par le compilateur et peut demander au développeur de remanier le code source pour satisfaire les contraintes nécessaires à l'extraction de l'abstraction polyédrique. Cependant cette forme donne la garantie au développeur que son code sera traité par l'optimiseur polyédrique s'il respecte les contraintes et l'autorise à communiquer des informations directement à l'extracteur qui pourra alors les intégrer. La possibilité donnée à l'utilisateur de fournir des informations à l'extracteur est essentielle pour effectuer des transformations semi-automatiques. Communiquer ces informations dans le code source est aisé.

Une fois que la représentation polyédrique est extraite du programme, on peut profiter de l'analyse offerte par le modèle polyédrique. La section suivante présente quelques outils classiques.

2.5 La plateforme de compilation polyédrique

Une plateforme de compilation polyédrique est composée de plusieurs outils. Les plus classiques permettent : de calculer le minimum lexicographique d'un polyèdre, d'analyser les dépendances, d'optimiser et paralléliser automatiquement.

Calculer le minimum ou le maximum lexicographique d'un ensemble de point entiers appartenant à un polyèdre convexe est une opération polyédrique souvent utile. Par exemple, cela permet de calculer les indices minimum et maximum utilisés pour accéder à un tableau ou permet de savoir à quel indice commence et finit une boucle. *PIP* [32] (*Parametric Integer Programming*) est une bibliothèque C permettant d'effectuer ces calculs. PIP gère les paramètres et les rationnels. Si le problème et paramétrique, Le résultat le sera aussi. Dans ce cas, PIP retourne plusieurs solutions en fonction des valeurs possibles des paramètres utilisés.

Un analyseur de dépendances, comme Candl (*Chunky ANalyzer for Dependences in Loops*), peut être utilisé pour détecter si une transformation est légale. Il permet, par exemple, de détecter les boucles parallèles ou si une transformation donne un SCoP *équivalent* à celui original.

Les optimiseurs polyédriques comme Pluto et *LeTSeE* [33] (the LEgal Transformation SpacE Explorator) trouvent automatiquement un " meilleur " ordonnancement pour optimiser le code (par exemple, par l'amélioration de la localité des données et/ou l'extraction du parallélisme). L'algorithme de Pluto est une technique de compilation *à l'état de l'art* se basant sur le modèle polyédrique afin de construire des transformations de boucles complexe avec un excellent compromis entre la localité des données et le niveau de parallélisme. Pluto trouve les hyperplans permettant d'obtenir des tuiles indépendantes grâce à la résolution de problèmes linéaires en nombres entiers. Il utilise un modèle de coût avec des cibles indépendantes [5]. LeTSeE est une plateforme de compilation polyédrique itérative. Il construit un ensemble d'ordonnancements pour le programme et explore cet espace à la recherche de la meilleure optimisation pour optimiser des cibles spécifiques [34].

Les optimiseurs dits " manuels " permettent à l'utilisateur d'exprimer lui même un meilleur ordonnancement. *Clay* [2, 16] (*Chunky Loop Alteration wizardrY*) en est un exemple. Il s'agit d'une plateforme de transformations haut-niveau basée sur le modèle polyédrique. Il autorise n'importe quel programmeur à interagir avec le modèle polyédrique sans aucune représentation polyédrique. L'utilisateur manipule

uniquement des directives haut-niveau permettant, par exemple, de fusionner, tuiler, décaler et/ou paralléliser des boucles. Contrairement aux optimiseurs automatiques comme Pluto, Clay est manuel, il peut être piloté par l'utilisateur afin de construire sa propre optimisation.

Le modèle polyédrique permet d'appliquer simplement des transformations complexes. Grâce à son analyse des dépendances précises, il est possible d'extraire le parallélisme ou d'améliorer la localité automatiquement. Plusieurs outils sont disponibles, ils composent la plateforme de compilation polyédrique nécessaire à la construction d'un compilateur polyédrique.

Lorsque la représentation polyédrique est optimisée, parallélisée et/ou vectorisée, on peut générer le code avec des outils comme CLooG ou isl. CLooG permet de générer un code source correspondant à la représentation polyédrique. Il se concentre sur la génération d'un code performant.

2.6 Conclusion

Ce chapitre vient de détailler le modèle polyédrique tel que nous l'utilisons dans ce document. Nous y avons introduit plusieurs définitions au sujet de la forme de la relation d'ordonnancement. Ces contraintes et conditions sont nécessaires à la construction d'un ensemble de transformations de code basé sur le modèle polyédrique (détaillé dans le chapitre 5).

De nombreux outils composent la plateforme de compilation polyédrique. Dans ce document, nous utilisons ceux permettant de créer une chaîne de compilation source-à-source. Nous pensons que le développeur doit pouvoir rester au centre des optimisations malgré la complexité technique de ces dernières. Avec les bons outils, les approches semi-automatiques peuvent se révéler efficaces dans la recherche conjointe de la productivité du développeur et de la performance du programme.

Ces outils sont donc des briques de base qui, une fois assemblées, permettent de construire un compilateur polyédrique source-à-source complet. Il est également possible de construire d'autres outils qui nécessitent une analyse statique profonde comme, par exemple, la recherche des accès invalides des tableaux. Il est également possible de créer son propre optimiseur pour des besoins particuliers comme, par exemple, la génération des communications nécessaires pour l'utilisation des machines à mémoires partagées. Les optimiseurs Pluto et LeTSeE donnent une optimisation parmi des milliers alors qu'il est difficile de choisir la version la plus adaptée à la majorité des cas d'autant plus que son existence n'est pas avérée. Il est également difficile de modifier le résultat pour l'adapter à son propre cas. Le chapitre suivant montre la fragilité des optimisations obtenues avec les optimiseurs polyédriques automatiques et, les chapitres ultérieurs abordent ce problème en explorant deux solutions. La première, automatique, évalue des dizaines de versions pour sélectionner et combiner les plus pertinentes afin de profiter de la meilleure optimisation possible. La seconde, semi-automatique, montre comment interagir avec la plateforme de compilation polyédrique en comprenant le résultat de l'optimiseur et en autorisant sa modification.

Chapitre 3

Discussions sur les limites des optimiseurs polyédriques

Sommaire

Les principaux optimiseurs polyédriques comme Pluto et LeTSeE présentés en section 2.5 ont leurs limites. Premièrement, leur fonctionnement est statique et automatique. Deuxièmement, les optimisations sont soit trop générales, soit trop spécialisées.

Automatique. Les optimiseurs polyédriques Pluto et LeTSeE sont automatiques. Même si l'utilisateur peut les appeler en spécifiant plusieurs options afin de modifier leur comportement, orienter les heuristiques et demander des optimisations particulières, il n'y a pas de communication à double sens entre l'utilisateur et ces optimiseurs. Et comme le retour de ces optimiseurs est un nouveau SCoP, l'utilisateur non expert en modèle polyédrique doit attendre la génération de code afin de pouvoir comprendre la nature de l'optimisation. Malheureusement, le code généré est souvent très verbeux et quasiment incompréhensible comme le montre la figure 2.2. Le modifier demande beaucoup de temps et un niveau technique très élevé. Comme ces optimiseurs sont automatiques, aucune interaction n'est nécessaire avec l'utilisateur et, de par leur fonctionnement, aucune réelle interaction avec l'utilisateur n'est possible. Cela pose problème car les optimisations générées sont trop générales ou trop spécialisées.

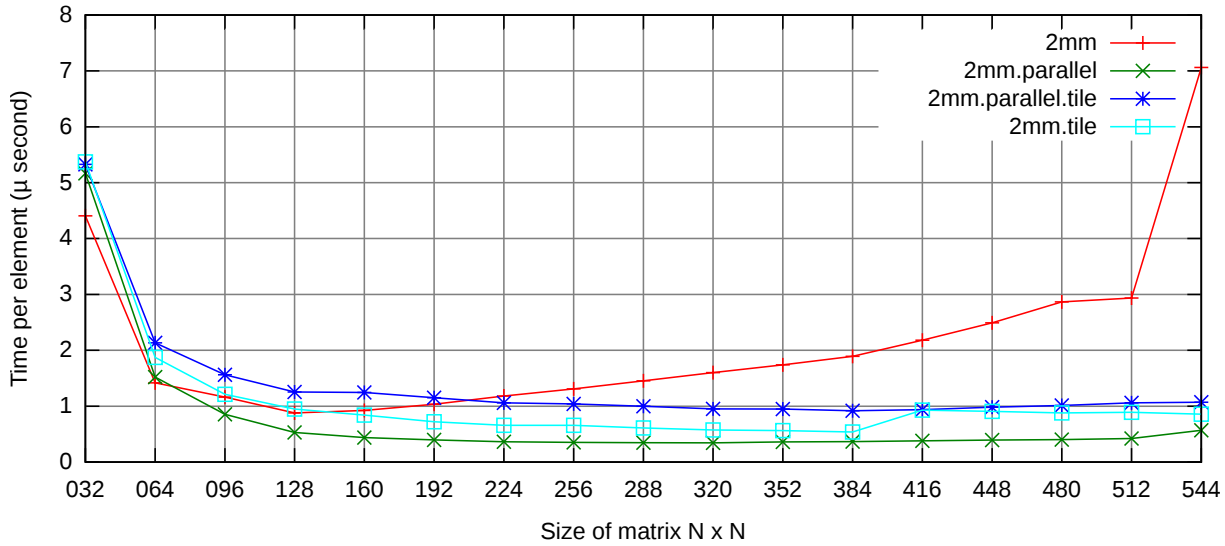
Générale. Les optimiseurs polyédriques qui utilisent l'algorithme de Pluto permettent principalement d'extraire le parallélisme et d'améliorer la localité des données. Pour cela, cet algorithme résout un problème d'optimisation linéaire afin de trouver une optimisation qui offre un bon compromis entre le parallélisme et la localité des données. De par le fonctionnement de ce genre d'optimiseurs polyédriques, l'optimisation proposée est générale, car les heuristiques utilisées ciblent principalement les machines les plus courantes. Il se peut que la version proposée soit non optimale sur l'architecture ciblée par l'utilisateur car trop générale.

Spécialisée. À l'inverse, un optimiseur polyédrique itératif comme LeTSeE va spécialiser l'optimisation pour une machine ciblée. Ces optimiseurs parcourent et évaluent des milliers d'ordonnancements. L'optimisation est retenue en fonction du critère à optimiser. Par exemple, LeTSeE évalue les optimisations en exécutant les différentes versions et retient l'optimisation qui s'est exécutée le plus rapidement. Le critère d'optimisation est donc le temps d'exécution. De par ce fonctionnement, les optimisations faites par ces optimiseurs ne sont pas toujours efficaces sur d'autres architectures que celle ciblée à la compilation ou dans certains environnements dynamiques. Ces optimisations sont donc trop spécialisées.

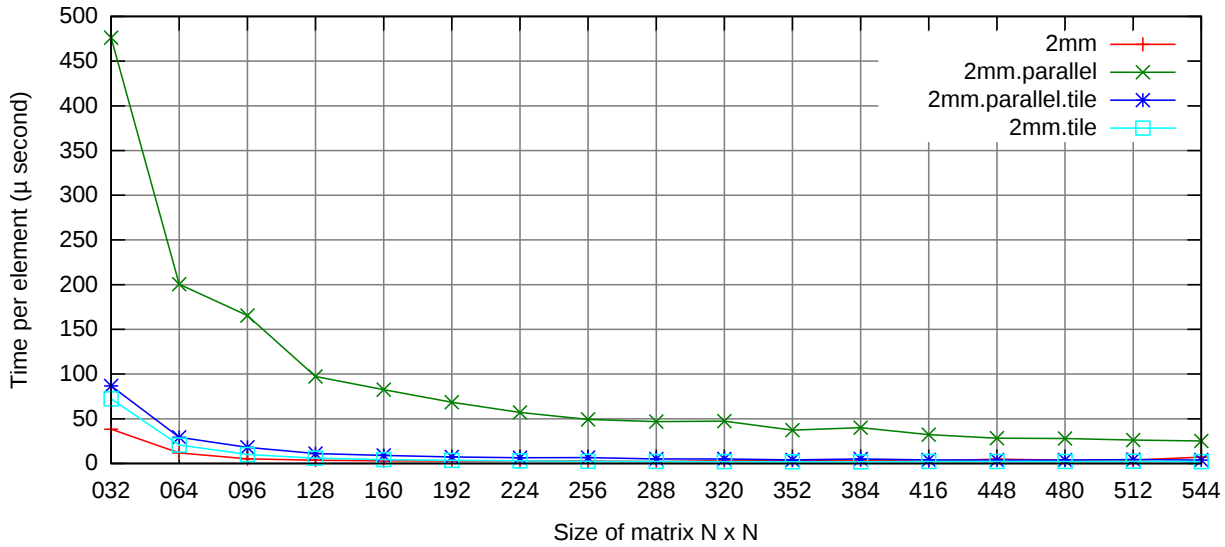
La section suivante présente plusieurs exemples montrant les limites des optimisations. En effet, la performance d'une optimisation n'est pas toujours portable et peut-être grandement liée à différents paramètres comme l'architecture, la taille des données ou même la charge système.

3.1 Exemples

La figure 3.1 montre différentes exécutions du *benchmarks* `2mm` provenant des PolyBench/C 3.2 [1] sur une machine NUMA (*Non-Uniform Memory Access*) composée de deux processeurs Intel Xeon E5656 6 cœurs. Les paramètres de taille de données choisis vont de 32 à 544 par pas de 32. Pour chaque exécution, on mesure le temps de calcul (on utilise la médianne) par élément à traiter en microsecondes. Dans la figure 3.1a, le programme s'exécute seul sur la machine (en plus du système d'exploitation) alors que dans la figure 3.1a, le programme s'exécute à côté de plusieurs autres programmes qui utilisent de nombreuses ressources matérielles car ils font beaucoup de calculs et demandent beaucoup de mémoire.



(a) sans charge système externe notable



(b) avec une charge système externe très élevée

Figure 3.1 – Exécutions avec des charges système différentes : benchmark `2mm` des PolyBench/C 3.2 sur Xeon (deux Intel Xeon CPU E5645 6 cœurs 2.40GHz, Linux 3.2, GCC 4.6.3)

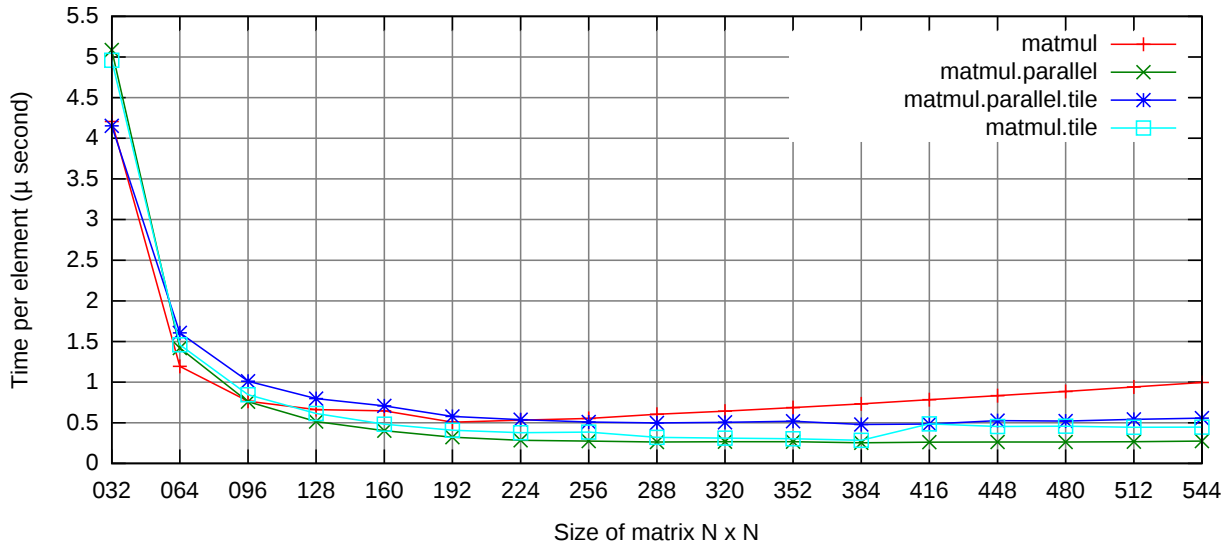
Dans la figure 3.1a, on remarque que la version séquentielle et naïve `2mm` est la plus rapide pour les petites tailles (32 et 64) alors que la version parallèle `2mm.parallel` est la plus rapide pour les autres tailles. Les versions tuilées `2mm.tile` et `2mm.parallel.tile` ont de bonnes performances mais le surcoût dû à la gestion des tuiles est supérieur au gain de cette optimisation.

Lorsque le programme s'exécute sur la même machine mais avec une charge système élevée, les résultats sont très différents. En effet, dans la figure 3.1b, on remarque que le temps par élément à traiter est très élevé pour la version parallèle (notamment pour la taille 32×32, il passe de 5 µs à 475 µs). Ici, la version

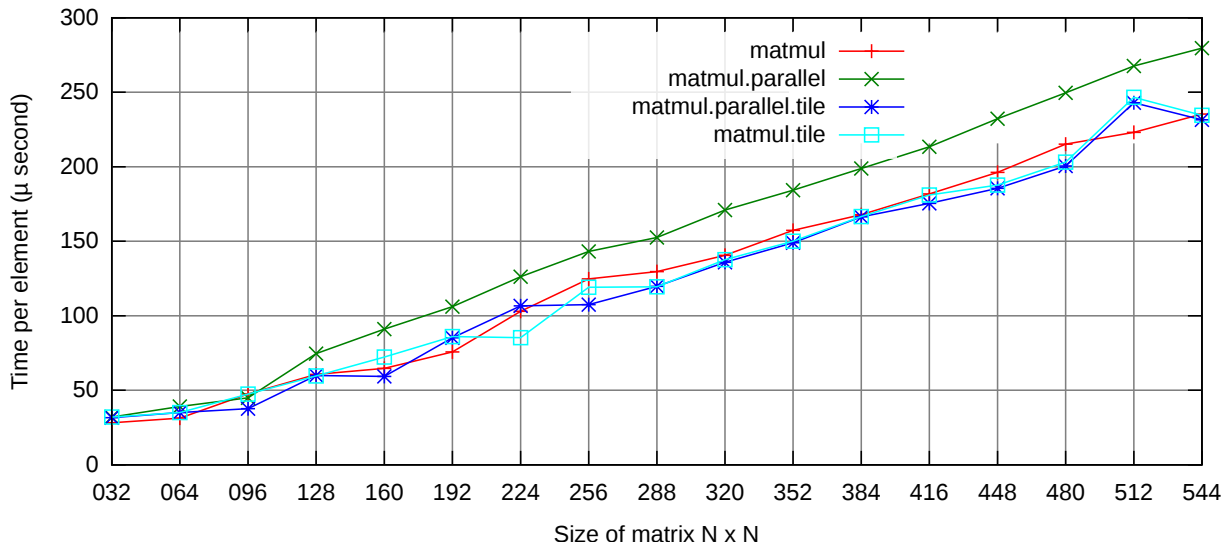
1. http://web.cse.ohio-state.edu/~pouchet/software/polybench/

parallèle est toujours la plus lente alors que les versions tuilées et la version séquentielle sont les plus rapides. La version séquentielle `2mm` est la plus rapide pour les tailles allant jusqu'à 192 inclus. Pour les autres tailles, malgré une minime différence, c'est la version tuilée non parallèle `2mm.tile` qui est la plus performante.

Dans ces différents *contextes d'exécution*, aucune version générée par Pluto 0.10, n'est systématiquement la meilleure. Pour nos contextes, si on ne devait choisir qu'une version, ce serait la version tuilée non parallèle. Dans le cas général, la version à choisir serait probablement la version tuilée *parallèle* même si elle n'est jamais la meilleure pour nos contextes. L'utilisation naïve d'OpenMP par Pluto n'est pas forcément adaptée aux architectures NUMA peu communes pour le grand public. Pour cette raison, la version tuilée *parallèle* est probablement celle à choisir lorsque le programme est prévu pour tourner sur une grande variété d'architectures et de contextes d'exécution.



(a) sur un double Intel Xeon CPU E5645 6 cœurs 2.40GHz, Linux 3.2, GCC 4.6.3



(b) sur LG GW620, ARMv6 1 cœur 600Mhz, Linux 2.6.32, GCC ARM Linux GNU ABI 4.7.2

Figure 3.2 – Exécutions sur des architectures différentes : produit matriciel sur Xeon et ARMv6

La figure 3.2 montre des résultats similaires. Sur le même double Intel Xeon (sans charge système), la version séquentielle est plus rapide que la version parallèle pour les petites tailles (32 et 64). Pour le reste des tailles, c'est la version parallèle qui est la plus rapide. Sur architecture ARMv6, la version parallèle est la plus lente alors que les autres s'alternent pour être la plus performante en fonction des tailles. Sur cette architecture mono-cœur, le parallélisme est inutile. Le tuilage est une bonne optimisation pour améliorer la localité des données mais dans la version dite « ikj » de la multiplication de matrice, la localité des données est déjà bonne.

Les deux résultats des figures 3.1a et 3.2 montrent la fragilité des optimisations face aux changements des paramètres comme la taille, l'architecture et la charge système. Il n'existe pas de version qui serait la meilleure dans l'ensemble des contextes d'exécution. Même si on ne considère qu'un seul paramètre parmi la taille, l'architecture et la charge système, il n'y a pas de meilleure version pour les différentes valeurs du seul paramètre considéré.

Néanmoins, il est possible de choisir une version qui offre de bonnes performances moyennes dans l'ensemble des contextes d'exécution. On dit que cette version est générale dans la mesure où elle est pertinente lorsque les paramètres d'exécution ne sont pas connus à l'avance où si le programme doit s'exécuter dans des contextes variés.

À l'inverse, on peut choisir la version la plus performante lorsque l'environnement d'exécution est fixé à l'avance. Comme cette version offrira les meilleures performances pour ce cas précis, on dit que cette version est spécialisée.

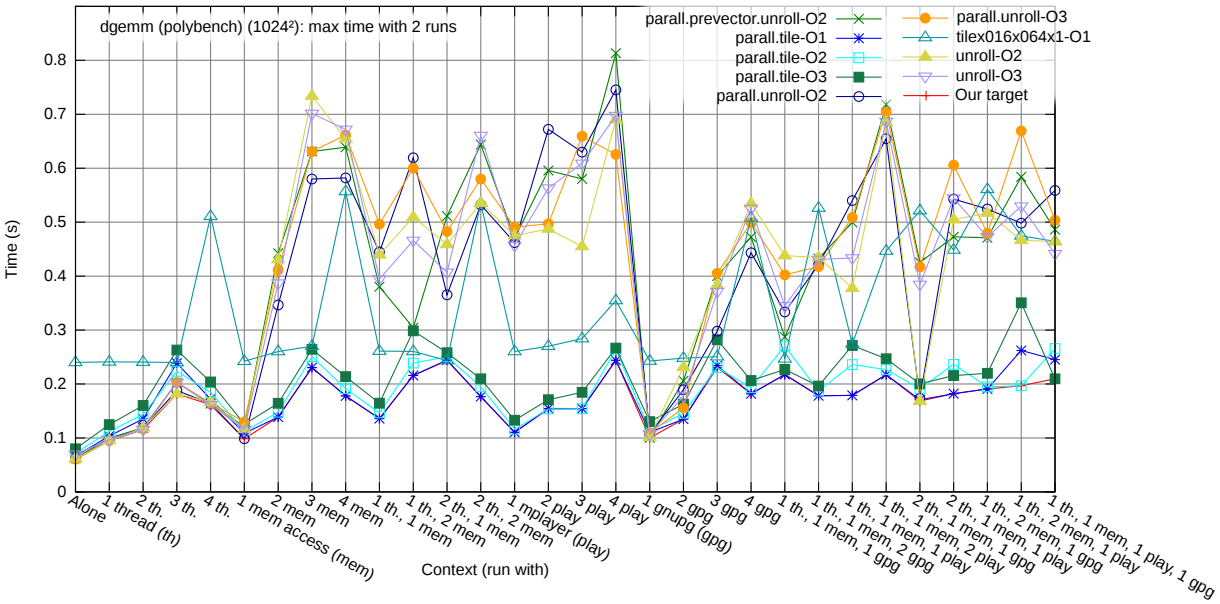


Figure 3.3 – Exécutions avec des charges système différentes : benchmark `dgemm` des PolyBench/C 3.2 sur Intel Quad Core Q6600 2.4 GHz, Linux 3.2, GCC 4.7.2

La figure 3.3 montre les temps d'exécution en secondes du benchmark `dgemm` venant des PolyBench/C 3.2. Différentes versions sont présentes, ces versions ont été générées par Pluto 0.10 et compilées avec GCC 4.7.2. Le nom de la version suit le schéma suivant : *option.de.pluto-niveau.d'optimisation.de.gcc*. Ces version ont été exécutées avec différentes charges système. *Alone* signifie que le programme s'exécute seul. *thread* ou *th* signifie que le programme tourne avec un autre processus qui effectue des calculs (tirage de nombres pseudo-aléatoires). *mem access* ou *mem* signifie que le programme tourne avec un autre processus qui utilise intensivement la mémoire afin de polluer les mémoires cache. *mplayer* ou *play* signifie que le programme tourne avec un autre processus qui affiche une vidéo 1080p. *gnupg* ou *gpg* signifie que le programme tourne avec un autre processus qui signe et chiffre des fichiers. Le nombre devant ces mots correspond au nombre de processus de type qui sont lancés pour cette exécution. Par exemple pour l'exécution de *2 th, 1 mem, 1 play*, il y a 5 processus qui tournent : une version de `dgemm`, deux processus effectuant beaucoup des calculs (*2 th*), un processus effectuant des accès mémoire intensifs (*1 mem*) et une lecture de vidéo 1080p (*1 play*).

Dans ce graphique, on a 9 versions et 30 contextes d'exécution. Ces versions ont été choisies pour leur performance : chaque version est au moins une fois la meilleure dans un contexte d'exécution. On remarque donc qu'en changeant uniquement la charge système, il faudrait combiner sans surcoût 9 versions pour obtenir les performances maximales dans l'ensemble des contextes d'exécutions. Une telle version correspond à la version *Our target*.

3.2 Conclusion

L'ensemble de ces exemples montre la fragilité des optimisations face aux différents contextes d'exécution. Comme ces versions sont obtenues automatiquement avec un optimiseur polyédrique, il est difficile de les modifier pour la plupart des développeurs.

Pour aborder cette problématique, on propose deux approches qui seront détaillés dans les chapitres suivants. La première, automatique, permet de générer un programme multi-versions qui sait s'adapter à son contexte d'exécution permettant de profiter des performances de la meilleure version pour ce contexte d'exécution. La deuxième, semi-automatique, permet au développeur de comprendre et de modifier les optimisations issues d'un optimiseur polyédrique sans connaissance de la représentation polyédrique.

Chapitre 4

Optimisations hybrides

Sommaire

Un des buts des compilateurs est d'optimiser, autant que possible, le code source de l'utilisateur. Malheureusement, les passes d'optimisation du compilateur doivent se limiter aux informations disponibles au moment de la compilation. Comme les architectures sont de plus en plus complexes et de plus en plus hétérogènes et dépendent de plus en plus de paramètres, il est de plus en plus difficile de choisir les bonnes optimisations durant la phase de compilation. Une solution naturelle est de se tourner vers des optimisations dynamiques où plus de valeurs des paramètres sont connues, même si ces dernières présentent des contraintes techniques. Contrairement à l'optimisation dite *statique* où le temps de compilation n'est pas problématique et où de nombreuses informations sur le code source sont disponibles, l'optimisation *dynamique* impose à l'optimiseur une réponse dans un délai très court et n'a pas toujours accès à l'ensemble du code source. Alors que la contrainte de temps empêche l'utilisation des techniques d'optimisation habituelles, la contrainte d'accès peut rendre impossible certaines optimisations.

Ce chapitre présente une nouvelle technique hybride assosiant optimisations statiques et dynamiques. Durant la phase statique, nous créons grâce à des techniques polyédriques de nombreuses versions compatibles entre-elles mais avec des comportements différents en termes de performances. Durant la phase dynamique, on sélectionne le plus rapidement possible la version la plus pertinente pour le cas d'exécution actuel.

4.1 Optimisations statiques & dynamiques

Il y a une vingtaine d'années les compilateurs optimiseurs cherchaient à exploiter les super-calculateurs parallèles. On parlait de « super-compilateurs pour super-ordinateurs » [35]. En effet, une problématique était la conception de compilateurs pour la haute performance sur des machines parallèles à grande échelle. L'architecture et le système, ainsi que la plupart des autres éléments de la configuration, étaient bien connus à l'avance ; et, peu de paramètres étaient révélés uniquement à l'exécution. De plus, sur ces super-ordinateurs parallèles, le lancement des applications était maîtrisé. Généralement, le programme s'exécutait tout seul sur le système ou sur un sous-ensemble du système. Dans cet environnement très statique, la *compilation itérative* et les techniques d'*auto-tuning* ont vu le jour. Elles sont efficaces pour trouver les meilleurs paramètres des optimisations. Elles permettent donc d'adapter les optimisations à différentes architectures et différentes tailles de problèmes qui restent connues à l'avance [36, 37, 34].

Ces techniques reposent sur une exploration empirique des paramètres qui ont le plus d'impact sur les performances comme, par exemple, les tailles de tuiles ou le facteur de déroulage de boucle. Pour trouver les valeurs idéales des paramètres pour une seule configuration, il faut compiler et exécuter de nombreuses versions d'un même programme et comparer les différentes performances obtenues. Le résultat permet de trouver la *meilleure version*. On sélectionne donc la version la plus performante mais, en réalité, il s'agit de la meilleure version pour cette plateforme particulière.

De nos jours toutes les machines grand public sont des systèmes multi-cœurs. Cela est aussi vrai pour les systèmes embarqués et les téléphones portables, de plus en plus sophistiqués. Parallèlement, de plus en plus de calculs demandés par l'utilisateur sont exécutés sur des serveurs distants. Lorsque le résultat du calcul est disponible, il est envoyé à la machine de l'utilisateur comme s'il s'agissait d'un terminal. Cette technique, appelée le *cloud computing*, demande une grosse puissance de calcul afin de répondre à l'ensemble des utilisateurs dans les meilleurs délais possibles. Cette puissance de calcul est disponible grâce à des serveurs de calculs distants. Ces serveurs sont dotés de processeurs multi-cœurs et peuvent intégrer des GPGPU. Les différentes machines composant le serveur peuvent être hétérogènes. Ces architectures ainsi que les nouvelles utilisations apportent de nouveaux facteurs dynamiques qui ne peuvent pas être prévus statiquement, même par la compilation itérative ou l'auto-tuning. C'est le cas, par exemple, du découpage et de la répartition des tâches, de la gestion des processus simultanés demandant beaucoup de ressources pour leurs calculs et, d'éventuelles migrations vers des architectures différentes durant la même exécution. Ces nouveaux cas, difficilement appréhendables par les techniques de compilation et d'optimisation tradionnelles, nécessitent de réfléchir à de nouvelles structures d'optimisation plus dynamiques que celles employées par le passé.

La compilation *just-in-time* est une solution courante et pratique lorsque l'environnement d'exécution du programme est hautement dynamique. Cependant, la complexité algorithmique et la vitesse d'exécution de ce type de compilation doivent être très faibles afin d'éviter que le surcoût dû à cette compilation *just-in-time* ne soit plus important que le gain des nouvelles optimisations faites grâce à cette technique. La qualité de ces optimisations reste fragile. À l'inverse, le modèle polyédrique a fait ses preuves pour les optimisations statiques. Malheureusement, les techniques basées sur ce modèle ont souvent une complexité algorithmique exponentielle [38]. Il est donc délicat d'utiliser directement une telle plateforme pour de la compilation dynamique. Cependant, l'utilisation de la plateforme de compilation polyédrique n'est parfois possible qu'à partir de l'exécution, lorsque les informations nécessaires deviennent disponibles [39]. Notre solution propose de mélanger les techniques d'optimisation statiques et dynamiques afin de bénéficier de la puissance des plateformes polyédriques lors de la compilation statique et d'adapter la décision de l'optimisation à prendre pendant l'exécution, c'est-à-dire dynamiquement.

Le bénéfice potentiel d'une telle technique est élevé car les environnements d'exécution sont maintenant très dynamiques. Cela provient de différents facteurs qui influencent directement les performances.

Une première explication vient des architectures matérielles : un même programme peut s'exécuter sur un très large spectre d'architectures. Par exemple, ces architectures ont des mémoires cache différentes et/ou un nombre de cœurs différent. Ces paramètres déterminent principalement le choix des meilleures optimisations à appliquer et des valeurs de leurs différents paramètres. Une décision en amont de ces optimisations et de ces paramètres n'est pas suffisante. Premièrement, les machines virtuelles et le cloud computing autorisent le programme à changer dynamiquement d'architecture durant son exécution. Ensuite, l'application peut aussi dépendre de paramètres dynamiques comme les données et leur taille. Ces paramètres peuvent être liés au matériel et/ou à l'utilisation et aux habitudes de l'utilisateur. Par exemple, l'affichage dépend des caractéristiques physiques de l'écran (taille, rafraîchissement, ...).

Une seconde explication a une origine logicielle : le système d'exploitation et son ordonnanceur introduisent d'autres facteurs dynamiques. Par exemple, un certain nombre de processus peut affecter un programme en polluant son cache et/ou en rendant la tâche compliquée pour l'ordonnanceur.

Au final, la meilleure optimisation est dépendante de ces paramètres dynamiques à la fois matériels et logiciels. En général, ces derniers ne sont connus qu'à l'exécution. Une solution trop en amont, au mieux, ne pourra pas les prendre en compte et, au pire, rendra l'optimisation choisie moins efficace que l'absence d'optimisation.

Les approches statiques ne prennent pas en compte ces différents facteurs et ne peuvent les prendre en compte que difficilement. Contrairement à l'exécution d'antan sur les « super-ordinateurs », de nombreux processus vivent en même temps sur le système. Ces processus peuvent se concurrencer suffisamment lors

de l'accès aux ressources au point qu'ils se gênent mutuellement. Des effets négatifs sur les performances peuvent être alors observés : les plus directs sont la pollution des caches et le surcoût de l'ordonnanceur.

La pollution des caches annule le bénéfice de cette mémoire indispensable aux performances de la majorité des programmes et peut même introduire des pénalités sur certaines architectures.

Lorsque l'ordonnanceur du système gère un nombre élevé de threads, il y a de nombreuses commutations de contextes qui demandent du temps de traitement supplémentaire afin de sauvegarder et restaurer l'état des processus.

Au final, il y a de nombreux facteurs dynamiques et les prendre tous en compte correctement s'avère déjà difficile à l'exécution et quasiment impossible à la compilation statique.

Notre approche est la suivante : on construit statiquement un programme qui a la possibilité de s'adapter dynamiquement. La nouveauté de notre solution est l'utilisation de *versions interchangeables* (*switcheable scheduling*). Il s'agit d'un ensemble de structurations de programme permettant de changer dynamiquement la version du code à exécuter à des points précis de son exécution. Cela est fait sans aucun retour en arrière (*rollback*) : tout calcul exécuté est correct et utile. La construction d'un tel programme se fait en trois étapes. Une première étape sélectionne les versions interchangeables pertinentes, c'est-à-dire, qui ont de bonnes performances pour au moins certains contextes d'exécution (mais pas forcément tous). La deuxième étape combine ces versions en construisant le programme capable de s'adapter en changeant de versions parmi celles sélectionnées. La troisième et dernière étape se passe à l'exécution : le programme choisit la version la plus performante pour l'exécution. Ce choix se fait en comparant les performances des différentes versions. La comparaison de ces performances se fait dans la phase d'*échantillonnage* : le programme exécute les versions sélectionnées sur des vraies données. Même l'échantillonnage reste du calcul utile car il contribue au résultat final : cela permet d'avoir un faible surcoût. Dans notre expérimentation sur une douzaine de contextes d'exécution, le programme construit grâce à cette technique a une capacité d'adaptation dynamique avantageuse par rapport au programme utilisant uniquement les optimisations statiques qui restent, par définition, limitées aux connaissances disponibles statiquement.

Les sections suivantes de ce chapitre présentent notre méthodologie pour la génération des versions interchangeables ainsi que la construction du programme multi-versions utilisant les versions interchangeables. Ensuite sont présentés les résultats expérimentaux et les travaux associés.

4.2 Versions interchangeables

Pour optimiser un programme, le compilateur polyédrique génère une nouvelle relation d'ordonnancement par instruction. Plusieurs ordonnancements sont possibles et apportent des caractéristiques différentes. Dans ce travail, on se concentre sur un ensemble d'ordonnancements particuliers qui ont la propriété d'être *interchangeables*.

Définition. Deux ordonnancements sont interchangeables si et seulement si, il existe des points de rencontre dans les deux versions des codes générés correspondants tel qu'il soit possible de continuer l'exécution depuis n'importe quelle version avec une autre, sans changer le résultat du programme.

Dans la terminologie du modèle polyédrique, cela signifie qu'il existe un couple de dates logiques, une par ordonnancement, telles que les ensembles des instances qui ont été ordonnancées avant cette date sont les mêmes pour les deux versions, quelle que soit la façon dont ces instances ont été ordonnancées à l'intérieur de l'ensemble. On appelle ces dates logiques les *dates d'interchangeabilité*. Pour simplifier leur calcul, mais sans perdre de généricité, on demande que les dates d'interchangeabilité correspondent à des instances existantes. L'ensemble des dates d'interchangeabilité pour un ordonnancement θ vers un ordonnancement θ' est appelé le *domaine d'itération des instances interchangeables vers* θ'. Cet ensemble correspond à nos points de rencontre.

Propriété. Pour une date d'interchangeabilité donnée dans un ordonnancement, il peut exister uniquement une seule date interchangeabilité correspondante dans un autre ordonnancement.

Explication. Chaque instance du programme original a une unique image dans le programme destination. Étant donné un ensemble d'instances déjà ordonnancées avant un *point de rencontre* (*meeting point*) dans une version, le point de rencontre correspondant dans l'autre version, s'il existe, est l'unique instance qui peut être exécutée juste après cet ensemble.

Propriété. Si les dimensions de sortie les plus externes de deux ordonnancements relient les dimensions d'entrée de façon identique, alors toute instance avant un changement de valeur de la dernière dimension de sortie identique appartient au domaine d'itération des instances interchangeables de l'ordonnancement correspondant vers l'autre ordonnancement.

Explication. Les dates logiques sont multidimensionnelles comme le sont les horloges : la première dimension pourrait correspondre aux jours (il s'agit de la dimension la plus significative dans notre exemple), vient ensuite la dimension suivante correspondante aux heures (cette dimension est moins significative), la dimension suivante correspond aux minutes, puis viennent les secondes et ainsi de suite. Pour chaque valeur des dimensions d'ordonnancement de sortie les plus externes correspond un ensemble d'instances ordonnancées. Si l'ordre d'exécution d'ensembles comme ce dernier est le même pour n'importe quelle version, alors au début de chaque ensemble, il est possible de sauter entre les différentes versions, sans s'occuper de l'ordre de l'ordonnancement interne à l'ensemble, c'est-à-dire, sans regarder les dimensions d'ordonnancement moins significatives. Comme les dimensions de sortie des ordonnancements relient les dimensions d'entrée de façon identique, on peut parler de dimensions de sortie " *communes* " même s'il ne s'agit ni du même ordonnancement, ni de la même instruction.

Par exemple, dans la figure 4.1, comme la dimension de sortie $t1$ est égale à i pour les deux relations d'ordonnancement, chaque instance $t2 = N$ du premier ordonnancement et chaque instance $t2 = 0$ du second ordonnancement, font partie du domaine d'itération des instances interchangeables. Il est possible de changer de version à ces points de rencontre, sans changer le résultat du programme original.

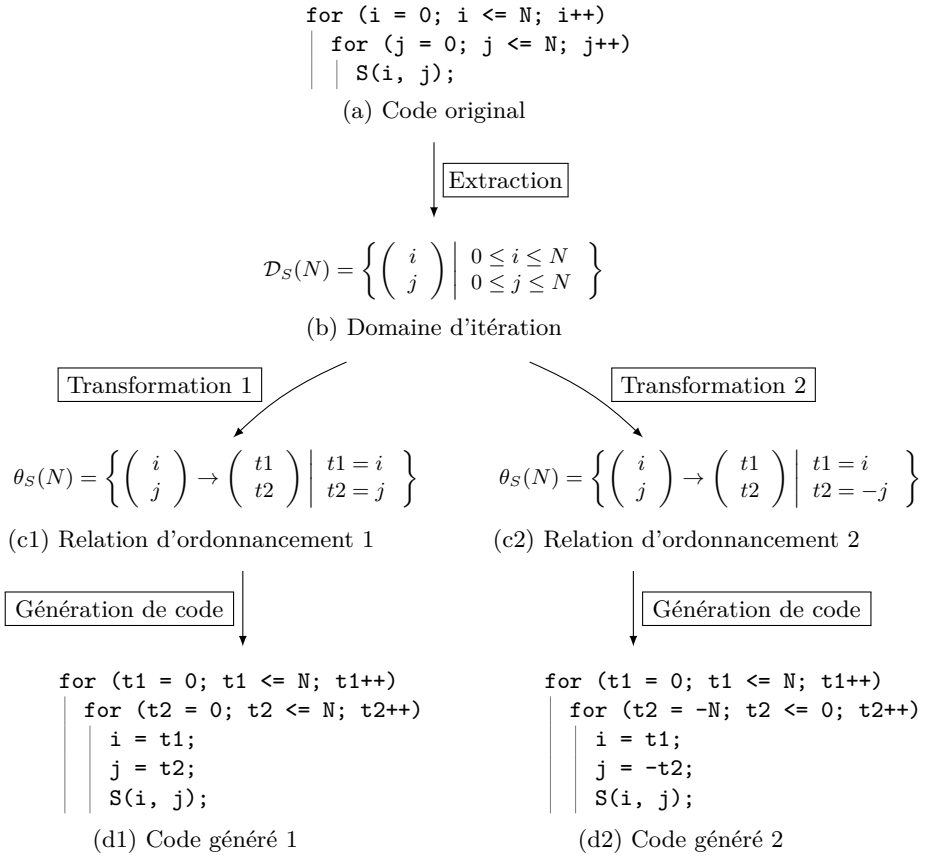


Figure 4.1 – Transformations polyédriques & Génération de code

À partir de ces deux propriétés, nous construisons un nouveau type de générateur de code produisant un code source multi-versions. Dans un premier temps, pour chaque version, on calcule un domaine d'itération des instances interchangeables. Ensuite, on génère le code final en ajoutant les instructions permettant de changer de version pour chaque point entier des domaines des instances interchangeables. On nomme ces instructions les instructions de saut. (Elles autorisent à sauter d'une version à une autre.) Le changement de version se fait à l'exécution avec un faible surcoût afin de ne pas annuler le bénéfice de cette technique.

4.2.1 Le domaine d'itération des instances interchangeables

À partir de la deuxième propriété, on construit le domaine d'itération des instances interchangeables. Ce domaine correspond à l'ensemble des vecteurs de sortie tel que :

1. Les dimensions de sortie communes les plus externes sont exprimées de la même façon pour tous les ordonnancements. Cela assure que toutes les versions exécutent les mêmes sous-ensembles d'instances et ce, dans le même ordre sans s'occuper de l'ordre d'ordonnancement des instances à l'intérieur de ces sous-ensembles. Le cas le plus important que nous gérons est le *stripmining* (équivalent d'un tuilage sur une seule dimension) et donc, par extension, le tuilage avec une restriction sur les tailles de tuiles : ces dernières doivent être un multiple de la plus petite taille de tuile. Cette restriction permet de simplifier le calcul du domaine d'itération des instances interchangeables. Le lien entre les différentes dimensions des différentes versions est connu statiquement : une itération pour une dimension donnée dans une version correspond à n itérations de la même dimension dans une autre version tuilée. On peut alors créer une simple contrainte affine sur l'existence des points de rendez-vous.

2. Les dimensions de sortie restantes sont affectées au minimum lexicographique des valeurs possibles (pour s'assurer que la date logique de l'instruction interchangeable soit au moins la même que la première instance ordonnancée à l'intérieur du sous-ensemble). En outre, nous ajoutons une autre dimension de sortie affectée à 0 pour s'assurer que l'instruction interchangeable soit exécutée avant la première instance du sous-ensemble.

La génération de code utilise les domaines d'itération des instances interchangeables pour insérer les *instructions interchangeables* dans le code final. L'instruction interchangeable est une instruction ajoutée par le générateur de code qui décide s'il faut changer de version ou non. À chaque point entier présent dans ce domaine correspond une possible exécution de l'instruction interchangeable. Comme l'exécution de l'instruction interchangeable introduit un surcoût, on ne souhaite pas l'exécuter à chaque point de rendez-vous et, les *domaines des instructions interchangeables* doivent donc être réduits à ceux uniquement nécessaires. Cela peut se faire facilement.

Une première solution est de prendre l'intersection du domaine avec les points voulus. Avec cette solution, les instructions interchangeables seront exécutées à intervalle constant tout au long des dimensions d'ordonnancement.

Une deuxième solution, avec le même effet, est d'appliquer un stripmining spécial sur certaines dimensions d'ordonnancement. Dans ce cas, les dimensions d'ordonnancement choisies sont remplacées par trois dimensions dans les domaines des instructions interchangeables et dans les relations d'ordonnancement. La dimension la plus externe itère sur les *strips* (tuile une dimension venant du stripmine). La dimension du milieu est affectée à 0 pour le domaine, et à 1 pour toutes les relations d'ordonnancement. La dimension la plus interne est affectée à 0 dans le domaine et itère sur les points entiers à l'intérieur des *strips* pour les relations d'ordonnancement. Cela n'affecte pas l'ordre des instances, mais cela insère une date d'interchangeabilité avant chaque *strip*.

Alors que la première solution est simple, la seconde nous autorise à changer de version dans le cas où il existe des dimensions parallèles : la dimension qui itère sur les *strips* est séquentielle, mais celle qui itère sur les points à l'intérieur des *strips* peut être parallèle.

4.2.2 Génération du code multi-versions

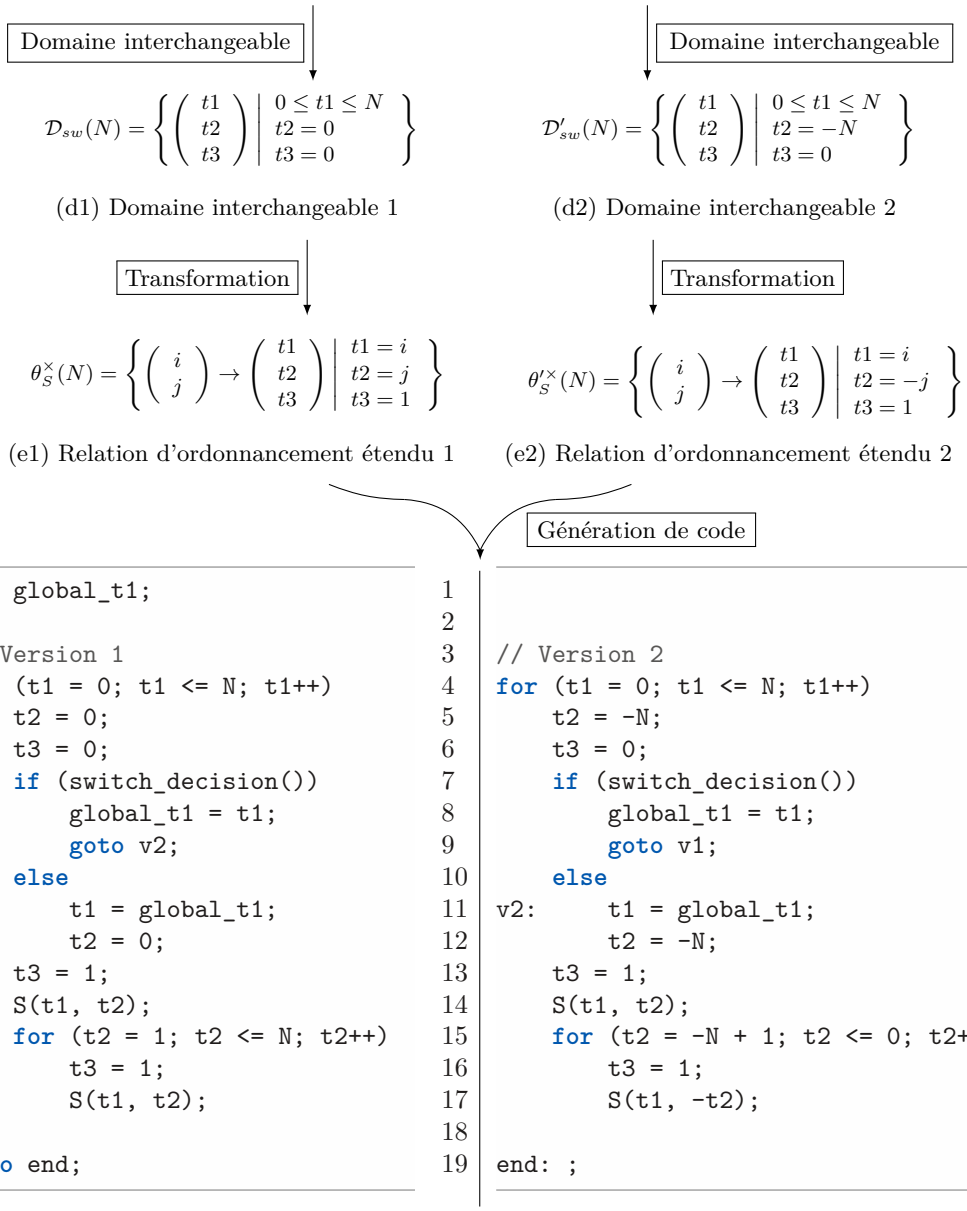
La génération de code doit permettre à plusieurs versions de s'interchanger, c'est-à-dire que le programme multi-versions doit pouvoir changer de version aux points de rendez-vous. La génération d'un tel code se fait en trois étapes.

Dans un premier temps, on étend l'ordonnancement original avec une dimension de sortie la plus interne affectée à 1. En effet, cette dimension de sortie a été affectée à 0 dans le domaine des instances interchangeables. Cela assure que l'instruction interchangeable sera exécutée avant n'importe quelle instance existante si elles sont exécutées à la même date logique. Si la dernière dimension de sortie n'est pas une dimension commune, une autre solution, sans étendre l'ordonnancement, est de soustraire 1 à cette expression dans le domaine des instructions interchangeables. Les figures 4.2(e1) et 4.2(e2) montrent les ordonnancements étendus des figures 4.1(c1) et 4.1(c2).

Ensuite, nous générons le code depuis les domaines et les ordonnancements originaux en utilisant une plateforme polyédrique classique. Dans nos expérimentations, nous avons utilisé CLooG. La seule

différence est que nous générons le code pour chaque version et que nous ajoutons le domaine des instances interchangeables correspondant à chaque génération de code d'une version. Chaque point entier du domaine des instances interchangeables correspond à une exécution de l'instruction interchangeable.

Pour terminer, nous ajoutons un peu de code pour la gestion du changement de version : des variables supplémentaires sont ajoutées afin de communiquer les coordonnées de sortie communes lors du changement de version.



(d1) Domaine interchangeable 1

(d2) Domaine interchangeable 2

(e1) Relation d'ordonnancement étendu 1

(e2) Relation d'ordonnancement étendu 2

```
int global_t1;                                  1
                                                2
// Version 1                                    3
for (t1 = 0; t1 <= N; t1++)                     4
    t2 = 0;                                     5
    t3 = 0;                                     6
    if (switch_decision())                      7
        global_t1 = t1;                         8
        goto v2;                                9
    else                                        10
v1:     t1 = global_t1;                         11
        t2 = 0;                                 12
    t3 = 1;                                     13
    S(t1, t2);                                  14
    for (t2 = 1; t2 <= N; t2++)                 15
        t3 = 1;                                 16
        S(t1, t2);                              17
                                                18
goto end;                                       19
```

```
                                                1
                                                2
// Version 2                                    3
for (t1 = 0; t1 <= N; t1++)                     4
    t2 = -N;                                    5
    t3 = 0;                                     6
    if (switch_decision())                      7
        global_t1 = t1;                         8
        goto v1;                                9
    else                                        10
v2:     t1 = global_t1;                         11
        t2 = -N;                                12
    t3 = 1;                                     13
    S(t1, t2);                                  14
    for (t2 = -N + 1; t2 <= 0; t2++)            15
        t3 = 1;                                 16
        S(t1, -t2);                             17
                                                18
end: ;                                          19
```

(f) Code final incluant les deux versions qui peuvent sauter d'une version à l'autre

Figure 4.2 – Exemple de génération d'un code multi-versions

L'instruction interchangeable elle-même est en deux parties. Premièrement, elle appelle l'évaluation des versions pour décider s'il faut changer de version ou non. Deuxièmement, une fois que la version à exécuter est identifiée, on réceptionne les coordonnées actuelles des dimensions communes . La figure 4.2(f) montre le code final (sur deux colonnes) pour notre exemple commencé en figure 4.1. La première partie de l'instruction interchangeable correspond au bloc du `if` et, la deuxième partie correspond au bloc du `else`.

4.2.3 Exécution

La décision du changement de version effectuée à l'exécution est aussi simple que possible afin de minimiser le surcoût. Elle se base uniquement sur le temps d'exécution et se compose de deux modes : la *surveillance* et *échantillonnage*.

Pendant la phase de surveillance, l'exécution vérifie uniquement si la performance est stable en mesurant le temps dépensé entre deux appels. Comme les instructions interchangeables sont insérées à pas réguliers le long des dimensions de sortie et que le temps d'exécution du SCoP n'est pas affecté par les valeurs des données, cette mesure est assez précise dans notre cas. Si c'est le premier appel à l'exécution ou si la phase de surveillance détecte une variation de performance, on active la phase d'échantillonnage. Des variations de performances peuvent survenir, par exemple, lors des changements du contexte d'exécution ou de la charge du système.

Durant la phase d'échantillonnage, toutes les versions disponibles dans le programme sont exécutées rapidement sur une petite quantité de données afin de détecter celle qui s'exécute le plus rapidement. Ensuite, on saute vers la version la plus performante et l'exécution retourne en mode surveillance. Une propriété très importante de cette stratégie est que chaque calcul contribue au résultat final : il n'y a pas de retour en arrière nécessaire si un mauvais choix d'optimisation est fait. Cette stratégie contribue au faible surcoût de la technique.

Les trois étapes de notre approche (génération des versions interchangeables, sélection des versions pertinantes pour la génération du code multi-versions, exécution) sont entièrement automatiques pour les programmes à base de nids de boucles imbriquées respectant les contraintes du modèle polyédrique. En effet, on sait générer automatiquement différentes versions et les combiner dans un programme multi-versions. Durant l'exécution, ce programme saura s'adapter en tirant profit de la meilleure version disponible. En pratique, l'obtention des performances n'est, en revanche, pas automatique. Un trop grand nombre de versions augmente le surcoût à l'exécution ; un travail de sélection des versions doit être fait avant de les intégrer dans le programme final.

4.3 Sélection des versions pertinentes

Un point clé de notre stratégie d'optimisation est la sélection et l'ordre des versions interchangeables à intégrer dans le code multi-versions. Pour cela, on génère et on évalue intensivement chaque version afin de l'étudier et d'extraire son comportement en fonction des différents contextes d'exécution.

Pour générer les versions, on utilise le compilateur polyédrique PoCC 1.2 [1] afin d'obtenir des ordonnancements efficaces. PoCC permet d'utiliser le moteur d'optimisation de Pluto mais également l'optimiseur itératif LeTSeE. La génération de versions interchangeables est faite en ajoutant les contraintes décrites dans la section précédente : à partir de la version originale, après avoir modifié le domaine d'itération, les autres versions sont créées en appelant Pluto ou LeTSeE avec différentes options, différentes stratégies et/ou différents paramètres comme, par exemple, la taille des tuiles. Ces différentes versions partagent des dimensions de sortie les plus externes. Plusieurs ordonnancements peuvent donner le même exécutable car le compilateur peut annuler certaines modifications, par exemple, le déplacement d'une dimension de sortie. Le compilateur peut donc générer le même code pour deux ordonnancements différents. Ces versions dites doublons sont bien évidemment ignorées.

Une fois que l'ensemble des versions a été généré pour un code donné, ces versions sont évaluées séparément en les exécutant dans différents contextes d'exécution pré-établis. Ces contextes se composent de plusieurs architectures, différentes tailles de données et différents niveaux de charge système. Un contexte d'exécution est une combinaison de ces multiples facteurs. On garde uniquement les versions qui ont été au moins une fois les meilleures pour au moins un contexte. Les expérimentations montrent que le nombre de versions sélectionnées de cette façon reste trop élevé. Un nombre de versions trop élevé ajoute un surcoût à l'exécution durant la phase d'échantillonnage et peut faire grossir l'exécutable suffisamment pour introduire une pénalité sur le cache d'instructions. Comme certaines versions ont des performances similaires dans différents contextes et dans le but de réduire le nombre de versions, on détecte les versions dites *dupliquées* afin de les ignorer elles-aussi. Une version est dupliquée avec une autre si son comportement suit le même que l'autre version, c'est-à-dire, si les performances des deux versions sont très similaires pour chaque contexte (mais pas forcément la même version pour chaque contexte.) (Dans nos

1. http://pocc.sf.net

expérimentations, on accepte une perte de performance de 10%). Au final, avec cette méthode, on sélectionne un nombre restreint de versions (quatre dans nos expérimentations qui est apparu empiriquement comme un bon compromis).

L'ordre dans lequel les versions sélectionnées sont exécutées pendant la phase d'échantillonnage est critique car les petites boucles peuvent être exécutées en entier avant la fin de l'échantillonnage. Pour cette raison, les versions les plus performantes dans la majorité des contextes qui utilisent des petites tailles sont utilisées en premier lors de l'échantillonnage.

4.4 Résultats expérimentaux

Cette approche a été évaluée en utilisant les versions interchangeables sur une sélection de contextes d'exécution réalistes. Les résultats expérimentaux nous montrent la capacité de notre technique à générer des programmes qui peuvent s'adapter eux-mêmes à leur environnement. Dans l'ensemble, la moyenne de l'accélération par rapport à une version fixée qui correspond à la version optimisée et parallélisée par le compilateur Pluto est de **1,67** dans notre cas. La moyenne géométrique est de **1,53**.

Notre protocole expérimental est en trois dimensions. Premièrement, on choisit des architectures cibles représentatives. Dans notre cas, elles sont composées d'un *SoC* (*System on Chip*) ARM et différentes variantes d'Intel x86. La carte ARM est une Olimex A20 ARM Cortex-A7 dual-core ; et les Intel x86 sont un Intel Core2 Quad CPU Q9550 2.83 GHz, un Intel Core2 Quad CPU Q6600 2.40 GHz et un Intel Core2 Quad CPU Q8200 2.33 GHz. Cette sélection d'architectures grand public couvre un ensemble représentatif de nombre de cœurs et tailles de caches différents. Deuxièmement, les tailles des données choisies sont petites et moyennes. On utilise les tailles définies dans les *benchmarks* utilisés (détaillées plus loin). Troisièmement et dernièrement, pour chaque architecture et pour chaque taille, on examine les performances des exécutions avec cinq charges systèmes : le programme s'exécute : seul, avec peu (un processus) ou beaucoup (un processus par cœur) de charge système générée par des calculs intensifs et, avec peu ou beaucoup de charge système générée par des accès à la mémoire (dans le but de polluer la mémoire cache). La combinaison de ces trois dimensions nous donne 40 contextes d'exécution pré-établis permettant d'obtenir le comportement des différentes versions candidates.

On considère 12 benchmarks provenant des PolyBench/C 3.2 [2]. Ces noyaux de calcul sont typiques des calculs intensifs. Notre sélection se concentre sur les noyaux comportant une boucle principale (l'ensemble des instructions est contenu dans cette boucle) car ce sont les programmes ciblés par notre technique. Dans le tableau de la figure 4.3, pour chaque noyau de calcul, il y a : une courte description (colonne « description »), le nombre de versions différentes qui ont été générées avec PoCC 1.2 (sans les versions doublons) (colonne « Nb versions »), le nombre de meilleures versions dans les 40 contextes testés (colonne « Nb meilleures ») et le nombre de versions finalement retenues pour le programme multi-versions (colonne « Nb final ») après avoir enlevé les meilleures versions qui ont le même comportement que d'autres si nous acceptons une baisse des performances de 10% (sans les versions dupliquées). Ces deux dernières métriques nous montrent que la meilleure version est, comme annoncé, dépendante du contexte d'exécution mais, aussi, qu'un nombre limité de versions est suffisant dans la majorité des cas. Cela permet d'avoir un effet limité sur la taille du code généré et le surcoût de l'échantillonnage.

Le tableau de la figure 4.4 donne la performance moyenne pour tous les contextes d'exécution pour chaque noyau de calcul. Il y a une ligne par benchmark et une colonne par version. Une valeur correspond à l'*accélération* de cette version de ce noyau de calcul par rapport à la version de référence : la version nommée « Moyenne ». La dernière colonne fait exception, il ne s'agit pas d'une version mais du rapport entre la taille de l'exécutable de notre solution et la taille de l'exécutable de Pluto 0.10.0. La version « Pire » correspond à la combinaison des pires versions. Il s'agit des versions avec les moins bonnes performances dans les différents contextes d'exécution. Même si on parle de pires versions, il ne faut pas oublier que ces versions ont été sélectionnées car elles étaient les plus performantes dans certains contextes. Leur médiocre performance, entre $1,8$ et $12,5$ fois moins performant que la version « Moyenne », avec une perte moyenne de **3,5** fois (moyenne géométrique à **4,2**). Cela montre bien que le choix de la version à exécuter doit être correctement fait. La version « Moyenne » est la moyenne de toutes les versions dans tous les contextes. Cela correspond à la performance moyenne comme si le choix de la version à

2. http://web.cse.ohio-state.edu/~pouchet/software/polybench/

Noyau de calcul	Description	Nb versions	Nb meilleures	Nb final
2mm	Linear algebra (BLAS3) 2 Matrix Multiplications (D = A × B ; E = C × D)	40	9	2
adi	Stencil (2D) Alternating Direction Implicit solver	67	9	4
choleski	Cholesky Decomposition	16	12	4
durbin	Toeplitz system solver	23	17	4
fdtd-apml	Stencil (3D), FDTD using Anisotropic Perfectly Matched Layer	50	10	2
gemm	Matrix-multiply and addition C = alpha × A × B + beta × C	37	18	4
gramschmidt	Gram-Schmidt decomposition	59	12	2
jacobi-1d	Stencil (1D) 1-D Jacobi stencil computation	24	11	3
jacobi-2d	Stencil (2D) 2-D Jacobi stencil computation	19	7	4
lu	Matrix decomposition	19	8	2
mvt	Matrix Vector Product and Transpose	16	8	2
seidel-2d	Stencil (2D) 2-D Seidel stencil computation	17	7	4
all	*mean*	*32.25*	*10.67*	*3.08*

Figure 4.3 – Informations sur les benchmark et les versions

exécuter était aléatoire. La version « Meilleure » correspond à la combinaison des meilleures versions au même titre que « Pire ». Il s'agit de l'accélération maximum possible avec cette solution : cette version se comporte comme si la meilleure version pour ce contexte était exécutée et cela sans aucun surcoût. Cette version est entre $1,9$ et $19,2$ fois plus performante que la version « Moyenne », avec un gain moyen de **6,75** fois (moyenne géométrique à **4,98**). Le gain potentiel, déjà montré par Pouchet et al. [34], est élevé et correspond à une stratégie proche de la compilation itérative. La version « Pluto » est la version par défaut de Pluto (version 0.10). Elle est, en moyenne, **4,64** fois plus performante que la version « Moyenne » (moyenne géométrique à **2,91**). La version « Interchangeable » est notre solution avec les versions interchangeables. Elle est, en moyenne, **5,98** fois plus performante que la version « Moyenne » (moyenne géométrique à **4,45**). Le surcoût introduit par l'échantillonnage et la surveillance reste acceptable. Par rapport à la version par défaut de Pluto, notre solution a une accélération de **1,68** en moyenne (moyenne géométrique à **1,53**). La colonne « Taille exécutable » montre que la taille de l'exécutable (du noyau de calcul) issue de la version interchangeable est légèrement plus grosse que celle de l'exécutable de Pluto, ce qui est très acceptable.

L'échantillonnage sur des mauvaises versions peut dégrader significativement les performances, comme avec le benchmark « gemm ». Pour le noyau de calcul « jacobi-1d », notre solution est moins performante que celle de Pluto. Cela peut s'expliquer facilement : la version de Pluto est suffisamment performante et le surcoût introduit par l'échantillonnage des versions interchangeables est trop important par rapport au bénéfice obtenu. Pour pallier ces problèmes, notre technique devrait inclure des tests dynamiques [40] afin d'éviter l'utilisation systématique des versions interchangeables dans ces cas.

Benchmark	Pire	Moyenne	Meilleure	Pluto	Interchangeable	Taille exécutable
2mm	0.38	1	3.56	1.48	3.14	1.13
adi	0.13	1	4.46	2.98	4.08	1.07
choleski	0.74	1	1.89	1.35	1.52	1.02
durbin	0.25	1	2.14	1.74	1.90	1.04
fdtd-apml	0.08	1	2.77	2.19	2.61	1.07
gemm	0.31	1	8.42	1.39	5.70	1.04
gramschmidt	0.10	1	18.27	17.34	17.36	0.99
jacobi-1d	0.17	1	19.15	16.30	15.71	1.10
jacobi-2d	0.25	1	8.24	4.08	7.87	1.38
lu	0.24	1	4.42	3.02	4.82	1.04
mvt	0.55	1	2.28	1.54	2.12	1.06
seidel-2d	0.26	1	5.37	2.21	4.97	1.11

Figure 4.4 – Performances potentielle (colonne « Meilleure ») et obtenue (colonne « Interchangeable »)

4.5 Travaux associés

Notre approche est basée sur le modèle polyédrique et le versionnage de boucles [41]. Malgré la puissance du modèle polyédrique, une approche dynamique est nécessaire car les optimiseurs polyédriques comme Pluto et LeTSeE donnent des optimisations sous-spécialisées ou sur-spécialisées. Ces optimisations ne sont pas toujours adaptées aux différents contextes d'exécution. Notre solution s'adapte au contexte d'exécution et permet donc d'éviter ce problème.

Les techniques d'optimisation habituellement statiques des compilateurs ont été utilisées pour aider les systèmes d'exécution à effectuer des optimisations dynamiquement. Par exemple, la plateforme ADAPT les utilise pour la génération et la spécialisation de certaines parties du code dynamiquement [42]. Comme le surcoût à l'exécution est élevé, cette technique fonctionne bien pour les programmes dont le temps d'exécution est suffisamment grand. Dans notre solution, nous préférons garder les techniques de compilation statiques, autant que possible, statiques afin de minimiser les coûts à l'exécution. D'autres travaux se rapprochent du nôtre. Qilin s'adapte au matériel hétérogène en utilisant dynamiquement le CPU ou le GPU pour exécuter les calculs [43]. Contrairement à notre méthode, il ne s'intéresse pas à la charge du système lors de l'exécution. Emani et al. proposent une technique permettant au programme de s'adapter à la charge du système externe tout au long de son exécution [44]. Le programme est parallélisé avec OpenMP et le nombre de cœurs utilisés par OpenMP est ajusté dynamiquement en fonction de la charge système externe. Dans cette technique, seul le nombre de cœurs varie alors que notre solution permet de restructurer le code pendant l'exécution.

Les techniques d'optimisation dynamiques agressives incluent la parallélisation spéculative comme le test `LRPD` [45, 46]. Cette technique génère une version optimisée optimiste, une version parallélisée par exemple. À l'exécution, c'est la version optimisée qui est lancée mais, dans le cas où cette version s'avère fausse, suite à une violation de dépendance par exemple, un retour en arrière est effectué et on exécute la version exacte. Notre technique cible des programmes qui peuvent être entièrement analysés à la compilation grâce au modèle polyédrique et, même si un mauvais choix est fait, aucun retour en arrière n'est nécessaire car chaque calcul est utile par construction. Cela permet de limiter considérablement la pénalité introduite par un mauvais choix.

Des optimisations dynamiques utilisant les techniques de compilations polyédriques existent mais sont relativement récentes. `EvolveTile` est une plateforme permettant d'itérer sur différentes tailles de tuiles et de sélectionner dynamiquement la taille de tuile la plus performante [47]. Notre approche supporte aussi ce genre d'optimisations mais avec plus de restrictions sur les tailles et les formes des tuiles dues aux contraintes inhérentes aux versions interchangeables. Cependant notre technique a un panel d'optimisations plus large. Pradelle et al. ciblent des mêmes programmes que notre technique et gèrent également différentes versions [40]. Leur approche évalue les différentes versions et construit des tests prédictibles en fonction de facteurs dynamiques (architecture, taille des données, ...) afin de choisir la meilleure version à utiliser juste avant son exécution. Notre approche a un grain plus fin car on se concentre sur l'interchangeabilité des versions en cours d'exécution. Dollinger et Loechner répondent à des problématiques

similaires pour la génération de code sur GPGPU [48, 49]. `VMAD` est une infrastructure pour effectuer du *profilage* dynamique avec la possibilité de découvrir des propriétés statiques qui n'étaient pas visibles statiquement durant la compilation, c'est-à-dire, sans exécution [39]. L'infrastructure `VMAD` supporte la sélection dynamique de versions. Certaines formes de versions interchangeables sont donc possibles. Srinivas et al. propose un *tuilage réactif* [50] permettant d'adapter la taille des tuiles dynamiquement en fonction des allocations mémoire faites par le système d'exploitation dans le but de ne pas être pénalisé par une mauvaise utilisation de la mémoire cache disponible.

4.6 Conclusion

L'utilisation de versions interchangeables permet de tirer parti des performances de la meilleure optimisation y compris dans un environnement d'exécution de plus en plus dynamique. Notre approche cible les programmes à base de nids de boucles imbriquées avec contrôle statique (affine plus exactement) répondant aux contraintes du modèle polyédrique. Comme les approches basées sur la compilation à la volée, notre solution se base aussi sur des techniques dynamiques dont le surcoût est faible. Cependant, contrairement à la compilation à la volée, on s'autorise à prendre plusieures heures voire même plusieurs jours afin d'explorer de nombreuses versions durant la phase de compilation statique. Comme certaines informations nécessaires à une meilleure optimisation ne sont pas disponibles statiquement, la sélection de la version à exécuter se fait durant l'exécution elle-même. De ce fait, notre solution diffère aussi des approches de compilation statiques qui génèrent des optimisations trop génériques ou trop spécialisées donnant des versions dont les performances ne sont pas assez élevées dans certains contextes et non portables.

En mélangeant les approches statiques et dynamiques, notre solution construit une technique de compilation polyédrique à état de l'art. Durant la phase de compilation statique, l'étude empirique permet de sélectionner les optimisations pertinentes. Durant l'exécution, notre programme permet de sauter à la meilleure optimisation selon le contexte d'exécution en cours avec un faible surcoût.

Notre solution introduit une classe spécifique de programmes qu'on appelle les versions interchangeables et une méthode de génération de code permettant de construire le programme qui sait profiter des avantages de ces optimisations. Nos expériences mettent en évidence le potentiel de cette technique ainsi que son efficacité à générer des programmes s'adaptant à un nombre varié d'environnements dynamiquement.

Même si cette approche permet au programme de s'adapter à son environnement d'exécution, elle a, sur le fond, les mêmes limitations que les autres approches tout-automatiques : les choix d'optimisations sont faits en fonction et au fur et à mesure des informations disponibles. Dans certains cas, l'utilisateur connait ces informations et pourrait alors les communiquer au compilateur. Grâce à ces informations, le compilateur peut, par exemple, optimiser plus vite et plus efficacement en réduisant son espace de recherche des versions et en limitant le nombre de versions dans l'exécutable final. Actuellement les intéractions avec le compilateur sont compliquées et limitées. Cela est d'autant plus vrai lorsqu'il s'agit de s'adresser à l'optimiseur polyédrique. Les chapitres suivants permettent de profiter manuellement de la puissance des plateformes polyédriques sans connaissance nécessaire du modèle polyédrique.

Chapitre 5

Transformations semi-automatiques

Sommaire

Il peut exister un gouffre entre les performances obtenues par un programme naïf optimisé par un compilateur automatique et celles provenant d'un programme écrit par un humain expert. En contrepartie, un gouffre est également présent si l'on compare leur temps de développement et leur temps de débogage. Actuellement les compilateurs offrent un très bon compromis entre la productivité et les performances du programme. Ces performances ne sont pas toujours celles attendues et certaines parties sensibles d'un programme peuvent avoir besoin d'une optimisation plus efficace mais qui demande un coût plus élevé.

Les échanges avec l'optimiseur des compilateurs sont limités pour ne pas dire inexistants. Le compilateur peut informer l'utilisateur et, l'utilisateur peut informer le compilateur. Malheureusement, dans la plupart des cas, la communication est à sens unique dans la mesure où il n'y a pas d'échange entre le développeur et le compilateur. Le compilateur peut émettre des messages pour dire où son optimisation a échoué. Par exemple, on peut demander à GCC d'afficher les instructions vectorisées avec les options `-ftree-vectorizer-verbose=6 -ftree-vectorize -fopt-info-vec-missed` mais les explications ne sont pas toujours suffisantes pour que la plupart des développeurs comprennent l'origine du problème et/ou sachent le résoudre. Dans l'autre sens, le développeur a la possibilité d'ajouter des informations destinées au compilateur, le plus souvent sous forme d'annotations spécifiques ou de mots clés. Ces annotations peuvent être dépendantes du compilateur ou offertes par le langage. Ces informations sont utiles au compilateur qui peut alors appliquer ses optimisations. Par exemple, les annotations

d'OpenMP permettent de paralléliser une boucle relativement facilement et fournissent plusieurs options afin de contrôler comment la parallélisation doit être faite. Le mot clé `inline` disponible en C99 et C++ permet d'autoriser le compilateur à remplacer l'appel de la fonction *inlinée* par son code directement comme si le programmeur n'avait pas écrit de fonction. Il s'agit d'une proposition faite par le développeur, c'est le compilateur qui choisira d'*inliner* ou pas la fonction. Avec GCC l'annotation `__attribute__((optimize("unroll-loops")))`, combinée avec l'option `-funroll-loops`, permet en principe de dérouler une boucle. Cependant, il reste très difficile de savoir si le compilateur a effectivement déroulé la boucle et, dans le cas où elle n'est pas déroulée, pourquoi il n'a pas fait l'optimisation demandée (forme de boucle invalide, optimisation jugée inutile ou dégradant les performances, autre optimisation plus intéressante, ...). Une autre possibilité est de modifier le code pour essayer de le faire correspondre aux cas gérés par les passes d'optimisations afin d'obtenir la transformation voulue. Cette solution n'est pas toujours une option possible car elle dépend fortement du compilateur et de sa version. De plus, c'est une tâche difficile qui se rapproche d'une optimisation manuelle et, comme toute optimisation faite par le développeur, elle est sensible au compilateur qui pourrait l'altérer ou tout simplement l'éliminer. Le choix des options des compilateurs et leur combinaison influence les performances de l'exécutable produit comme le montre Fursin [51] qui propose une infrastructure collective permettant de suggérer les arguments à passer au compilateur en fonction du code source et de la machine ciblée. Ces suggestions sont statistiques et empiriques, elles se basent sur les données collectées précédemment.

L'intéraction avec le compilateur est déjà limitée dans le cas normal et cela est renforcé avec l'optimiseur polyédrique. Dans le cas général, le développeur et les différentes passes d'optimisation du compilateur peuvent se contredire et, même si le compilateur est au service de l'utilisateur, il a le dernier mot. Dans le cas qui nous intéresse, l'optimiseur polyédrique, l'intéraction est particulièrement difficile car si un développeur veut l'utiliser directement, il doit maîtriser le modèle polyédrique et être en mesure d'injecter ses ordonnancements. Des travaux permettent déjà au programmeur de profiter des transformations agressives offertes par la plateforme de compilation polyédrique. UTF [52, 53] est le premier d'entre-eux. Les transformations supportées permettent notamment de réorganiser les instructions, de fusionner et distribuer les boucles, d'incliner et de reverser les boucles mais aussi de tuiler et découper les boucles. Ces transformations disponibles sous la forme de directives étaient traduites en ordonnancements. URUK [28] permet d'écrire des séquences de transformations de boucles classiques sous certaines contraintes (ordonnancement *unimodulaire*). URUK utilise une ancienne représentation polyédrique : les *fonctions*. Elle est moins générique que celle présentée dans ce travail : les unions de relations basiques. CHiLL [27, 54] a beaucoup de choses en commun avec URUK. Il utilise une représentation polyédrique plus proche de la nôtre en ajoutant des dimensions équivalentes à nos dimensions β. CHiLL maintient un graphe des dépendances à chaque transformation pour les vérifier mais aussi pour générer le code. Ces plateformes de compilation polyédriques offrent à l'utilisateur un moyen de les utiliser afin d'effectuer des transformations de boucles classiques mais ont deux limitations principales. Premièrement, ils n'offrent pas toute la puissance du modèle polyédrique car les transformations ne permettent pas de construire *n'importe quel ordonnancement polyédrique* (respectant la contrainte définie en 2.2.3). Deuxièmement, il s'agit encore d'une communication à sens unique car seul l'utilisateur ordonne des transformations à la plateforme de compilation polyédrique. Ce dernier n'a pas la possibilité d'améliorer une transformation calculée automatiquement ou de lui proposer une séquence de transformations alternative.

Pour répondre à une partie du problème, il faudrait que l'utilisateur ait la possibilité d'avoir le dernier mot sur l'optimiseur disponible dans le compilateur. Pour cela, nous proposons de transformer le résultat de l'optimiseur polyédrique en une séquence de transformations compréhensible par le développeur. Cette séquence de transformations peut être modifiée par le développeur et être donnée au générateur de code. De cette façon, l'optimisation retenue et effective sera, soit celle de l'optimiseur polyédrique éventuellement vérifiée et validée par le développeur, soit celle de l'optimiseur polyédrique modifiée par le développeur pour être conforme à l'optimisation voulue, soit directement celle du développeur qui peut, ou pas, s'inspirer de celle de l'optimiseur polyédrique.

Dans un premier temps, on propose de revisiter mais aussi d'étendre les transformations classiques de boucles basées sur une plateforme de compilation polyédrique. Notre plateforme se nomme *Clay* [16, 2] (Chunky Loop Alteration wizardrY). Il est écrit en C et utilise OpenScop Library.

5.1 Transformations de code

Cette section présente les transformations de boucles exprimées dans le modèle polyédrique. La plupart de ces transformations sont dites revisitées car elles correspondent aux transformations classiques [35]. Au contraire, certaines sont nouvelles et permettent d'exprimer, avec les directives de transformations classiques, l'ensemble complet des *ordonnancements polyédrique* possibles.

Chaque transformation est détaillée et un exemple est donné. Les descriptions formelles sont écrites avec des notations mathématiques. Comme toutes nos transformations définissent un nouvel ordonnancement, elles modifient uniquement la relation d'ordonnancement.

Voici les différentes notations mathématiques utilisées :

$\vec{v}$ Un vecteur nommé v
Exemple : $(1, 2, 3)$

$\mathcal{E}$ Un ensemble d'élements nommé $\mathcal{E}$
Exemple : $\{2, 0, 1\}$
Exemple : $\{(1, 2), (0, 2), (3)\}$

M Une matrice nommée M
Exemple : $\begin{pmatrix} 11 & 12 \\ 21 & 22 \end{pmatrix}$

$\vec{v}_i$ i^{e} élément de $\vec{v}$
Exemple : $(1, 2, 3)_2 = 2$

$\vec{v}_{n_1..n_2}$ Un vecteur avec les éléments n_1 à n_2 de $\vec{v}$
Exemple : $(1, 2, 3, 4, 5)_{2..4} = (2, 3, 4)$

$\rho(\vec{v})$ Fonction qui retourne le plus long préfixe du vecteur $\vec{v}$ donné soit $\vec{v}_{1..\dim \vec{v}-1}$
Exemple : $\rho((1, 2, 3)) = (1, 2)$

θ Relation d'ordonnancement $\theta(\vec{p}) = \bigcup_i \mathcal{T}_i$

$\mathcal{T}$ Relation basique d'ordonnancement $\mathcal{T}(\vec{p}) = \left\{ \vec{\iota}_\mathcal{T} \to \vec{\sigma}_\mathcal{T} \,\middle|\, K_\mathcal{T} \geq \vec{0} \right\}$

$\vec{p}$ Vecteur des paramètres Exemple : (N, M)

$\vec{\iota}_\mathcal{T} \to \vec{\sigma}_\mathcal{T}$ Le vecteur des dimensions d'entrée de $\mathcal{T}$ ($\vec{\iota}_\mathcal{T}$) est lié au vecteur des dimensions de sortie de $\mathcal{T}$ ($\vec{\sigma}_\mathcal{T}$)

$K_\mathcal{T}$ Matrices des contraintes de $\mathcal{T}$
Exemple :

$$\begin{array}{c|ccccc|cc|cc|c|c|l} & \beta_1 & \alpha_1 & \beta_2 & \alpha_2 & \beta_3 & i & j & N & M & 1 & & \\ \hline & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = 0 & \vec{\beta}_{\mathcal{T},1} = 0 \\ & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & = 0 & \vec{\alpha}_{\mathcal{T},1} = i \\ & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & = 0 & \vec{\beta}_{\mathcal{T},2} = 0 \\ & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & = 0 & \vec{\alpha}_{\mathcal{T},2} = j \\ & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & = 0 & \vec{\beta}_{\mathcal{T},3} = 0 \end{array}$$

$\vec{\iota}_\mathcal{T}$ Vecteur des dimensions d'entrée de $\mathcal{T}$ Exemple : $(-1,-1)$ extrait de $K_\mathcal{T}$

$\vec{\sigma}_\mathcal{T}$ Vecteur des dimensions de sortie de $\mathcal{T}$ Exemple : $(-1,-1,-1,-1,-1)$ extrait de $K_\mathcal{T}$

$\vec{\alpha}_\mathcal{T}$ Vecteur des dimensions α of $\mathcal{T}$ Exemple : $(-1,-1)$ extrait de $K_\mathcal{T}$

$\vec{\beta}_\mathcal{T}$ Vecteur des dimensions β of $\mathcal{T}$ Exemple : $(0,0,0)$ extrait de $K_\mathcal{T}$

$\vec{\rho}$ Préfixe d'un $\vec{\beta}_\mathcal{T}$ ou d'un autre $\vec{\rho}$ (si le vecteur est vide, cela correspond à la racine)
On parle de β-préfixe. (Ne pas confondre avec la fonction $\rho(\vec{v})$.)

$\mathcal{T}_*$ Ensemble de toutes les relations basiques de toutes les relations d'ordonnancement

$\mathcal{T}_{\vec{\rho},*}$ Sous-ensemble de $\mathcal{T}_*$ limité aux relations basiques qui ont le même $\vec{\rho}$

$\mathcal{T}_{\vec{\rho},i}$ $i^{\text{ième}}$ relation de $\mathcal{T}_{\vec{\rho},*}$ ($\mathcal{T}_{\vec{\rho},*}$ sont triés selon les valeurs des $\vec{\beta}_\mathcal{T}$)

$\mathcal{T}_{\vec{\rho},>}$ Sous-ensemble de $\mathcal{T}_*$ limité aux relations basiques
$\left\{\mathcal{T} : \vec{\beta}_{\mathcal{T},1..\dim\,\vec{\rho}-1} = \rho(\vec{\rho}) \text{ et } \vec{\beta}_{\mathcal{T},\dim\,\vec{\rho}} > \vec{\rho}_{\dim\,\vec{\rho}}\right\}$
Cela correspond aux $\mathcal{T}$ des instructions qui sont dans les boucles après celle désignée par $\vec{\rho}$ (mais au même niveau d'imbrication)

$\mathcal{T}_{\vec{\rho},\geq}$ Sous-ensemble de $\mathcal{T}_*$ limité aux relations basiques
$\left\{\mathcal{T} : \vec{\beta}_{\mathcal{T},1..\dim\,\vec{\rho}-1} = \rho(\vec{\rho}) \text{ et } \vec{\beta}_{\mathcal{T},\dim\,\vec{\rho}} \geq \vec{\rho}_{\dim\,\vec{\rho}}\right\}$
Cela correspond aux $\mathcal{T}$ des instructions qui sont dans la boucle désignée par $\vec{\rho}$ et dans les boucles suivantes (mais au même niveau d'imbrication)

$\mathcal{T}_{\vec{\rho},next}$ Sous-ensemble de $\mathcal{T}_*$ limité aux relations basiques
$\left\{\mathcal{T} : \vec{\beta}_{\mathcal{T},1..\dim\,\vec{\rho}-1} = \rho(\vec{\rho}) \text{ et } \vec{\beta}_{\mathcal{T},\dim\,\vec{\rho}} = \vec{\rho}_{\dim\,\vec{\rho}} + 1\right\}$
Cela correspond aux $\mathcal{T}$ des instructions qui sont dans la boucle juste après celle désignée par $\vec{\rho}$ au même niveau d'imbrication

Exemple :

```
for (...)
{
  for (...)
  {
    S1, β⃗_T = (0,0,0), T_(0,0),1   | ∈ T_(0,0),*
    S2, β⃗_T = (0,0,1), T_(0,0),2   |
  }
  for (...)
  {
    S3, β⃗_T = (0,1,0)              | ∈ T_(0,0),next | ∈ T_(0,0),≥
    for (...)                      |                |
    {                              |                |
      S4, β⃗_T = (0,1,1,0)          |                |
    }                              |                |
    S5, β⃗_T = (0,1,2)              |                | ∈ T_(0,0),>
  }                                |                |
  for (...)                        |                |
  {                                |                |
    S6, β⃗_T = (0,2,0)              |                |
    S7, β⃗_T = (0,2,1)              |                |
  }
}
for (...)
{
  S8, β⃗_T = (1,0)
}
```

Dans le reste de la section, nous détaillons chaque transformation. Chaque descriptif commence par la syntaxe de la transformation. La syntaxe est composée du nom de la transformation et de ses paramètres. Ensuite, une description formelle et informelle est donnée. La partie dite formelle est écrite en langage mathématique et la partie informelle traduit de manière intuitive la notation mathématique. En suivant le même modèle, les conditions sont données. Il s'agit de *préconditions*, c'est-à-dire que les conditions peuvent et doivent être vérifiées avant d'appliquer la transformation. Ensuite, la définition permet d'expliquer ce que fait la transformation et comment elle le fait. Pour finir, nous présentons un exemple. Cet exemple est en trois parties. Pour chaque partie, il y a la représentation avant la transformation et après. La première partie représente l'exemple avec un code source en C. La deuxième partie de l'exemple utilise la représentation du modèle polyédrique et la troisième et dernière partie correspond à la représentation graphique des polyèdres. Chaque transformation travaille sur un SCoP entier qui peut donc être composé de plusieurs instructions.

Transformations syntaxiques

Les trois transformations *syntaxiques* suivantes opèrent uniquement sur les valeurs des β. Ces valeurs nous permettent de réorganiser les instructions et les boucles en les réordonnant, les fusionnant ou les distribuant. Ces transformations sont *reorder*, *fuse* et *distribute*.

5.1.1 Réorganiser les instructions

Syntaxe : $reorder(\vec{\rho}, \vec{v})$
$reorder(\beta$-prefixe, vecteur d'entiers correspondant au nouvel ordre des instructions$)$

Effet :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*},\ \vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} \leftarrow \vec{v}_{\vec{\beta}_{\mathcal{T}},\dim \vec{\rho}+1}$ Pour chaque instruction qui est dans la boucle désignée par $\vec{\rho}$, on réécrit la valeur de chaque $\vec{\beta}$ permettant la réorganisation

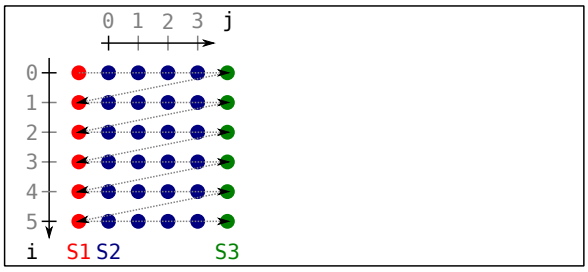
Conditions :
$\dim \vec{v} = \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho},*}} \left(\vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1}\right) + 1$ Le nombre de valeurs dans $\vec{v}$ est le même que le nombre d'instructions et de boucles qui sont directement dans la boucle désignée par $\vec{\rho}$
$\forall i\ \ 1 \leq i \leq \dim \vec{v},\ \ 0 \leq \vec{v}_i \leq \dim \vec{v} - 1$ Toutes les valeurs de $\vec{v}$ sont comprises entre 0 et la taille de $\vec{v}$ (non inclus)
$\forall i, j\ \ i \neq j,\ \ \vec{v}_i \neq \vec{v}_j$ Toutes les valeurs de $\vec{v}$ sont uniques

Définition :
Réorganise les instructions et les boucles directement présentes dans la boucle désignée par $\vec{\rho}$ et selon le $\vec{v}$ donné. Le nouvel emplacement dans la boucle de l'instruction ou de la boucle à l'ancien emplacement i est la valeur du i^{e} élément de $\vec{v}$.

Exemple : $reorder((0),(2,0,1))$

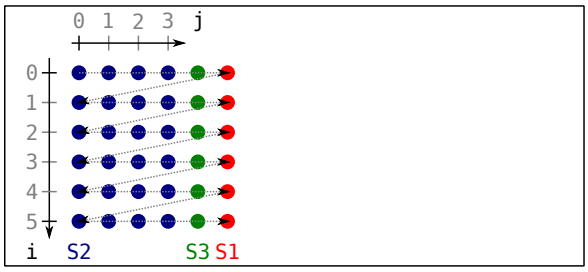
```
for (i = 0; i < N; ++i)
{
    S1(i);
    for (j = 0; j < M; ++j) { S2(i, j); }
    S3(i);
}
```

$\vec{\beta}_{\theta_{S1}} = (0,0)$
$\vec{\beta}_{\theta_{S2}} = (0,1,0)$
$\vec{\beta}_{\theta_{S3}} = (0,2)$




```
1 for (i = 0; i < N; ++i)                    1
2 {                                          2
3     for (j = 0; j < M; ++j) { S2(i, j); }  3
4     S3(i);                                 4
5     S1(i);                                 5
6 }                                          6
```

$\vec{\beta}_{\theta_{S1}} = (0,\mathbf{2})$
$\vec{\beta}_{\theta_{S2}} = (0,\mathbf{0},0)$
$\vec{\beta}_{\theta_{S3}} = (0,\mathbf{1})$



5.1.2 Fusionner deux boucles

Syntaxe : $fuse\_next(\vec{\rho})$
$fuse\_next(\beta$-prefixe$)$

Effet :
$\mathit{offset} \leftarrow \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho},*}} \left(\vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1}\right)$ *offset* correspond à la valeur maximale des $\vec{\beta}_{\mathcal{T}}$ des instructions (et des boucles si on les considère comme des instructions) qui sont directement dans la boucle désignée par $\vec{\rho}$
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},next}, \vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} + \mathit{offset} + 1$ On modifie la valeur de chaque $\vec{\beta}_{\mathcal{T}}$ qui sont dans la

boucle suivante pour que toutes les instructions soient après celle de la boucle désignée par $\vec{\rho}$ après la fusion
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},>}, \vec{\beta}_{\mathcal{T},\dim \vec{\rho}} \leftarrow \vec{\beta}_{\mathcal{T},\dim \vec{\rho}} - 1$ On décrémente la valeur de chaque $\vec{\beta}_{\mathcal{T}}$ des instructions qui sont dans les boucles suivantes pour fusionner nos boucles et garder l'ordre naturel des $\vec{\beta}_{\mathcal{T}}$ des instructions qui sont dans les boucles suivantes

Conditions :
$\exists \vec{\beta}_{\mathcal{T}} : \vec{\beta}_{\mathcal{T},1..\dim \vec{\rho}-1} = \vec{\rho}_{1..\dim \vec{\rho}-1}$ et $\vec{\beta}_{\mathcal{T},\dim \vec{\rho}} = \vec{\rho}_{\dim \vec{\rho}} + 1$ et $\dim \vec{\beta}_{\mathcal{T}} > \dim \vec{\rho}$ Il existe une instruction (ou boucle) qui est au même niveau que la boucle désignée par $\vec{\rho}$ et cette instruction (ou boucle) est dans la boucle juste après celle désignée par $\vec{\rho}$

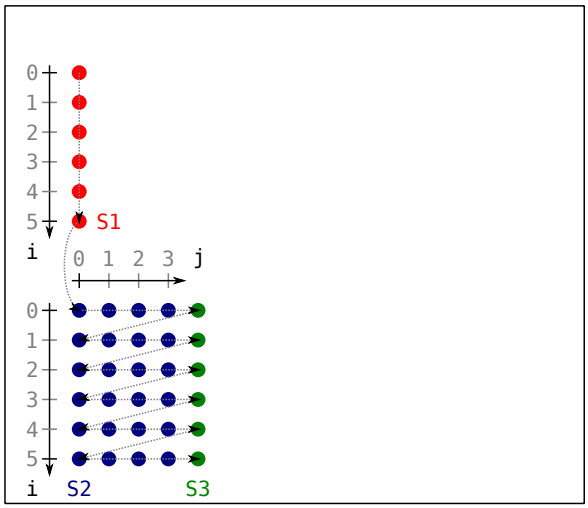
Définition :
Fusionne la boucle désignée par $\vec{\rho}$ avec la boucle suivante. L'ordre des statements dans la boucle fusionnée est le même qu'avant la fusion. Les valeurs des β restent continues.

Exemple : $fuse\_next((0))$

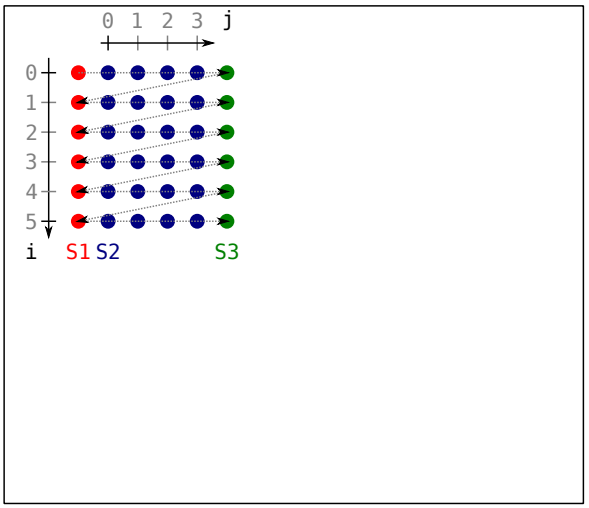
```
for (i = 0; i < N; ++i)
{
    S1(i);
}
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j) { S2(i, j); }
    S3(i);
}
```

```
1 for (i = 0; i < N; ++i)                          1
2 {                                                2
3     S1(i);                                       3
4                                                  4
5                                                  5
6                                                  6
7     for (j = 0; j < M; ++j) { S2(i, j); }        7
8     S3(i);                                       8
9 }                                                9
```

$\vec{\beta}_{\theta_{S1}} = (0, 0)$
$\vec{\beta}_{\theta_{S2}} = (1, 0, 0)$
$\vec{\beta}_{\theta_{S3}} = (1, 1)$



$\vec{\beta}_{\theta_{S1}} = (0, 0)$
$\vec{\beta}_{\theta_{S2}} = (\mathbf{0}, \mathbf{1}, 0)$
$\vec{\beta}_{\theta_{S3}} = (\mathbf{0}, \mathbf{2})$



5.1.3 Distribuer deux boucles

Syntaxe : $distribute(\vec{\rho}, n)$
$distribute(\beta$-prefixe, nombre d'instructions ou de boucles à garder dans la boucle à distribuer$)$

Effet :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},>}, \vec{\beta}_{\mathcal{T},\dim \vec{\rho}} \leftarrow \vec{\beta}_{\mathcal{T},\dim \vec{\rho}} + 1$ On modifie la valeur de chaque $\vec{\beta}_{\mathcal{T}}$ des instructions qui sont dans les boucles après celle désignée par $\vec{\rho}$
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}} : \vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} < n, \vec{\beta}_{\mathcal{T},\dim \vec{\rho}} \leftarrow \vec{\beta}_{\mathcal{T},\dim \vec{\rho}} + 1$ et $\vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} - n$ On modifie la valeur de chaque $\vec{\beta}_{\mathcal{T}}$ des instructions qui sont dans la boucle désignée par $\vec{\rho}$ et qui doivent en sortir (celles après l'instruction à la n^{e} position) pour : les placer dans une nouvelle boucle ; et réécrire les valeurs des $\vec{\beta}_{\mathcal{T}}$ au niveau de la nouvelle boucle en commençant à 0.

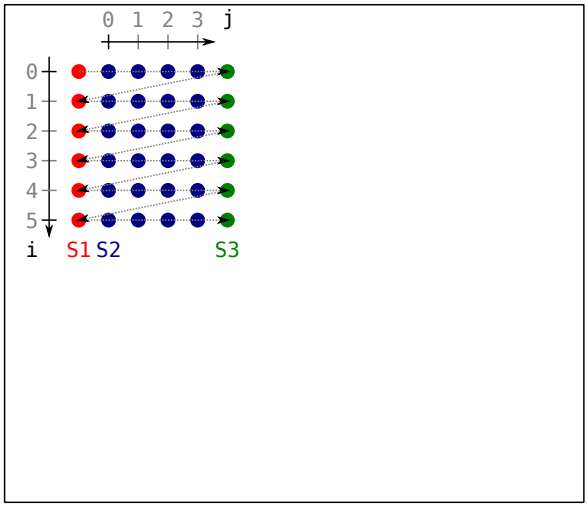
Conditions :
$1 \leq n \leq \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho},*}} \left(\vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1} \right)$ n est inférieur au nombre d'instructions ou de boucles qui sont directement dans la boucle désignée par $\vec{\rho}$ et vaut au moins un.

Définition :
Découpe la boucle en deux en gardant n instructions ou boucles dans la boucle originale. Les valeurs des $\vec{\beta}_{\mathcal{T}}$ sont normalisées.

Exemple : *distribute*$((0), 2)$

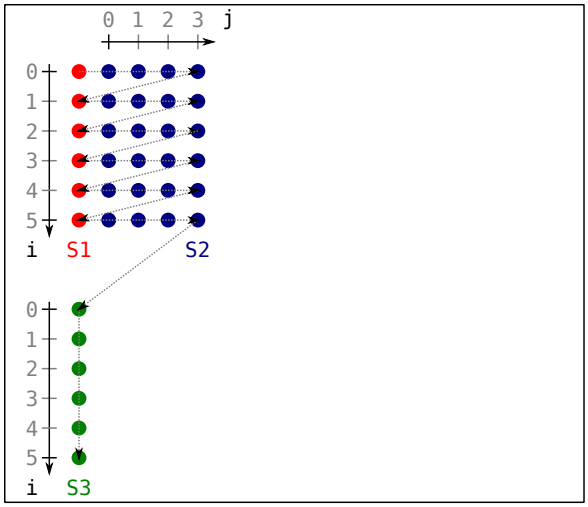
```
for (i = 0; i < N; ++i)
{
    S1(i);
    for (j = 0; j < M; ++j) { S2(i, j); }



    S3(i);
}
```

$\vec{\beta}_{\theta_{S1}} = (0, 0)$
$\vec{\beta}_{\theta_{S2}} = (0, 1, 0)$
$\vec{\beta}_{\theta_{S3}} = (0, 2)$




```
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);
}
```

$\vec{\beta}_{\theta_{S1}} = (0, 0)$
$\vec{\beta}_{\theta_{S2}} = (0, 1, 0)$
$\vec{\beta}_{\theta_{S3}} = (\mathbf{1}, \mathbf{0})$



Ces trois transformations (*reorder*, *fuse_next*, *distribute*) modifient uniquement les valeurs des $\vec{\beta}_{\mathcal{T}}$ dans les relations d'ordonnancement des instructions concernées. Même si on perd la notion de boucles dans la représentation polyédrique, on peut dire qu'à haut-niveau ces transformations permettent de manipuler les boucles et l'ordre des instructions. Les valeurs des $\vec{\beta}_{\mathcal{T}}$ déjà présentes peuvent donc être modifiées arbitrairement grâce à ces transformations dans le respect des contraintes choisies lors de la présentation du modèle polyédrique dans la section 2.2.3.

Les douzes transformations suivantes modifient les valeurs des dimensions de sortie $\vec{\alpha}_{\mathcal{T}}$. Ces valeurs nous permettent de réorganiser les instances des instructions entre elles. Les transformations présentées sont les transformations syntaxiques *reverse*, *shift*, *skew*, *reshape*, *grain*, *densify*, *interchange*, *stripmine*, *linearize*, *index_set_split*, *collapse*.

5.1.4 Renverser une boucle

Syntaxe : $reverse(\vec{\rho})$
$reverse(\beta\text{-prefixe})$

Effet :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}, \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \mapsto -\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$ On substitue la dimension $\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$ des instructions de la boucle désignée par $-\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$.

Effet (en utilisant $skew$) :
$skew(\vec{\rho}, \dim \vec{\rho}, -2)$

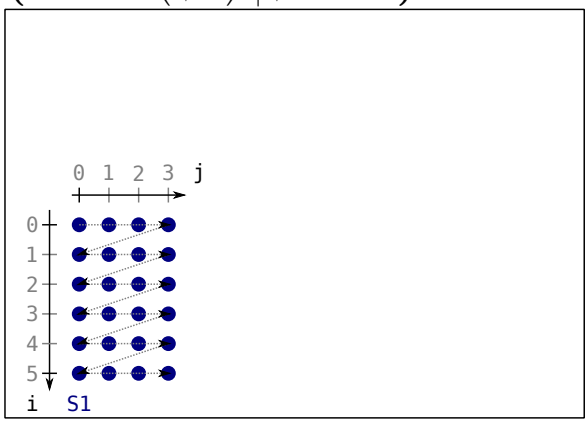
Définition :
Renverse les instances des instructions qui sont dans la boucle désignée par $\vec{\rho}$. On peut l'exprimer avec un cas particulier de $skew$.

Exemple : $reverse((0))$

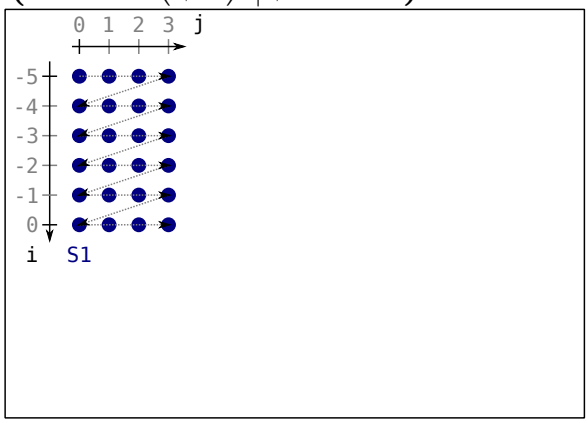
```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

```
1 for (i = -N + 1; i <= 0; ++i)
2 {
3     for (j = 0; j < M; ++j)
4     {
5         S1(-i, j);
6     }
7 }
```

$$\theta_{S1}(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$$\theta_{S1}(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & \mathbf{-i} \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.5 Déplacer des instances des instructions

Syntaxe : $shift(\vec{\rho}, i, amount)$
$shift(\beta\text{-prefixe}$, indice de la dimension α à déplacer,
quantité potentiellement paramétrique pour le déplacement$)$

Effet générique (si la dimension $\vec{\alpha}_{\mathcal{T},i}$ est définie explicitement ou implicitement) :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}, \vec{\alpha}_{\mathcal{T},i} \mapsto \vec{\alpha}_{\mathcal{T},i} + amount$ On substitue la dimension $\vec{\alpha}_{\mathcal{T},i}$ des instructions de la boucle désignée par $\vec{\rho}$ par elle-même plus la constante $amount$.

Effet spécifique (si la dimension $\vec{\alpha}_{\mathcal{T},i}$ est définit explicitement) :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}, \vec{\alpha}_{\mathcal{T},i} \leftarrow \vec{\alpha}_{\mathcal{T},i} + amount$ On ajoute à la dimension $\vec{\alpha}_{\mathcal{T},i}$ des instructions de la boucle désignée

par $\vec{\rho}$ la constante *amount*.

Conditions :
$1 \leq i \leq \dim \vec{\rho} \leq \dim \vec{\alpha}_{\mathcal{T}}$ $\vec{\alpha}_{\mathcal{T},i}$ est valide et présent dans la boucle désignée par $\vec{\rho}$
$amount = \vec{v} \times \vec{p} + N, \vec{v} \in \mathbb{Z}^{\dim \vec{p}}, C \in \mathbb{Z}$ *amount* est une expression affine qui est une combinaison des paramètres et d'une constante

Définition :
On déplace toutes les instances des instructions de la boucle désignée par $\vec{\rho}$ dans l'espace d'itération de la constante *amount*. Si la dimension $\vec{\alpha}_{\mathcal{T},i}$ est définie explicitement, sa définition apparaît dans une seule équation que l'on peut simplement modifier en ajoutant *amount*. Si elle est définie implicitement, il faut effectuer une substitution dans toutes les inégalités où $\vec{\alpha}_{\mathcal{T},i}$ apparaît afin de préserver les autres dimensions.

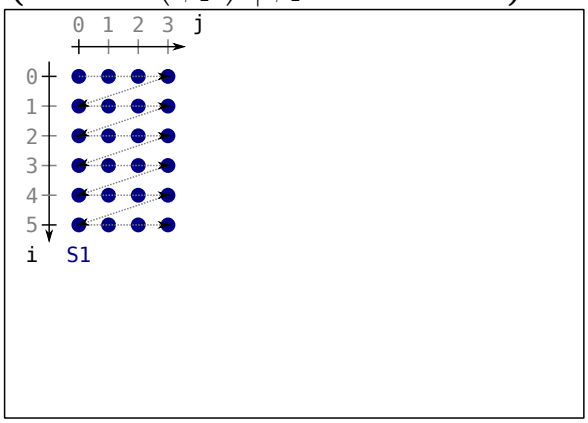
Exemple : $shift((0,0), i, (N, -2))$

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

```
1 for (i = N - 2; i < 2 * N - 2; ++i)
2 {
3     for (j = 0; j < M; ++j)
4     {
5         S1(-N + i + 2, j);
6     }
7 }
```

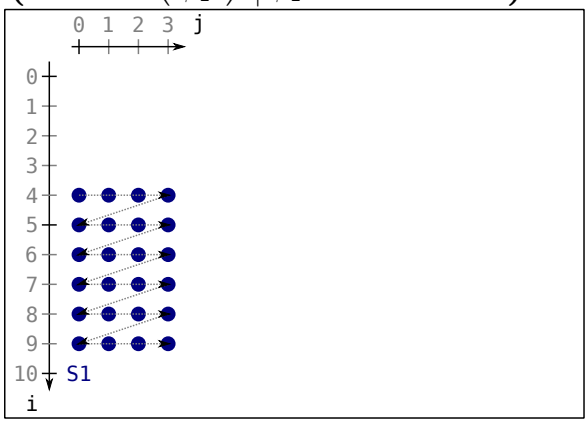
$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i + \mathbf{N - 2} \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.6 Incliner les instances des instructions

Syntaxe : $skew(\vec{\rho}, i, factor)$
$skew(\beta$-prefixe, indice de la dimension α pour le déplacement, coefficient pour le déplacement$)$

Effet :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}, \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} + factor \cdot \vec{\alpha}_{\mathcal{T},i}$ On substitue la dimension $\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$ des instructions de la boucle désignée par $\vec{\rho}$ par une autre multipliée par $factor$.

Définition :
Incline les itérations de la boucle dim $\vec{\rho}$ par un facteur de l'indice de boucle i.

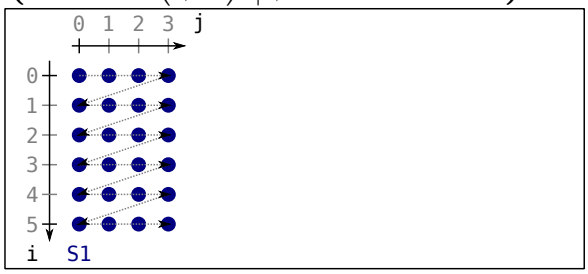
Exemple : $skew((0,0),i,1)$

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

```
1 for (i = 0; i < N + M - 1; ++i)                1
2 {                                              2
3     for (j = 0; j < M + i; ++j)                3
4     {                                          4
5         S1(i, j - i);                          5
6     }                                          6
7 }                                              7
```

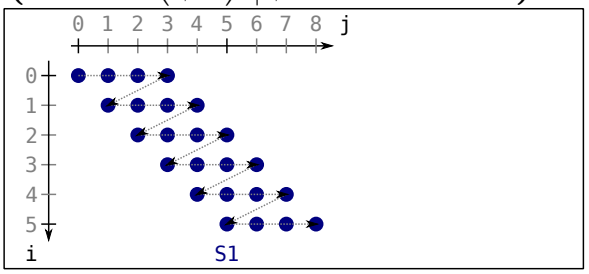
$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j - \mathbf{1} \times \boldsymbol{\alpha_1} \\ \beta_2 & = & 0 \end{array} \right\}$$



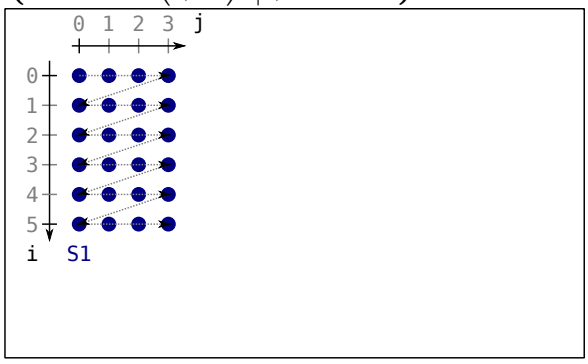
Exemple : $skew((0),j,1)$

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

```
1 for (i = 0; i < N + M - 1; ++i)                1
2 {                                              2
3     for (j = 0; max(0, -N + i + 1); ++j)       3
4     {                                          4
5         S1(i - j, j);                          5
6     }                                          6
7 }                                              7
```

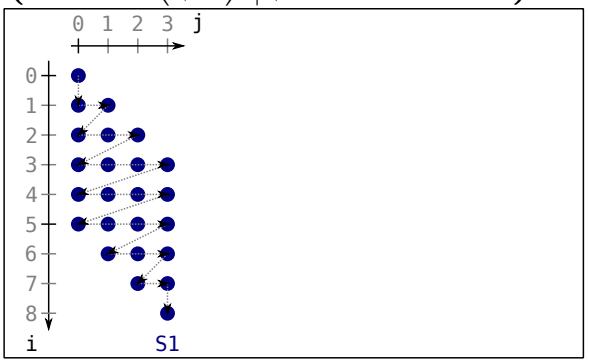
$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i - \mathbf{1} \times \boldsymbol{\alpha_2} \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.7 Déformer les instances des instructions

Syntaxe : $reshape(\vec{\rho}, i, factor)$
$reshape(\beta$-prefixe, indice de la dimension α pour la déformation, coefficient pour la déformation)

Effet :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}, \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} \mapsto \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}} + factor \cdot \vec{\iota}_{\mathcal{T},i}$ On substitue la dimension $\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$ des instructions de la boucle désignée par $\vec{\rho}$ par la dimension $\vec{\iota}_{\mathcal{T},i}$ multipliée par $factor$.

Conditions :
Si les dimensions de sortie $\alpha_{\mathcal{T},i}$ ou $\alpha_{\mathcal{T},\dim \vec{\rho}}$, ou les deux sont définies implicitements, il n'y a pas de condition.
Mais si elles sont définies explicitement : $\alpha_{\mathcal{T},i} = \vec{v} \cdot \vec{\iota}^{\mathcal{T}} + f_1(\vec{p}) + C_1$ et $\alpha_{\mathcal{T},\dim \vec{\rho}} = \vec{w} \cdot \vec{\iota}^{\mathcal{T}} + f_2(\vec{p}) + C_2$, $\left(\nexists d : d = \frac{\vec{v}_j}{\vec{w}_j} \forall j \neq i\right) \vee \left(\frac{\vec{v}_j}{\vec{w}_j + k\vec{v}_j} \neq d\right)$. alors $\alpha_{\mathcal{T},\dim \vec{\rho}} = w \cdot \vec{\iota}_i + f_2(\vec{p}) + C_2$, $w \neq -k$
La transformation ne doit donner ni des équations linéairement dépendantes, ni des dimensions définies explicitement contantes.

Définition :
On reforme l'espace contenant toutes les instances des instructions de la boucle désignée par $\vec{\rho}$ en fonction de l'itérateur de boucle original i multiplié par un facteur *factor*. Pour que la transformation respecte la condition de validité globale, les conditions de la transformations assurent la condition d'existence de l'ordonnancement.

Exemple :
Pour les cas basiques, les exemples de *reshape* sont les mêmes que ceux de *skew* car les dimensions de sortie α sont directement définies en fonction des dimensions d'entrée ι.

5.1.8 Espacer les instances des instructions

Syntaxe : $grain(\vec{\rho}, i, factor)$
$grain(\beta$-prefixe, indice de la dimension α pour l'espacement, coefficient pour l'espacement$)$

Effet :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}, \forall$(in)égalité $\left(\vec{u} \cdot \vec{\alpha}^T + f(\vec{\iota}, \vec{p}) + C \geq 0\right) \in \mathcal{T} : \vec{u}_i \neq 0$, remplacer l'(in)égalité par
$\left(factor \cdot \vec{u} \cdot \vec{\alpha}^T - (factor - 1) \cdot \vec{u}_i \cdot \vec{\alpha}_i + factor \cdot f(\vec{\iota}, \vec{p}) + factor \cdot C \geq 0\right)$
On identifie les (in)équations avec une certaine forme afin d'introduire le facteur d'espacement aux endroits adéquats

Conditions :
$1 \leq i \leq \dim \vec{\rho} \leq \dim \vec{\alpha}_{\mathcal{T}}$ $\vec{\alpha}_{\mathcal{T},i}$ est valide et présent dans la boucle désignée par $\vec{\rho}$
$factor \geq 1$ Le $factor$ est supérieur ou égal à 1

Définition :
Pour chaque instruction dans la boucle désignée par $\vec{\rho}$, on espace les instances des instructions en les exécutant toutes les $n \times factor$ fois.

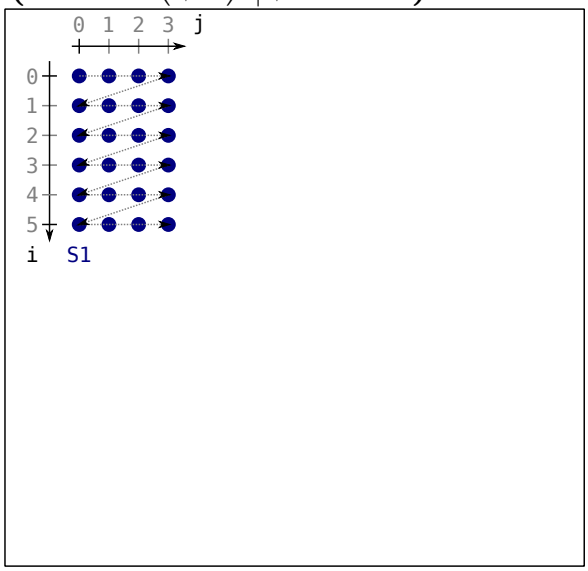
Exemple : $grain((0,0), i, 3)$

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

```
1 for (i = 0; i < 3 * N; ++i)                          1
2 {                                                    2
3     for (j = 0; j < M; ++j)                          3
4     {                                                4
5         if (i % 3 == 0) { S1(i/3, j); }              5
6     }                                                6
7 }                                                    7
```

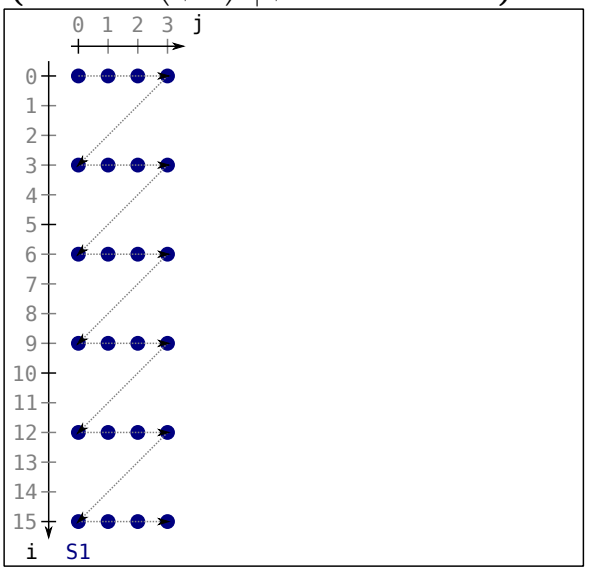
$\theta_{S1}(\mathrm{N}, \mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N}, \mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & \mathbf{3} \times i \\ \beta_1 & = & 0 \\ \mathbf{3} \times \alpha_2 & = & \mathbf{3} \times j \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.9 Rapprocher des instances des instructions

Syntaxe : $densify(\vec{\rho}, i)$
$densify(\beta\text{-prefixe, indice de la dimension } \alpha \text{ à rapprocher au maximum})$

Effet :
$\exists\, factor \in \mathbb{N} : factor > 1, \forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}$,
$\forall (in)égalité \left(factor \cdot \vec{u} \cdot \vec{\alpha}^T - (factor - 1) \cdot \vec{u}_i \cdot \vec{\alpha}_i + factor \cdot f(\vec{\iota}, \vec{p}) + factor \cdot C \geq 0\right) \in \mathcal{T}$,
$\vec{u} \in \mathbb{Z}^{\dim \vec{\alpha}}, C \in \mathbb{Z}, f$ est une fonction linéaire
remplacer l'(in)égalité par $\left(\vec{u} \cdot \vec{\alpha}^T + f(\vec{\iota}, \vec{p}) + C \geq 0\right)$
Tout comme *grain*, $densify$ identifie les (in)équations avec une forme particulière (elles ont été *grainées*) afin d'enlever le facteur d'espacement maximum

Conditions :
$1 \leq i \leq \dim \vec{\rho} \leq \dim \vec{\alpha}_{\mathcal{T}}$ $\vec{\alpha}_{\mathcal{T},i}$ est valide et présent dans la boucle désignée par $\vec{\rho}$

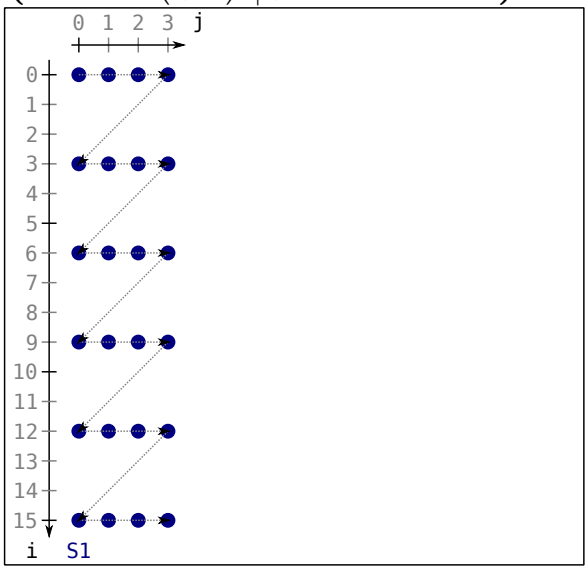
Définition :
Pour chaque instruction dont le β-prefixe est $\vec{\rho}$, on enlève tout espace entre deux exécutions consécutives des instances pour la dimension $\vec{\alpha}_{\mathcal{T},i}$.

Exemple : $densify((0,0), i)$

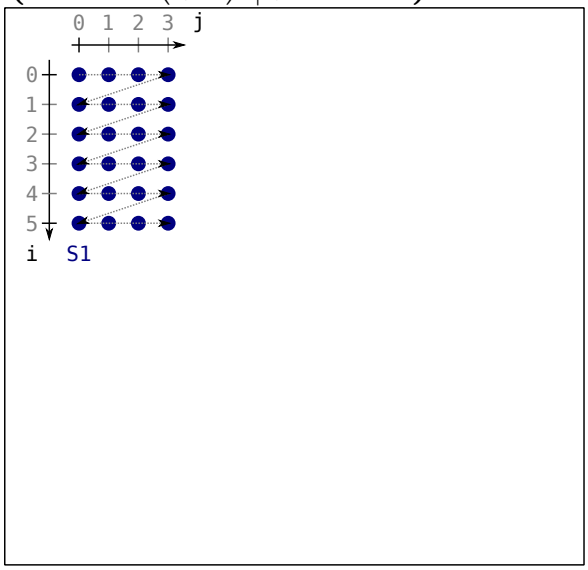
```
for (i = 0; i < 3 * N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        if (i % 3 == 0) { S1(i/3, j); }
    }
}
```

```
1 for (i = 0; i < N; ++i)                1
2 {                                      2
3     for (j = 0; j < M; ++j)            3
4     {                                  4
5         S1(i, j);                      5
6     }                                  6
7 }                                      7
```

$\theta_{S1}(\mathrm{N},\mathrm{M}) =$
$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & \mathbf{3} \times i \\ \beta_1 & = & 0 \\ \mathbf{3} \times \alpha_2 & = & \mathbf{3} \times j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$
$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.10 Interchanger deux boucles

Syntaxe : *interchange*$(\vec{\rho}, i)$
interchange(β-prefixe, indice de la dimension α à interchanger avec celle désignée par $\vec{\rho}$)

Effet : $\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}$, $\vec{\alpha}_{\mathcal{T},i} \leftrightarrow \vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$ On substitue la dimension $\vec{\alpha}_{\mathcal{T},i}$ des instructions de la boucle désignée par $\vec{\rho}$ par la dimension $\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$ et on substitue la dimension $\vec{\alpha}_{\mathcal{T},\dim \vec{\rho}}$ des instructions de la boucle désignée par $\vec{\rho}$ par la dimension $\vec{\alpha}_{\mathcal{T},i}$; les deux substitutions se font en même temps

Conditions :
$1 \leq i < \dim \vec{\rho}$ i correspond à la profondeur de la boucle à interchanger avec la boucle désignée par $\vec{\rho}$

Définition :
Interchange la boucle i avec la boucle désignée par $\vec{\rho}$.

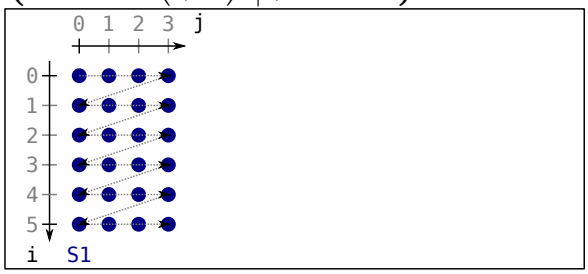
Exemple : *interchange*$((0,0), i)$

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

```
1 for (j = 0; j < M; ++j)                1
2 {                                      2
3     for (i = 0; i < N; ++i)            3
4     {                                  4
5         S1(i, j);                      5
6     }                                  6
7 }                                      7
```

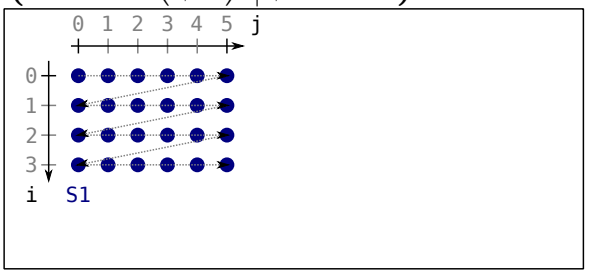
$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & \mathbf{j} \\ \beta_1 & = & 0 \\ \alpha_2 & = & \mathbf{i} \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.11 Décomposer une boucle

Syntaxe : *stripmine*($\vec{\rho}$, *size*)
stripmine(β-prefixe, taille du strip (tuile 1D))

Effet :
$i \leftarrow \dim \vec{\rho}$

$$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*},\ \ \mathcal{T} \leftarrow \left\{ \vec{\imath}_{\mathcal{T}} \rightarrow (\vec{\sigma}_{\mathcal{T},0..2\times i-1}, \beta, \alpha, \vec{\sigma}_{\mathcal{T},2\times i..\dim \vec{\sigma}_{\mathcal{T}}}) \middle| \begin{pmatrix} K_{\mathcal{T}} \\ -size \cdot \vec{\alpha}_{\mathcal{T},i} + \vec{\alpha}_{\mathcal{T},i+1} \\ size \cdot \vec{\alpha}_{\mathcal{T},i} + size - 1 - \vec{\alpha}_{\mathcal{T},i+1} \end{pmatrix} \geq \vec{0} \right\}$$

On ajoute une nouvelle dimension devant la dimention à *stripminer* telle que :
$size \cdot \vec{\alpha}_{\mathcal{T},i} \leq \vec{\alpha}_{\mathcal{T},i+1} \leq size \cdot \vec{\alpha}_{\mathcal{T},i} + size - 1$

Conditions :
$size \geq 1$ *size* est un entier supérieur ou égal à 1

Définition :
Pour chaque instruction dans la boucle désignée par $\vec{\rho}$, on décompose cette boucle en deux boucles consécutives telles que : pour chaque itération de la première boucle, la deuxième boucle itère sur au plus *size* itérations de la boucle originale.

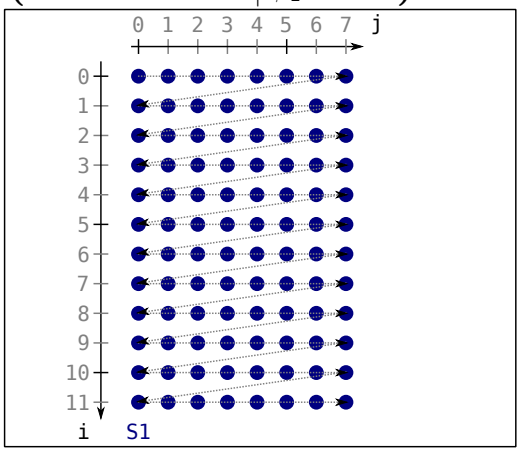
Exemple : *stripmine*((0), 4)

```
for (i = 0;
     i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

```
1  for (s = 0; s < floord(N - 1, 4); ++s)
2  {
3      for (i = 4 * s;
4           i <= min(N - 1, 4 * s + 3); ++i)
5      {
6          for (j = 0; j < M; ++j)
7          {
8              S1(i, j);
9          }
10     }
11 }
```

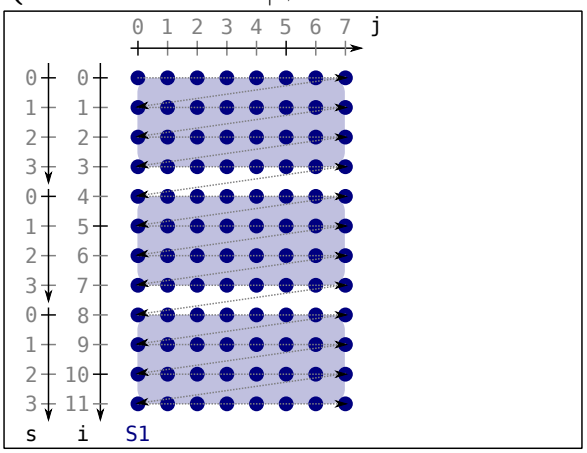
$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{rcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \boldsymbol{\alpha_1} \\ \boldsymbol{\beta_1} \\ \boldsymbol{\alpha_{1'}} \\ \boldsymbol{\beta_{1'}} \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{rcl} \beta_0 & = & 0 \\ -\boldsymbol{s \cdot \vec{\alpha}_{\mathcal{T},1} + \vec{\alpha}_{\mathcal{T},1'}} & \boldsymbol{\geq} & \boldsymbol{0} \\ \boldsymbol{s \cdot \vec{\alpha}_{\mathcal{T},1} + s - 1 - \vec{\alpha}_{\mathcal{T},1'}} & \boldsymbol{\geq} & \boldsymbol{0} \\ \boldsymbol{\beta_1} & = & 0 \\ \boldsymbol{\alpha_{1'}} & = & i \\ \boldsymbol{\beta_{1'}} & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.12 Linéariser une boucle

Syntaxe : $linearize(\vec{\rho})$
$linearize(\beta\text{-prefixe})$

Effet :
$i \leftarrow \dim \vec{\rho}$

$$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*},\ \mathcal{T} \leftarrow \left\{ \vec{\imath}_{\mathcal{T}} \rightarrow (\vec{\sigma}_{\mathcal{T},1..((i-1)\times 2)-1}, \vec{\sigma}_{\mathcal{T},i\times 2..\dim \vec{\sigma}_{\mathcal{T}}}) \,\middle|\, K'_{\mathcal{T}} \geq \vec{0} \right\} : K_{\mathcal{T}} = \begin{pmatrix} K'_{\mathcal{T}} \\ -size \cdot \vec{\alpha}_{\mathcal{T},i} + \vec{\alpha}_{\mathcal{T},i+1} \\ size \cdot \vec{\alpha}_{\mathcal{T},i} + size - 1 - \vec{\alpha}_{\mathcal{T},i+1} \end{pmatrix}$$

On enlève la boucle qui précède celle désignée par $\vec{\rho}$ et on enlève toutes les inéquations qui utilisent cette dimension

Conditions :
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho}}, \begin{pmatrix} -size \cdot \vec{\alpha}_{\mathcal{T},i} + \vec{\alpha}_{\mathcal{T},i+1} \\ size \cdot \vec{\alpha}_{\mathcal{T},i} + size - 1 - \vec{\alpha}_{\mathcal{T},i+1} \end{pmatrix} \in K_{\mathcal{T}}$ La boucle avait été stripminée

Définition :
La boucle désignée par $\vec{\rho}$ est linéarisée en intégrant les itérations de la boucle précédente.

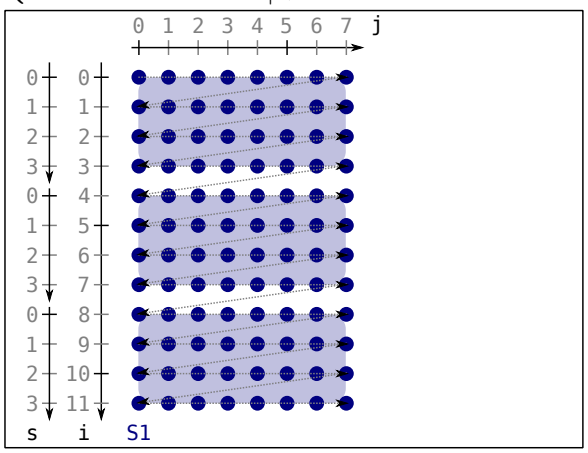
Exemple : $linearize((0,0))$

```
for (s = 0; s < floord(N - 1, 4); ++s)
{
    for (i = 4 * s;
         i <= min(N - 1, 4 * s + 3); ++i)
    {
        for (j = 0; j < M; ++j)
        {
            S1(i, j);
        }
    }
}
```

```

for (i = 0;
     i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        S1(i, j);
    }
}
```

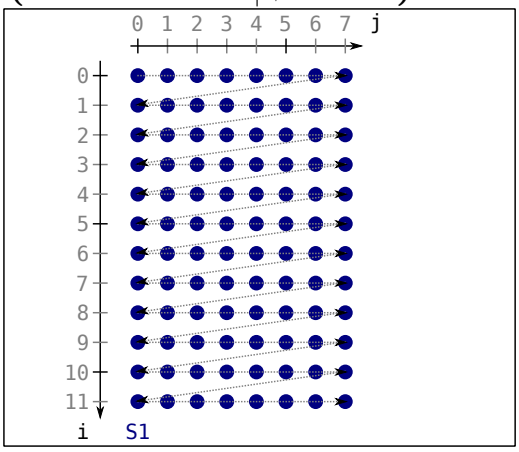
$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \boldsymbol{\alpha_1} \\ \boldsymbol{\beta_1} \\ \boldsymbol{\alpha_{1'}} \\ \boldsymbol{\beta_{1'}} \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \boldsymbol{-s \cdot \vec{\alpha}_{\mathcal{T},1} + \vec{\alpha}_{\mathcal{T},1'}} & \boldsymbol{\geq} & \boldsymbol{0} \\ \boldsymbol{s \cdot \vec{\alpha}_{\mathcal{T},1} + s - 1 - \vec{\alpha}_{\mathcal{T},1'}} & \boldsymbol{\geq} & \boldsymbol{0} \\ \boldsymbol{\beta_1} & = & 0 \\ \boldsymbol{\alpha_{1'}} & = & i \\ \boldsymbol{\beta_{1'}} & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \rightarrow \begin{pmatrix} \beta_0 \\ \\ \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \\ \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



5.1.13 Découper des instances des instructions

Syntaxe : *index_set_split*($\vec{\rho}$, *constraint*)
index_set_split(β-prefixe, contrainte sélectionnant la première partie des instances des instructions)

Effet :
offset $\leftarrow \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho},*}} \left(\vec{\beta}_{\mathcal{T},\dim \vec{\rho}+1}\right)$
$\forall \mathcal{T} \in \mathcal{T}_{\vec{\rho},*}, \mathcal{T} \mapsto \mathcal{T}' \bigcup \mathcal{T}'' : \mathcal{T}' \leftarrow \mathcal{T} \cap constraint, \mathcal{T}'' \leftarrow \mathcal{T} \cap \overline{constraint} \cap \vec{\beta}_{\mathcal{T}'',\dim \vec{\rho}+1} \leftarrow \vec{\beta}_{\mathcal{T}'',\dim \vec{\rho}+1} +$ *offset* $+ 1$ On replace $\mathcal{T}$ par une union de deux relations disjointes dépendantes de *constraint*

Conditions :
$constraint = \vec{u} \times \vec{\alpha}_{\mathcal{T}} + \vec{v} \times \vec{\iota}_{\mathcal{T}} + \vec{w} \times \vec{p} + C$
$\vec{u} \in \mathbb{Z}^{\dim \vec{\alpha}}, \vec{v} \in \mathbb{Z}^{\dim \vec{\iota}}, \vec{w} \in \mathbb{Z}^{\dim \vec{p}}, C \in \mathbb{Z}$ *constraint* est une expression affine qui est une combinaison des dimensions de sortie α, des dimensions d'entrée, des paramètres et d'une constante

Définition :
Sépare les instances des instructions en deux parties selon *constraint*. La contrainte est ajoutée sur les relations convexes concernées et pour chacune d'entre elles, une nouvelle relation convexe est créée avec la contrainte opposée. L'unicité des β est respectée.

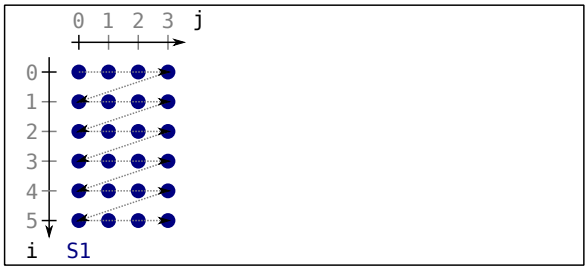
Exemple : *index_set_split*$((0,0), i - j > 0)$

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {


            S1(i, j);









    }
}
```

```
1  for (i = 0; i < N; ++i)                          1
2  {                                                2
3      for (j = 0; j < M; ++j)                      3
4      {                                            4
5          if (i - j > 0)                           5
6          {                                        6
7              S1(i, j); // Part 1                  7
8          }                                        8
9          else                                     9
10         {                                        10
11             S1(i, j); // Part 2                  11
12         }                                        12
13     }                                            13
14 }                                                14
```

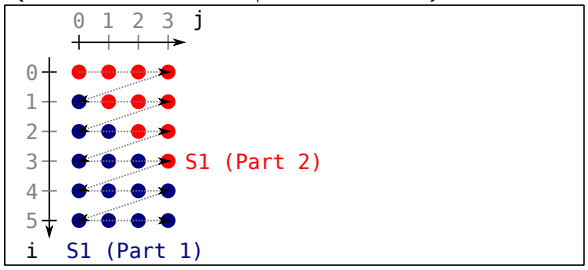
$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{rcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$



$\theta_{S1}(\mathrm{N},\mathrm{M}) =$

$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{rcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \\ \boldsymbol{i-j} & > & 0 \end{array} \right\}$$
$$\cup$$
$$\left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{rcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \\ \boldsymbol{i-j} & \leq & 0 \end{array} \right\}$$



5.1.14 Regrouper des instances des instructions

Syntaxe : $collapse(\vec{\rho})$
$collapse(\beta\text{-prefixe})$

Effet :
$\forall \mathcal{T}', \mathcal{T}'' \in \mathcal{T}_{\vec{\rho},*}$: $\mathcal{T}' = \mathcal{T} \cap constraint \wedge \mathcal{T}'' \approx \mathcal{T} \cap \overline{constraint} \wedge$
$\vec{\beta}_{\mathcal{T}',1..\dim \vec{\rho}} = \vec{\beta}_{\mathcal{T}'',1..\dim \vec{\rho}} = \vec{\rho} \wedge \vec{\beta}_{\mathcal{T}',\dim \vec{\rho}+1} + 1 = \vec{\beta}_{\mathcal{T}'',\dim \vec{\rho}}$,
$(\mathcal{T}' \bigcup \mathcal{T}'') \mapsto \mathcal{T}$
collapse annule *index_set_split* en identifiant la contraite qui a été utilisée pour séparer les instructions avec *index_set_split* et, l'annule en replaçant les différentes unions de la relation en une seule

Conditions :
$constraint = \vec{u} \times \vec{\alpha}_{\mathcal{T}} + \vec{v} \times \vec{\iota}_{\mathcal{T}} + \vec{w} \times \vec{p} + C$; $\vec{u} \in \mathbb{Z}^{\dim \vec{\alpha}}, \vec{v} \in \mathbb{Z}^{\dim \vec{\iota}}, \vec{w} \in \mathbb{Z}^{\dim \vec{p}}, C \in \mathbb{Z}$ *constraint* est une expression affine qui est une combinaison des dimensions de sortie α, des dimensions d'entrée, des paramètres et d'une constante

Définition :
Regroupe les instances des instructions qui avaient été séparées en deux parties disjointes selon *constraint*. Les deux parties de l'union sont remplacées par une seule relation convexe.

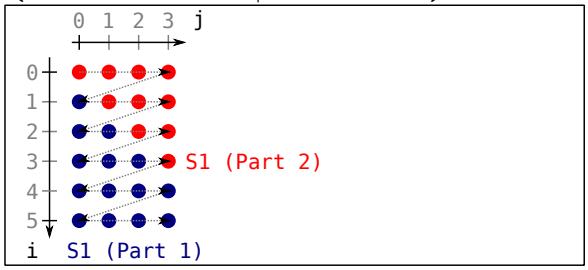
Exemple : $collapse((0,0))$

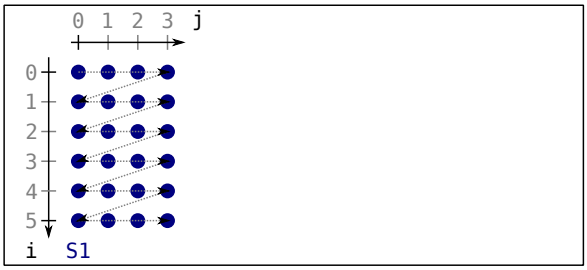
```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {
        if (i - j > 0)
        {
            S1(i, j); // Part 1
        }
        else
        {
            S1(i, j); // Part 2
        }
    }
}
```

```
for (i = 0; i < N; ++i)
{
    for (j = 0; j < M; ++j)
    {

            S1(i, j);


    }
}
```

$$\theta_{S1}(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \\ \boldsymbol{i-j} & > & 0 \end{array} \right\} \cup \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \\ \boldsymbol{i-j} & \leq & 0 \end{array} \right\}$$

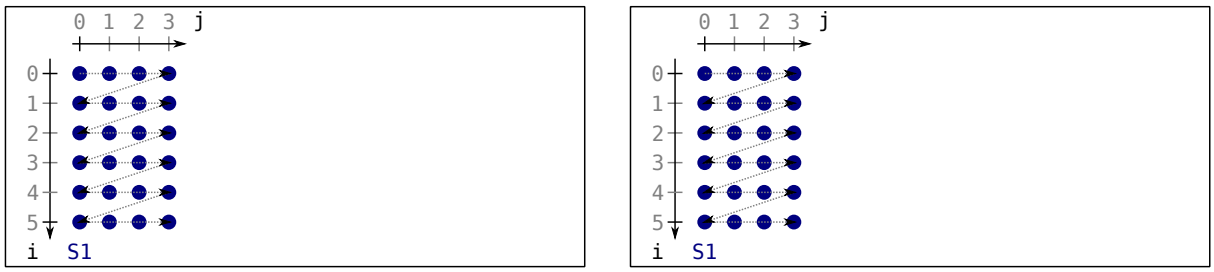
$$\theta_{S1}(\mathrm{N},\mathrm{M}) = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} \beta_0 \\ \alpha_1 \\ \beta_1 \\ \alpha_2 \\ \beta_2 \end{pmatrix} \middle| \begin{array}{lcl} \beta_0 & = & 0 \\ \alpha_1 & = & i \\ \beta_1 & = & 0 \\ \alpha_2 & = & j \\ \beta_2 & = & 0 \end{array} \right\}$$





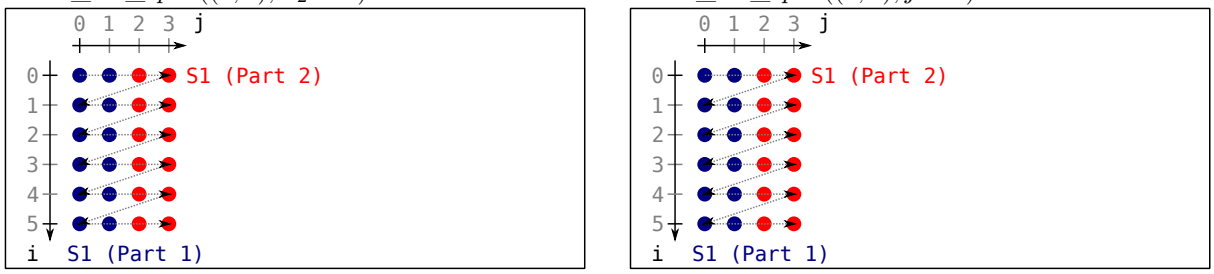
Discussion sur la modification des coefficients des dimensions d'entrée ou de sortie

Certaines transformations utilisent les coefficients des dimensions d'entrée et des dimensions de sortie. Par exemple, la contrainte *constraint* de la transformation *index_set_split* est arbitraire. En revanche, certaines transformations n'utilisent que les dimensions de sortie, comme la transformation *skew* et d'autres, comme *reshape*, n'utilisent que les dimensions d'entrée.



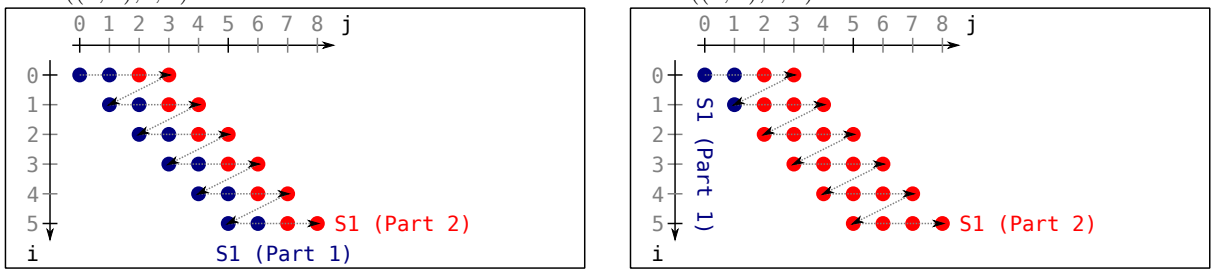
(a) Ordonnancement original

$index\_set\_split((0,0), \alpha_2 < 2)$ $index\_set\_split((0,0), j < 2)$



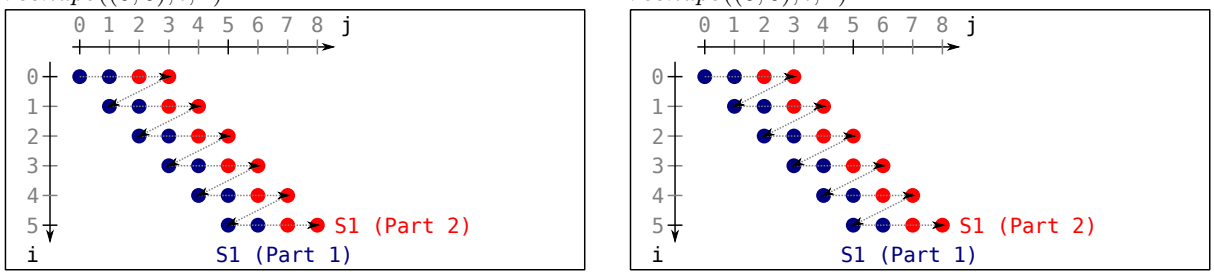
(b) Découpage des instances

$index\_set\_split((0,0), \alpha_2 < 2)$ $index\_set\_split((0,0), j < 2)$
$skew((0,0), i, 1)$ $skew((0,0), i, 1)$



(c) Découpage + Inclinaison des instances

$index\_set\_split((0,0), \alpha_2 < 2)$ $index\_set\_split((0,0), j < 2)$
$reshape((0,0), i, 1)$ $reshape((0,0), i, 1)$



(d) Découpage + Déformation des instances

Figure 5.1 – Composition des transformations *index_set_split*, *skew* et *reshape*

La figure 5.1 montre l'infuence des dimensions d'entrée et des dimensions de sortie sur l'ordonnancement. Les ordonnancements de la figure 5.1a correspondent à l'ordonnancement original. Les ordonnancements de la figure 5.1b sont ceux après la transformation *index_set_split* permettant de découper les instances de l'instruction en deux parties. Dans l'ordonnancement de gauche de la figure 5.1b, le découpage se fait en fonction de la dimension de sortie α_2 alors que dans l'ordonnancement de droite, le découpage se fait en fonction de la dimension d'entrée j. Comme $\alpha_2 = j$, l'ordonnancement est le même. Dans la figure 5.1c, on applique une nouvelle transformation : *skew*. Selon l'utilisation des dimensions d'entrée ou de sortie par *index_set_split*, le comportement n'est pas le même. En effet, dans l'ordonnancement de gauche de la figure 5.1c, le découpage des instances suit la transformation d'inclinaison car il est exprimé en fonction de la dimension de sortie alors que dans l'ordonnancement de droite, le découpage des instances reste fixe car il est exprimé en fonction de la dimension d'entrée. Dans la figure 5.1d, les ordonnancements obtenus sont les mêmes pour les deux *index_set_split* car *reshape* utilise la dimension d'entrée.

Transformations sémantiques

Comme nous avons choisi la possibilité d'attacher des informations aux dimensions de sortie (voir section 2.2.3), certaines transformations dites sémantiques ne font que rajouter ces informations sur les bonnes dimensions de sorties. C'est le cas de la transformation *parallelize*.

5.1.15 Paralléliser les instances des instructions

Syntaxe : *parallelize*($\vec{\rho}$)
parallelize(β-prefixe)

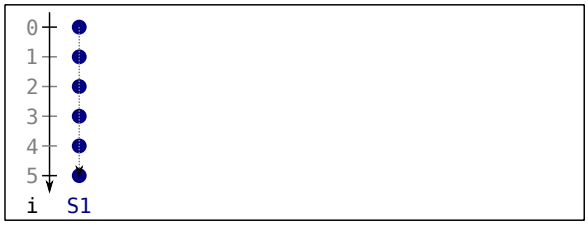
Effet :
Ajoute l'information sémantique nécessaire à la parallélisation de la boucle désignée par $\vec{\rho}$

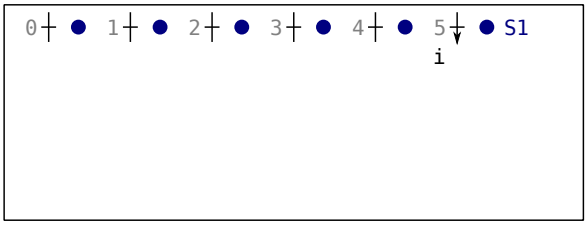
Définition :
Pour chaque instruction qui est dans la boucle désignée par $\vec{\rho}$, on marque la boucle i comme parallèle. Cela permet de demander au générateur de code de paralléliser.

Exemple : *parallelize*((0))

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

```
#pragma omp parallel for
for (i = 0; i < N; ++i)
{
    S1(i);
}
```





D'autres transformations syntaxiques auraient pu être ajoutées. Ces transformations sont des combinaisons des transformations déjà présentées. Par exemple, le tuilage, une transformation haut niveau bien connue, correspond à une composition de *stripmine* et *interchange*.

D'autres transformations sémantiques existent mais ne sont pas détaillées ici. Tout comme *parallelize*, d'autres informations peuvent être ajoutées sur les différentes dimensions de sortie de l'ordonnancement. Lorsque *parallelize* note une dimension de sortie α comme parallèle, le générateur de code peut paralléliser la boucle for correspondante. Si on marque comme parallèle une dimension de sortie β, le générateur peut paralléliser les instructions et boucles à l'intérieur de la dimension parallélisée en utilisant les *sections* d'OpenMP. Ces informations sémantiques peuvent aussi permettre de vectoriser les boucles et les

dérouler. Là aussi ces informations sont destinées au générateur de code car ces transformations ne sont pas des transformations polyédriques dans la mesure où elle ne peuvent pas être représentées dans le modèle polyédrique utilisé.

Pour obtenir les relations d'ordonnancement, c'est-à-dire l'optimisation correspondant à la séquence de transformations, il suffit d'appliquer les transformations les unes après les autres en partant du SCoP original. On obtient alors un SCoP transformé. Pour obtenir le code source final, il suffit d'appeler le générateur de code.

Notre travail porte principalement sur la définition d'un jeu de transformations *composable* et *complet*.

Composable. Comme chaque transformation permet de respecter la condition de validité globale, une séquence de transformations Clay donne toujours, par construction, des relations d'ordonnancement qui respectent la condition de validité globale. La légalité de la séquence de transformations peut être vérifiée une seule fois, après avoir appliqué l'intégralité des transformations de la séquence. Cette propriété est discutée en section 5.2.2.

Complet. Grâce à la composabilité des transformations Clay, il est possible de les combiner afin de modifier arbitrairement une relation d'ordonnancement dans le respect de la validité globale. Cela permet de construire n'importe quel ordonnancement polyédrique : tout ordonnancement respectant la condition de validé globale est donc représentable par une séquence de transformations Clay. Cette propriété est discutée en section 5.2.3.

Ces transformations modifient uniquement les relations d'ordonnancement des instructions. En ignorant le domaine d'itération, nos transformations sont plus robustes et plus générales que les approches précédentes. La section suivante discute de la *validité* et de la *légalité* du SCoP avec de telles transformations.

5.2 Propriétés des transformations

Plusieurs points diffèrent entre nos transformations et les précédentes plateformes de transformations de boucles basées sur le modèle polyédrique. Premièrement, nos transformations lisent et modifient uniquement les relations d'ordonnancement des instructions. Deuxièmement, nos transformations peuvent s'enchaîner les unes après les autres sans vérifier la *validité* de la transformation après chaque modification et sans maintenir un graphe supplémentaire à côté et en plus de la représentation polyédrique (comme le fait URUK). Troisièmement, aucun changement n'est nécessaire pour la méthode permettant de vérifier la *légalité* de la séquence de transformation. Des outils comme Candl peuvent donc être utilisés. Il est inutile de vérifier la légalité après chaque transformation car la vérification de la légalité peut se faire à la fin, c'est-à-dire, juste avant la génération de code, après avoir appliqué toutes les transformations. De plus, une transformation individuelle peut amener le SCoP à ne plus être légal temporairement. (cela peut être corrigé dans la suite des transformations). Quatrièmement, notre ensemble de transformations permet d'exprimer n'importe quel *ordonnancement polyédrique* sans connaissance du modèle polyédrique mais en utilisant les directives de haut niveau présentées. Notre plateforme de transformations haut-niveau basée sur le modèle polyédrique est la première à être complète.

5.2.1 Discussion sur l'invariabilité des domaines d'itération

Nos transformations lisent et modifient uniquement les relations d'ordonnancement des instructions. Il s'agit d'une propriété importante. Cela permet d'ignorer les autres relations permettant de gagner en performance et en généricité. En performance, car aucune vérification ou modification ne doit être faite dans les autres relations du SCoP ou à l'extérieur du SCoP. En généricité, car nos transformations peuvent s'appliquer sur n'importe quel domaine d'itération.

Il s'agit à la fois d'une force mais aussi d'une légère faiblesse. Si on autorisait la lecture des domaines d'itération, on pourrait rendre certaines transformations plus puissantes, plus simples et/ou plus optimisées. Par exemple, la transformation permettant de linéariser pourrait être appliquée sur une dimension issue du domaine d'itération, au lieu de ne servir qu'à annuler une décomposition de boucle faite dans la relation d'ordonnancement. Un autre exemple est la duplication de la relation d'ordonnancement

avec la transformation *index_set_split*. En effet, une relation basique est ajoutée avec la contrainte opposée. Selon le domaine d'itération, cette contrainte n'est peut être jamais vérifiée. Donc, en lisant le domaine d'itération on pourrait supprimer les relations basiques inutiles des relations d'ordonnancement. Comme le générateur de code détecte déjà ces relations basiques inutiles, leur présence n'influence pas la qualité du code généré. Cependant, les supprimer plus tôt permettrait d'éviter d'obtenir des SCoP inutilement verbeux. Un dernier exemple, avec les directives des transformations actuelles : pour travailler sur un ensemble d'instances des instructions, dont la taille est spécifique, on doit d'abord découper les instances des instructions (avec *index_set_split*) et travailler sur les instances voulues (sur la bonne relation basique). Cette méthode est générique mais selon les valeurs du domaine d'itération, cette étape de découpage pour appliquer la transformation voulue n'est pas toujours nécessaire ou peut être plus simple.

Malgré ces limitations, modifier uniquement les relations d'ordonnancement des instructions permet d'appliquer n'importe quelle transformation polyédrique sans polluer et sans se soucier des autres relations du SCoP. Cette contrainte rend notre plateforme plus performante et plus générique et simplifie les vérifications de validité et légalité.

5.2.2 Validité & Légalité

La validité de nos transformations est correcte par construction. Chaque transformation s'assure que la nouvelle relation d'ordonnancement ordonnance bien les mêmes instances des instructions que la relation d'ordonnancement d'avant la transformation. La nouvelle relation d'ordonnancement ordonnance bien chaque instance de l'instruction (qui était ordonnancée) provenant du domaine d'itération. Comme nos transformations ciblent uniquement les domaines des itérations et les relations d'ordonnancements *d-compatibles*, nos transformations permettent de construire uniquement les relations d'ordonnancement respectant la condition de validité globale.

Comme les domaines d'itération restent inchangés, le test de légalité permettant de vérifier si une dépendance est violée ou non effectué par un outil d'analyse des dépendances comme Candl, fonctionne directement (*out of the box*). Il suffit de lui donner le SCoP original et le SCoP après transformation pour obtenir le résultat.

De ce fait, notre jeu de transformations est directement composable. Combiner les transformations syntaxiques entre elles ne demande pas de vérification supplémentaire autre que le respect des conditions venant de chaque transformation. Appliquer plusieurs transformations sémantiques ne pose pas de souci non plus. En revanche, appliquer une transformation syntaxique qui modifie une dimension de sortie portant de l'information sémantique est plus délicat. Par défaut, si on applique une transformation syntaxique sur une telle dimension de sortie, l'information sémantique est supprimée. Ce comportement est motivé par l'exemple suivant : si on fusionne deux boucles dont une ou les deux sont marquées comme parallèles, la parallélisation de la boucle fusionnée n'est pas forcément légale. Oleksandr Zinenko a implémenté un outil [1] permettant de vérifier si une boucle est directement parallèle utilisant Candl. Comme nos transformations n'utilisent que les relations d'ordonnancement, nous ne n'utilisons pas actuellement cet outil mais des futurs travaux porteront sur la combinaison des transformations syntaxes et sémantiques.

Même si nos transformations se limitent aux relations d'ordonnancement respectant la condition de validité globale, elles permettent de construire l'ensemble de ces relations et, par extension, permettent de construire n'importe quel *ordonnancement polyédrique*.

5.2.3 Complétude du jeu de transformation

Chaque transformation permet de modifier une partie spécifique de l'ordonnancement. En combinant ces transformations, on peut alors modifier les relations d'ordonnancement de façon arbitraire et donc, de construire n'importe quel ordonnancement polyédrique respectant la condition de validité globale.

La relation d'ordonnancement se décompose en trois parties différentes. La première partie concerne les valeurs des β-vecteurs. Cette partie peut être naturellement modifiée avec *reorder*, *fuse* et *distribute* mais également avec les transformations *stripmine* et *linearize* ainsi que les transformations *index_set_split*

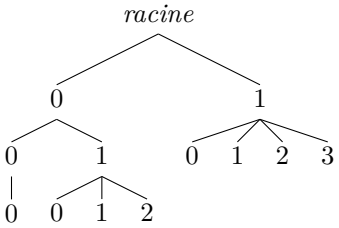
1. https://github.com/ftynse/clope

et *collapse*. La deuxième partie concerne les équations, sans celles qui portent sur les dimensions β. Ces équations peuvent être modifiées avec les transformations *shift*, *skew* et *reshape*. La troisième et dernière partie concerne les inéquations. Ces dernières, plus complexes, peuvent être modifiées avec les transformations *stripmine*, *linearize*, *index_set_split* et *collapse*.

Modification des β-vecteurs

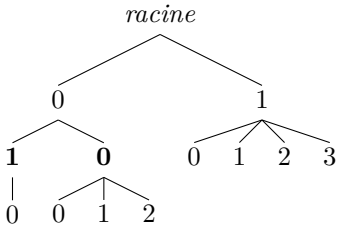
Les équations décrivant les valeurs des β-vecteurs permettent d'encoder la position des instructions dans les différentes boucles. On peut représenter les valeurs des β-vecteurs avec un arbre.

Par exemple, les β-vecteurs suivants : $(0,0,0)$, $(0,1,0)$, $(0,1,1)$, $(0,1,2)$ et $(1,0)$, $(1,1)$, $(1,2)$, $(1,3)$ peuvent être représentés avec cet arbre :

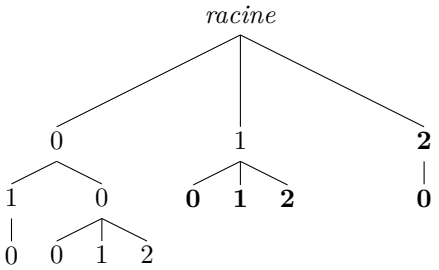


Les transformations *reorder*, $fuse$ et *distribute* impactent directement sur les valeurs des β-vecteurs et uniquement sur ces valeurs. Cela permet donc de changer arbitrairement et facilement les valeurs des β-vecteurs.

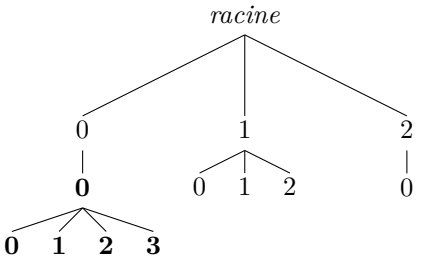
Étant donné un nœud de l'arbre, la transformation *reorder* permet de changer les valeurs des β-vecteurs de tous les fils. Dans notre exemple, $reorder((0),(1,0))$ donne l'arbre suivant. On remarque que les sous-arbres $(0,0)$ et $(1,0)$ ont inversé leur position :



La transformation *distribute* permet de séparer un nœud en deux nœuds en partageant les fils arbitrairement. Dans notre exemple, $distribute((1),3)$ donne l'arbre suivant. On remarque que le sous-arbre (1) avait quatre sous-arbres fils $(1,0)$, $(1,1)$, $(1,2)$, et $(1,3)$. Après la transformation, le sous-arbre (1) n'a plus que trois fils $(1,0)$, $(1,1)$, $(1,2)$ et l'ancien fils $(1,3)$ est devenu le sous-arbre $(2,0)$:

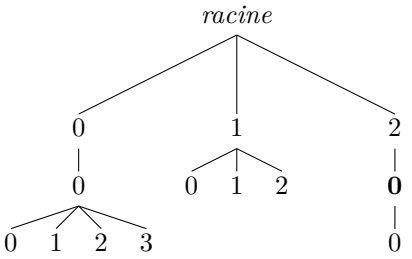


Au contraire, *fuse_next* rassemble deux nœuds en un seul. Dans notre exemple, $fuse\_next((0,1))$ donne l'arbre suivant. On remarque que les sous-arbres $(0,1)$ et $(0,0)$ qui avaient respectivement un fils $(0,1,0)$ et trois fils $(0,0,0)$, $(0,0,1)$ et $(0,0,2)$ ont fusionné pour donner l'arbre $(0,0)$ qui a maintenant quatre fils $(0,0,0)$, $(0,0,1)$, $(0,0,2)$ et $(0,0,3)$:

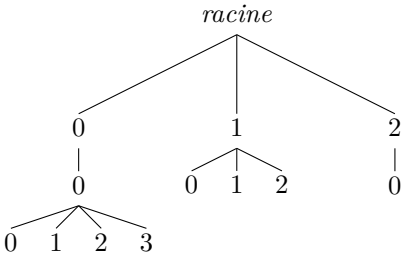


Grâce à ces transformations, *reorder*, *distribute* et $fuse\_next$, il est possible de changer les valeurs des β-vecteurs arbitrairement. En revanche, les transformations *stripmine* et *linearize* qui permettent, respectivement, d'ajouter et supprimer une α-dimension, ajoutent et suppriment une β-dimension par effet de bord.

Dans notre exemple, on peut stripminer la boucle désignée par la β-boucle (2). Cette boucle contient une seule instruction, identifiée par le β-vecteur $(2, 0)$. Si on applique la transformation $stripmine((2), i, 32)$, on obtient l'arbre suivant. Les valeurs i et 32 qui correspondent respectivement à la dimension *alpha* à stripminer et à la taille des strips (tuiles 1D) n'apparaissent pas ici car ces paramètres ne modifient pas les valeurs des β-vecteurs. On remarque uniquement qu'une nouvelle β-dimension est ajoutée et donc que le nouveau β-vecteur de l'ancienne instruction $(2, 0)$ est maintenant $(2, 0, 0)$:

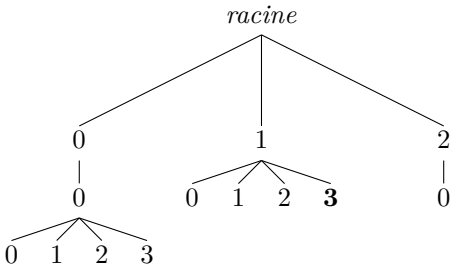


La transformation *collapse* permet d'annuler ce stripmine. Il suffit d'appeler la directive $collapse((2, 0))$ pour obtenir l'arbre suivant dans notre exemple. On remarque qu'il s'agit du même arbre qu'avant le stripmine :

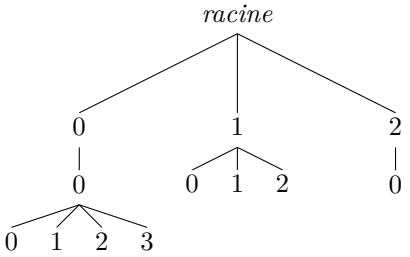


Les dernières transformations qui modifient les β-vecteurs sont $index\_set\_split$ et *collapse*. Ces deux transformations permettent respectivement, d'augmenter et de diminuer le nombre de feuilles de l'arbre.

Si on applique la transformation $index\_set\_split((1, 2), i > 32)$, on obtient l'arbre suivant. On remarque ici aussi que le paramètre $j > 32$ n'apparaît pas car il ne modifie pas les β-vecteurs. L'ajout d'une relation basique dans la relation d'ordonnancement s'accompagne par l'ajout d'un nœud dans l'arbre. L'instruction anciennement désignée par le β-vecteur $(1, 2)$ a maintenant deux β-vecteurs $(1, 2)$ et $(1, 3)$. Ces deux β-vecteurs sont dans deux relations basiques de la même relation d'ordonnancement de la même instruction :



La transformation *collapse*((1, 2)) élimine toutes les relations basiques introduites par *index_set_split*. On obtient donc l'arbre suivant. On remarque que, comme lors de l'annulation du *stripmine* par *collapse*, l'arbre est le même que celui d'avant la transformation *index_set_split* :



Grâce aux transformations *reorder*, $fuse$, *distribute*, *stripmine*, *linearize*, *index_set_split* et *collapse*, on peut modifier l'arbre arbitrairement et donc, on peut modifier les β-vecteurs arbitrairement. Les β-vecteurs sont exprimés avec des équations portant sur les dimensions de sortie β. Dans la section suivante, on s'intéresse aux autres équations, celles portant sur les dimensions de sortie α.

Modification des équations

Si on utilise la *forme de sortie* pour la relation d'ordonnancement, chaque équation portant sur les dimensions de sortie α définit à elle seule une dimension de sortie *explicite*. Cette équation est de la forme $\alpha_{explicit} = \vec{u} \times \vec{\alpha}^T + \vec{v} \times \vec{i}^T \vec{w} \times \vec{p}^T + C$. On dispose des transformations $shift$, *reshape* et *skew* pour modifier les différentes parties de l'équation arbitrairement.

La transformation $shift$ permet de modifier arbitrairement les coefficients $\vec{w}$ et C, c'est-à-dire, les coefficients des paramètres et la constante.

La transformation *reshape* permet de modifier arbitrairement les coefficients $\vec{v}$, c'est-à-dire, les coefficients des dimensions d'entrée.

Comme l'équation définit une α-dimension explicite, la transformation *skew* permet de modifier les coefficients $\vec{u}$, c'est-à-dire, les coefficients des dimensions de sortie α. En effet, cette transformation permet d'ajouter une valeur choisie au coefficient $\vec{u}$. Pour choisir un coefficient y arbitraire et qu'un coefficient x soit déjà présent, il faut appeler la transformation *skew* avec le paramètre $y - x$.

Les transformations *grain* et $densify$ nous permettent aussi de modifier les équations qui nous intéressent. Cependant, ces transformations ne nous sont pas nécessaires pour montrer la complétude de notre jeu de transformations. En effet, l'utilisation de la transformation *reshape* suffit pour nos modifications car la dimension de sortie *alpha* est définie explicitement.

Lorsque la relation d'ordonnancement est représentée sous la forme de sortie, on peut modifier arbitrairement les équations portant sur les dimensions α grâce aux transformations $shift$, *reshape* et *skew*. Les équations restantes sont des inéquations qui portent aussi sur les dimensions α. Elles sont traitées dans la section suivante.

Modification des inéquations

La structure des inéquations reste limitée et a une forme particulière afin de respecter la condition de validité globale. Les transformations *stripmine*, *linearize*, *index_set_split* et *collapse* permettent de créer ou supprimer ces inéquations.

Si on trouve un couple d'inéquations décrivant une α-dimension définie implicitement, cette dernière a été définie avec la transformation *stripmine*. Il s'agit de la seule forme possible pour ce couple d'inéquations. En effet, cette forme est la seule à assurer que chaque instance de l'instruction soit ordonnancée à une date logique unique, nécessaire à la condition de validité globale. Cette transformation peut être défaite avec *linearize*.

Toutes les autres inéquations sont arbitraires et peuvent être ajoutées grâce à la transformation *index_set_split* et enlevées avec la transformation *collapse*. Cependant, pour ajouter cette inéquation tout en respectant la condition de validité globale, il faut que l'inéquation inverse soit présente dans une autre relation basique qui compose la relation d'ordonnancement. Si cette inéquation inverse n'apparaît pas, la relation d'ordonnancement ne respecte pas la condition de validité globale car on ne peut pas s'assurer que toutes les instances de l'instruction soient ordonnancées.

Grâce aux transformations *stripmine*, *linearize*, *index_set_split* et *collapse*, chaque partie des inéquations qui composent la relation d'ordonnancement peut être modifiée arbitrairement.

Grâce à la modification des équations, comprenant la définition des dimensions β et la définition des dimensions α définies explicitement, et grâce à la modification des inéquations comprenant, la définition des dimensions α définies implicitement correspondant aux dimensions stripminées et les inéquations permettant de découper les instances des instructions, l'ensemble des transformations proposé permet de modifier arbitrairement tous les éléments des relations d'ordonnancement dans le respect de la condition de validité globale. Toute transformation valide est donc représentable par une séquence de transformations de cet ensemble : notre jeu de transformations est complet.

5.3 Travaux associés

Nous avons déjà discuté des plateformes UTF [52, 53], URUK [28] et CHiLL [27, 54] au début du chapitre. Ces plateformes exposent une interface haut niveau et sont basés sur le modèle polyédrique. Notre solution *Clay* est plus générique. Premièrement, elle relâche les contraintes d'unimodularité et d'invertibilité. Deuxièmement, elle utilise les unions de relations basiques, une représentation polyédrique plus générale. Troisièmement, elle lit et modifie uniquement les relations d'ordonnancement permettant d'assurer la validité de la transformation quels que soient les domaines d'itération. Le test de légalité peut se faire uniquement à la fin de la séquence de transformations, il n'est pas nécessaire de l'effectuer après chaque étape. Pour finir, notre solution est complète : elle permet de construire n'importe quel ordonnancement polyédrique.

Plusieurs travaux permettent au développeur d'utiliser des transformations de boucles complexes en implémentant une plateforme de transformations de boucles haut niveau. Xlanguage [55] propose au développeur d'ajouter des pragmas destinés au compilateur. Ces pragmas permettent d'effectuer des transformations de boucles syntaxiques dans un code source en C. Plusieurs transformations sont supportées ; par exemple, il est possible de dérouler et stripminer une boucle. POET [56] sauvegarde et paramétrise des transformations comme le déroulage de boucle ou le tuilage. En interne, les expressions sont stockées dans un arbre syntaxique abstrait (AST en anglais, pour Abstract Syntax Tree). Goofi [57] (*Graphical Optimization Of Fortran Implementations*) offre une interface visuelle pour transformer des boucles en code Fortran. Contrairement à notre solution, aucune de ces approches n'utilise le modèle polyédrique. De ce fait, elles ne profitent pas de son analyse précise nécessaire à la conception de séquences de transformations complexes.

5.4 Conclusion

Dans ce chapitre, nous avons défini un ensemble de directives de transformations haut niveau qui peuvent être syntaxiques ou sémantiques. Ces transformations sont accessibles à l'utilisateur avec le vocabulaire déjà utilisé par les développeurs pour les transformations les plus classiques. Même si cela est transparent à l'utilisateur, en interne, cette plateforme de transformations utilise le modèle polyédrique afin de profiter de sa puissance d'analyse et de restructuration du code.

Notre plateforme de transformations haut-niveau est une évolution d'UTF et de URUK qui permettaient déjà au développeur de construire des transformations en utilisant le vocabulaire des transformations classiques de boucles. Bastoul a commencé Clay [16] en mettant à jour la représentation polyédrique utilisée. Clay utilise l'union de relations basiques, plus générale que les fonctions. Notre contribution principale rend Clay composable et complet tout en restant expressif. Il s'agit d'un travail d'équipe avec Oleksandr Zinenko, Stéphane Huot et Cédric Bastoul [2].

En combinant nos transformations, il est possible de construire n'importe quel ordonnancement polyédrique. Pour cela, notre plateforme lit et modifie uniquement les relations d'ordonnancement des instructions en respectant la condition de validité globale. Dans cette condition, notre jeu de transformations est le premier à être complet.

Comme notre jeu de transformations est complet et comme la plupart de ces transformations modifient uniquement une partie spécifique de la relation d'ordonnancement, nous avons la certitude qu'il est possible de déduire les transformations faites par un outil externe uniquement en analysant les relations d'ordonnancement. Le chapitre suivant présente le premier algorithme permettant de construire une séquence de transformations étant donné un SCoP original et un SCoP transformé.

Chapitre 6

Résultat de l'optimiseur polyédrique

Sommaire

L'algorithme de Pluto [5] est aujourd'hui considéré comme l'état de l'art en matière d'optimiseur polyédrique. Il est aujourd'hui utilisé dans la plupart des plateformes de compilation polyédriques. Il est par exemple présent dans Graphite, la plateforme de compilation polyédrique de GCC, depuis sa version 4.8 en 2013 [1]. Il est également utilisé dans Polly, la plateforme de compilation polyédrique de LLVM [2] et, même si l'optimiseur par défaut est ISL depuis 2014, ce dernier est basé sur l'algorithme de Pluto. Cet algorithme prend en entrée un SCoP et retourne un nouveau SCoP. Les optimisations faites par l'algorithme sont encodées dans la représentation polyédrique. Les domaines d'itération et les relations d'ordonnancement sont modifiés. Généralement, les domaines d'itération encodent les transformations de tuilage et, les relations d'ordonnancement contiennent les autres transformations. De par le fonctionnement de l'algorithme de Pluto, il est difficile de comprendre la nature de ces transformations en observant le SCoP transformé car il s'agit uniquement de valeurs entières dans les relations d'ordonnancement et/ou de nouvelles (in)équations. Seule la génération du code permet de révéler la nature de ces transformations à un non expert mais le code source généré est souvent peu lisible comme dans notre exemple avec `jacobi-2d-imper` dans le chapitre 2.

L'algorithme de Pluto mais également celui de LeTSeE, donnent une optimisation parmi des centaines voire des milliers d'ordonnancements possibles [33]. Le résultat de ces optimiseurs est une seule optimisation qui est jugée la plus performante selon différents critères. Ces critères peuvent être donnés par des *fonctions objectifs* portant sur la parallélisation, la vectorisation et l'amélioration de la localité des données, mais peuvent aussi seulement porter sur le temps d'exécution constaté de l'optimisation, comme dans le cas de LeTSeE. Aucune transformation de code n'apparaît dans cette partie car l'ensemble de l'optimisation se fait sur la représentation polyédrique. De par le fonctionnement de ces algorithmes, l'optimiseur polyédrique se comporte comme une boîte noire car, même si son fonctionnement est connu, la nature de son résultat, c'est-à-dire la transformation obtenue, reste incompréhensible et non modifiable pour l'utilisateur du compilateur, à moins de retravailler le code généré.

Il est possible d'encoder l'ensemble de la transformation polyédrique d'un optimiseur dans les différentes relations d'ordonnancement du SCoP. Comme notre jeu de transformations Clay est complet, nous pouvons traduire la transformation de l'optimiseur sous la forme d'une séquence de transformations Clay. On peut donc présenter le résultat de l'optimiseur polyédrique au développeur avec des directives de transformations haut-niveau. Ce dernier pourra alors comprendre et même modifier la séquence de transformations pour obtenir l'optimisation voulue.

1. https://gcc.gnu.org/wiki/Graphite-4.8
2. http://polly.llvm.org/

Ce chapitre présente, dans un premier temps, comment il est possible d'inverser chaque transformation. Dans un second temps, nous proposons un algorithme permettant de retrouver une séquence de transformations permettant de passer d'un SCoP original à un SCoP transformé. Cet algorithme a été implémenté et testé avec succès sur les PolyBench/C par Oleksandr Zinenko. Pour chaque benchmark, l'algorithme a retrouvé une séquence de transformations équivalente à l'optimisation effectuée par Pluto (sans l'option `--tile`).

6.1 Transformations inversibles

La première étape dans la recherche d'une séquence de transformations permettant de passer d'un SCoP à un autre est de pouvoir annuler les différentes transformations. Une transformation est dite *inverse* d'une autre si elle permet de l'annuler. On cherche à établir un tableau qui associe à chaque transformation sa transformation inverse. Une transformation est l'inverse d'une autre si elle permet d'obtenir à nouveau l'ordonnancement d'origine, c'est-à-dire d'avant la transformation en l'appliquant à l'ordonnancement transformé. Certaines transformations syntaxiques peuvent être directement annulées avec la même transformation, avec les mêmes paramètres, comme la transformation *reverse*, ou avec des paramètres différents, comme la transformation $shift$ où il faut déduire la quantité *amount*. Les autres transformations syntaxiques sont annulées par une transformation différente, par exemple, $fuse$ permet d'annuler *distribute*. Les transformations sémantiques, par nature, ne modifient ni la structure ni les valeurs des relations d'ordonnancement.

Transformation + Transformation inverse
reorder
$shift$
skew
reverse
reshape
interchange

Transformation	**Transformation inverse**
$fuse\_next$	*distribute*
grain	*densify*
stripmine	*linearize*
index_set_split	*collapse*

Pour annuler la transformation $reorder(\vec{\rho}, \vec{v})$, il suffit de rappeler la même transformation, *reorder*, avec la même β-boucle $\vec{\rho}$. Seul le vecteur $\vec{v}$ diffère. On nomme le vecteur d'entiers de la transformation inverse $\vec{w}$. Ce nouveau vecteur d'entiers $\vec{w}$ peut être construit à partir de $\vec{v}$. $\exists \vec{w} : \vec{w}_{i-1} = j : \vec{v}_{j-1} = i$ avec $1 \leq i, j \leq \dim \vec{v}$ et $\dim \vec{v} = \dim \vec{w}$. On peut construire le vecteur $\vec{w}$ de la façon suivante : on lit le vecteur $\vec{v}$ et pour chaque valeur v à l'indice i, la valeur de $\vec{w}_v$ est i. Les indices commencent à 0 ici (et uniquement ici).

La transformation $fuse\_next(\vec{\rho})$ peut être annulée avec $distribute(\vec{\rho}, n)$ et inversement. Dans le premier cas, il faut déduire n à partir des instructions avant la fusion. $n = \max_{\mathcal{T} \in \mathcal{T}_{\vec{\rho},*}} \left(\vec{\beta}_{\mathcal{T}, \dim \vec{\rho}+1} \right)$. n est donc égal au nombre d'instructions présentes dans la boucle à fusionner (avant la fusion).

Dans le cas inverse, $fuse\_next(\vec{\rho})$ permet d'annuler $distribute(\vec{\rho}, n)$ quel que soit n car $fuse\_next(\vec{\rho})$ travaille sur les $\vec{\rho}$ et non sur des instructions.

$reverse(\vec{\rho})$ est l'inverse d'elle-même avec la même β-boucle $\vec{\rho}$.

$shift(\vec{\rho}, i, amount)$ peut être annulée par $shift$ avec la même β-boucle $\vec{\rho}$ et la même dimension α nommée i mais avec une quantité *amount* différente. *amount* est de la forme $\vec{v} \times \vec{p} + N$. Il suffit d'annuler le déplacement par *amount* par une nouvelle quantité nommée $amount'$ telle que $amount' = \vec{w} \times \vec{p} + -N$: $\vec{w}_i = -\vec{v}_i$ avec $1 \leq i, j \leq \dim \vec{v}$ et $\dim \vec{v} = \dim \vec{w}$. $amount'$ est construit à partir de *amount* en prenant l'opposé de chaque coefficient des paramètres et l'opposé de la constante.

De façon similaire à *reverse* et à $shift$, on peut annuler $skew(\vec{\rho}, i, factor)$ par $skew(\vec{\rho}, i, -factor)$.

$reshape(\vec{\rho}, i, factor)$ permet de modifier arbitrairement les coefficients des dimensions de sortie sous certaines conditions. Comme on suppose que l'ordonnancement d'origine était valide, on peut annuler *reshape* avec $reshape(\vec{\rho}, i, factor')$. $factor'$ correspond au coefficient de $\vec{\iota}_{\mathcal{T},i}$ avant le premier *reshape*. Aucune condition n'a besoin d'être vérifiée.

Comme *distribute* et $fuse\_next$, la transformation $grain(\vec{\rho}, i, factor)$ est annulée par $densify(\vec{\rho}, i)$ quel que soit la valeur de *factor*). Si deux transformations *grain* avaient été appliquées, l'appel de $densify$ annule les deux. Pour annuler uniquement le dernier, il faut appeler $densify(\vec{\rho}, i)$ et $grain(\vec{\rho}, i, old\_factor)$ de façon à rétablir l'espacement entre les instances des instructions présent avant le deuxième *grain*. Il faut donc extraire old_factor de la relation le facteur d'espacement d'avant la dernière transformation *grain*. Ce n'est pas problématique car cela est fait pour le fonctionnement de la transformation $densify$.

L'inversion de $densify(\vec{\rho})$ par *grain* se fait après la déduction du *factor*. La déduction de ce facteur est déjà faite pour le fonctionnement de la transformation $densify$ ou pour annuler proprement *grain* si plusieurs transformations *grain* avaient été effectuées.

De la même façon que *reverse*, $interchange(\vec{\rho}, i)$ est l'inverse d'elle-même avec la même β-boucle $\vec{\rho}$ et la même dimension α nommée i.

$stripmine(\vec{\rho}, size)$ et $linearize(\vec{\rho})$ sont complémentaires. *linearize* annule le *stripmine* quel que soit la valeur de *size*. Comme *stripmine* ne permet pas de modifier les valeurs des dimensions déjà stripminées, il n'est pas nécessaire d'annuler un *stripmine* par un *linearize* puis un *stripmine* comme cela était le cas pour *grain* et $densify$.

stripmine permet de rétablir une dimension linéarisée. Il faut extraire la taille du strip (tuile 1D) *size*. Cette opération est déjà faite par *linearize* lors de la vérification des conditions.

La transformation $index\_set\_split(\vec{\rho}, constraint)$ est annulée par $collapse(\vec{\rho})$. Comme le couple *grain* et $densify$, si plusieurs transformations $index\_set\_split$ ont été appliquées, il faut appeler *collapse* et réappliquer les anciens $index\_set\_split$ que l'on souhaite garder. Il suffit de détecter les contraintes présentes avant le dernier $index\_set\_split$. Cela est déjà fait par la transformation *collapse* qui identifie la contrainte et la contrainte opposée correspondante.

Comme $parallelize(\vec{\rho})$ est une transformation sémantique, elle ne modifie ni la structure, ni les valeurs de la relation d'ordonnancement. Cette information sémantique peut être ajoutée ou enlevée à tout moment. Cela s'applique à l'ensemble des transformations sémantiques.

Il est possible de généraliser les transformations inverses aux séquences de transformations. Les différences entre le SCoP original et le SCoP optimisé doivent porter uniquement sur les relations d'ordonnancement et ces relations d'ordonnancement doivent respecter la condition de validité globale. Comme ce n'est pas le cas des relations d'ordonnancement issues de l'algorithme de Pluto, on propose de " corriger " les relations d'ordonnancement en les *alignant*.

6.2 Alignement des relations d'ordonnancement

La première étape de notre technique pour trouver une séquence de transformations permettant de passer d'un SCoP original à un SCoP transformé est d'*aligner* les différentes relations d'ordonnancement du premier SCoP avec le deuxième et d'obtenir les mêmes β-vecteurs. On cherche donc à relier chaque relation d'ordonnancement du SCoP original à son homologue dans le SCoP transformé. Avec cette étape, les deux SCoPs sont dit alignés, c'est-à-dire que la struture des relations d'ordonnancement (nombre de dimensions, nombre de relations basiques) et les valeurs des β-vecteurs sont les mêmes.

Pour effectuer cette opération, on a besoin des transformations *reorder*, $fuse\_next$, *distribute*, *stripmine*, *linearize*, $index\_set\_split$ et *collapse* afin de changer arbitrairement la structure des dimensions β et les valeurs des β-vecteurs. Les transformations *reorder*, $fuse\_next$ et *distribute* permettent de changer les valeurs des β-vecteurs. Les transformations *stripmine* et *linearize* permettent de modifier le nombre de dimensions de sortie α et β. Enfin, les transformations $index\_set\_split$ et *collapse* permettent de changer le nombre de relations basiques de la relation d'ordonnancement. Appliquer le pseudo-*stripmine* sur la " mauvaise " dimension n'est pas un problème car les transformations Clay nous autorisent la modification arbitraire des relations d'ordonnancement. En revanche la séquence de transformations retrouvée sera plus longue et probablement plus difficilement compréhensible.

Nous proposons l'algorithme suivant pour résoudre le problème d'alignement des relations d'ordonnancement. Son fonctionnement est simple, il faut d'abord retrouver la structure des dimensions de sortie avec *stripmine*, *linearize*, *index_set_split* et *collapse* puis retrouver les valeurs des β-vecteurs avec *reorder*, *fuse_next* et *distribute*.

Retrouver la structure des dimensions de sortie. Dans certains cas, on pourra déduire les paramètres des transformations *stripmine* et *index_set_split*. Mais, ce n'est pas toujours possible. On remplace alors les transformations *stripmine* et *index_set_split* par, respectivement, un pseudo-*stripmine* qui ajoute une dimension constante et, par un pseudo-*index_set_split* qui ajoute une condition fausse. Ces deux pseudo-transformations nous permettent de trouver la bonne structure des relations d'ordonnancement, c'est-à-dire, avec le bon nombre de dimensions et le bon nombre de relations basiques par relation. Les valeurs des dimensions stripminées et celles des inéquations permettant le découpage des instances des instructions seront trouvées à l'étape suivante. L'algorithme d'alignement ne s'occupe pas de ces valeurs.

Recherche des valeurs des β-vecteurs. Lorsque la bonne structure des dimensions de sortie α et β est obtenue, on peut utiliser *reorder*, *fuse_next*, *distribute* pour obtenir les mêmes valeurs des β-vecteurs. Un algorithme naïf serait de distribuer toutes les instructions afin qu'elles ne partagent aucune boucle commune. Dans cette configuration, seule la première valeur des β-vecteurs compte, toutes les autres sont à 0. Ensuite, avec la transformation *reorder*, on peut réordonner toutes les instructions pour quelles soient dans le même ordre que celles du SCoP transformé. Pour finir, on utilise *fuse_next* afin de refusionner les boucles si besoin. Cet algorithme simple peut générer plus de transformations que nécessaire. Par exemple, c'est le cas si les instructions sont déjà dans le bon ordre mais ne partagent pas les mêmes boucles communes. Dans ce cas, on propose l'algorithme Match_Betas qui vérifie à différentes profondeurs si le *distribute* est nécessaire.

Algorithm 1: Match_Betas

```
1  let ρ⃗ be the current β-prefix ;
2  let d = dim ρ⃗ + 1 ;
3  let n = max_{T∈θ} β⃗_d where β⃗_d is the β-vector of T ;
4  let n' = max_{T'∈θ'} β⃗'_d where β⃗'_d is the β-vector of T' ;
5  while ∃β⃗, β⃗' : β⃗_d ≠ β⃗'_d, β⃗_d → min do
       /* split away until depth d                                   */
6      for i = dim β⃗ downto d do
7          Reorder(β⃗_{1...(i−1)}, put β⃗_d last) ;
8          Distribute(β⃗, i) ;
9      if β⃗_d ≤ n then
10         Reorder(β⃗_{1...(d−1)}, put β⃗_d right after β⃗'_d) ;
11         FuseNext(β⃗_{1...d}) ;
           /* split away betas that do not belong to the same transformed prefix */
12         foreach β⃗, β⃗' : β⃗_d ≠ β⃗'_d do
13             Reorder(β⃗_{1...(i−1)}, put β⃗_d last) ;
14             Distribute(β⃗, i) ;
15     else
16         Reorder(β⃗_{1...(d−1)}, put β⃗_d at β⃗'_d) ;
   /* n = n' at this point; recurse to sub-prefixes                 */
17 for i = 0 to n do
18     recurse with ρ⃗ ← (ρ⃗_1, ρ⃗_2, ..., ρ⃗_{dim ρ⃗}, i) ;
```

6.3 Recherche des coefficients des dimensions α

Après l'étape précédente permettant d'aligner les relations d'ordonnancement respectives, la structure des relations d'ordonnancement et les valeurs des β-vecteurs sont retrouvées. Cependant, les coefficients des dimensions d'entrée, des dimensions de sortie α et des paramètres ainsi que les constantes ne sont pas encore corrects.

On remarque que l'ajout d'une dimension définie explicitement ne peut être fait que par la transformation *skew*. Si on utilise l'élimination de Gauss-Jordan afin d'obtenir une forme triangulaire, il est possible de modifier les coefficients individuellement avec la transformation *reshape*. Après chaque transformation, la substitution de dimensions définies implicitement peut avoir introduit d'autres transformations. Il faut le vérifier et si besoin compléter les transformations *stripmine*, *index_set_split* et *grain* déjà retrouvées.

On propose l'algorithme White_Boxing pour retrouver les coefficients des dimensions α et compléter la séquence de transformations permettant de passer du SCoP original au SCoP transformé. Comme les transformations apparentées à *skew* (*skew*, *reshape*, *reverse*, *interchange*) peuvent nuire aux pré-conditions des transformations *linearize* et *collapse*, les transformations inverses *stripmine* et *index_set_split* sont détectées le plus tôt possibles. En utilisant l'élimination de Gauss-Jordan, on préfère l'utilisation de la transformation *skew* plutôt que *reshape* même si on obtient une séquence de transformations plus grande. Ce choix a été fait car *skew* est une transformation classique de code présente dans l'ensemble des plateformes de transformations contrairement à la transformation *reshape*.

Algorithm 2: White_Boxing

1 Transform all relations in both the original and optimized scheduling unions to the *output* form.;
2 **repeat**
3 *restart iteration after each step with* ↻;
4 Match_Betas ($\vec{\rho} \leftarrow ()$);
5 **foreach** *statement* S **do**
6 **foreach** *implicitly defined dimension* d **do**
7 Complementary(StripMine) ↻;
8 Complementary(IndexSetSplit) ↻;
9 **foreach** *dimension* d **do**
10 Complementary(Grain) ↻;
11 **foreach** $\vec{\alpha}_i$, $i \rightarrow \max$, *explicitly defined by* $\{\alpha_i = \vec{v} \cdot \vec{\imath}^T + \vec{w} \cdot \vec{p}^T + C\}$ *in* $\mathcal{T}$ ***or*** *in* $\mathcal{T}'$ **do**
12 **foreach** $j \leftarrow 1..(i-1)$ *:* $v_j \neq 0$ **do**
13 **let** x_j be the v_j value in the explicit definition of α_j ;
14 *use inverse transformation if* $\in \theta'_S$;
15 Grain($\beta_{\mathcal{T},1..i}$, $\mathrm{lcm}(x_j, v_j)/v_j$) ;
16 Skew($\beta_{\mathcal{T},1..i}$, j, $-\mathrm{lcm}(x_j, v_j)/x_j$) ;
17 Densify($\beta_{\mathcal{T},1..i}$) ↻;
18 **if** $\vec{v}_i < 0$ **then**
19 Reverse($\beta_{\mathcal{T},1..i}$) ↻;
20 **foreach** $\vec{\alpha}_i$, $\vec{\alpha}'_i$ *both explicitly defined by* $\{\alpha_i = \vec{v} \cdot \vec{\imath}^T + \vec{w} \cdot \vec{p}^T + C\}$ *in* $\mathcal{T}$, $\mathcal{T}'$ **do**
21 **foreach** $\vec{v}_j \neq \vec{v}'_j$ **do**
22 Reshape($\beta_{\mathcal{T},1..i}$, $\vec{v}'_j - \vec{v}_j$) ↻;
23 repeat step 22 for $\vec{w}$ and C with Shift ;
24 **foreach** $\vec{\alpha}_i$, $\vec{\alpha}'_i$ *both implicitly defined by* $\{\vec{u} \cdot \vec{\alpha}^T + \vec{v} \cdot \vec{\imath}^T + \vec{w} \cdot \vec{p}^T + C \geq 0\}$ *in* $\mathcal{T}, \mathcal{T}'$ **do**
25 **foreach** $\vec{u}_j \neq \vec{u}'_j$ **do**
26 Skew($\beta_{\mathcal{T},1..i}$, j, $\vec{u}'_j - \vec{u}_j$) ↻;
27 repeat step 26 for $\vec{v}$ with Reshape ;
28 repeat step 26 for $\vec{w}$ and C with Shift ;
29 **until** *no transformation happened in the loop*;

Cet algorithme a été implémenté par Oleksandr Zinenko dans *Chlore*[3] et a été exécuté sur les benchmarks des PolyBench/C 4.1 optimisé par Pluto 0.11.4.

La figure 6.1 correspond à la séquence de transformations retrouvée par Chlore pour le benchmark `jacobi-2d-imper` (présenté dans le chapitre 2) des PolyBench/C 4.1 optimisé par Pluto 0.11.4. Il s'agit du résultat de Chlore. Même s'il faut un peu de temps pour analyser et comprendre exactement cette séquence de transformations, l'effort demandé est moins important qu'avec le code source généré par CLooG dont la version complète est en annexe B dans la figure B.1. Ici, le résultat est présenté sous sa forme brute, des commentaires ou même des animations pourraient être ajoutés afin de faciliter la compréhension des différentes étapes. De plus, comme cette séquence n'est pas unique, il est probablement possible d'en trouver une plus simple pour le développeur.

```
fuse([0,0]);
fuse([0,0,0]);
reshape([0,0,0,1], 2, 3, 1);
grain([0,0,0,1], 1, 2);
reshape([0,0,0,1], 1, 3, -1);
interchange([0,0,0,1], 2, 3, 0);
skew([0,0,0,1], 2, 3, -1);
reverse([0,0,0,1], 2);
interchange([0,0,0,1], 1, 3, 0);
skew([0,0,0,1], 1, 3, -1);
interchange([0,0,0,1], 1, 2, 0);
skew([0,0,0,1], 1, 2, -1);
densify([0,0,0,1], 1);
shift([0,0,0,1], 3, [0,0], 1);
shift([0,0,0,1], 2, [0,0], 1);
reshape([0,0,0,0], 2, 3, 1);
grain([0,0,0,0], 1, 2);
reshape([0,0,0,0], 1, 3, -1);
interchange([0,0,0,0], 2, 3, 0);
skew([0,0,0,0], 2, 3, -1);
reverse([0,0,0,0], 2);
interchange([0,0,0,0], 1, 3, 0);
skew([0,0,0,0], 1, 3, -1);
interchange([0,0,0,0], 1, 2, 0);
skew([0,0,0,0], 1, 2, -1);
densify([0,0,0,0], 1);
```

Figure 6.1 – Séquence de transformations retrouvée par Chlore pour le benchmark `jacobi-2d-imper` des PolyBench/C 4.1, optimisé par Pluto 0.11.4 (sans l'option `--tile`)

La figure 6.2 donne le nombre d'instructions et le nombre de transformations pour passer d'un ordonnancement original à celui optimisé par Pluto sur les PolyBench/C. Le nombre de statements va de 1 à 42 pour une moyenne de 6. Le nombre de transformations va de 1 à 143 pour une moyenne de 24.

La figure 6.3 donne les temps d'exécution en seconde de Chlore pour retrouver les optimisations faites par Pluto sur les PolyBench/C. Pour chaque benchmark, le temps d'exécution donné correspond à la médiane de 12 exécutions sur un i7-4850HQ 2.30 GHz. Chlore a été compilé avec le compilateur Clang 3.8 avec l'option `-O3`. Le code source de Chlore n'a bénéficié d'aucune optimisation manuelle particulière. Le temps d'exécution va de $0,01$ à $0,416$ seconde. La médiane de toutes les médianes est de $0,032$ seconde et la moyenne est à $0,055$ seconde.

3. https://periscop.github.io/chlore/

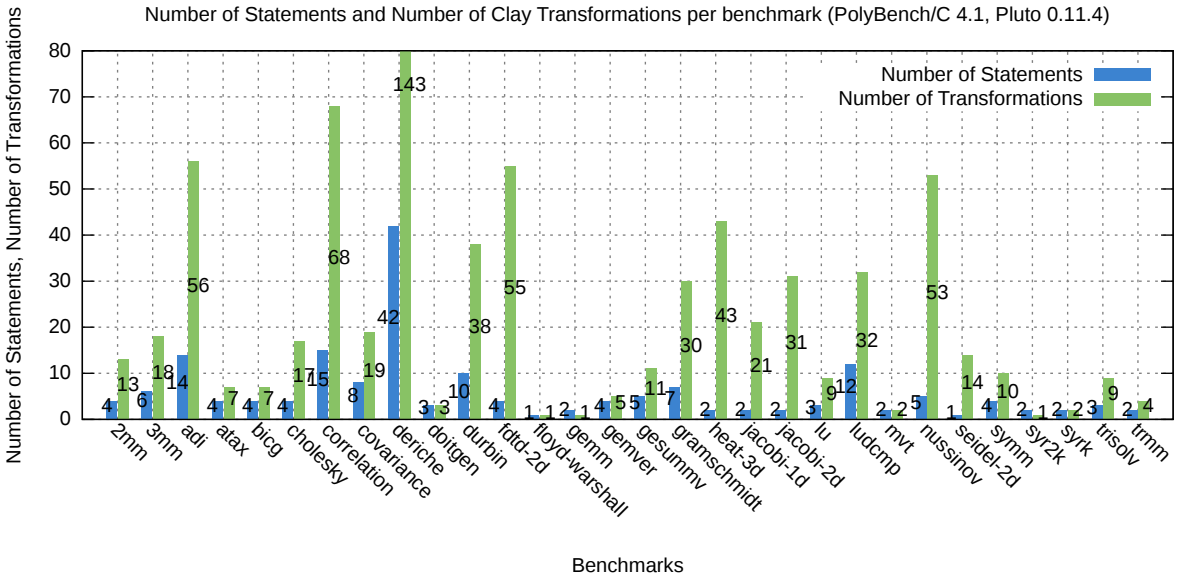


Figure 6.2 – Nombre d'instructions et de transformations Clay par benchmark (PolyBench/C 4.1, Pluto 0.11.4)

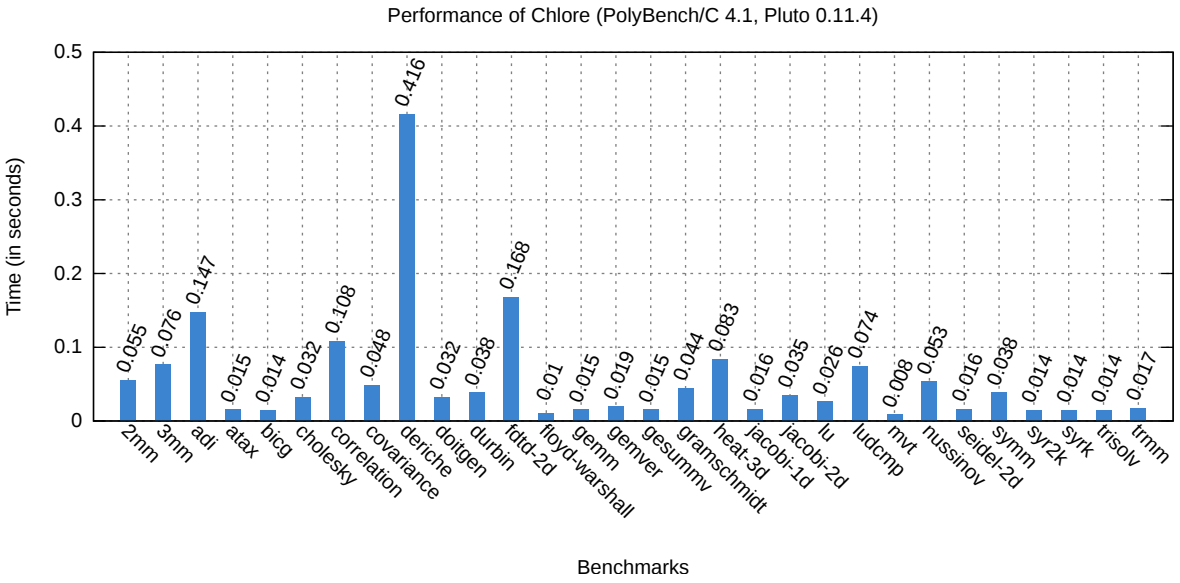


Figure 6.3 – Temps d'exécution (en seconde) de Chlore par benchmark (PolyBench/C 4.1, Pluto 0.11.4, Intel i7-4850HQ 2.30 GHz, Clang 3.8 avec l'option `-O3`)

6.4 Interaction utilisateur-compilateur

Notre approche autorise une nouvelle forme d'interaction entre le compilateur et le développeur : ce dernier peut comprendre l'optimisation polyédrique et la modifier sans utiliser la représentation polyédrique. Habituellement, l'utilisation d'un compilateur polyédrique source à source, comme Pluto, est la suivante. L'utilisateur donne au compilateur son code source. Le compilateur appelle alors un parseur qui traduit les SCoP du code source dans la représentation polyédrique utilisée. L'optimiseur peut ensuite effectuer son travail, en interne, sur la représentation polyédrique. Une fois l'optimisation effectuée, le générateur de code est appelé et le code source résultat est renvoyé à l'utilisateur. L'utilisateur n'a pas connaissance du modèle polyédrique mais n'a pas la possibilité d'interagir avec l'optimiseur.

En retrouvant une séquence de transformations correspondant à l'ordonnancement fait par l'optimiseur polyédrique, on peut proposer à l'utilisateur d'interagir avec la plateforme de compilation polyédrique sans connaissance du dit modèle. Plusieurs scénarios sont possibles.

Formes du résultat de l'optimiseur polyédrique. Si on donne un code source au compilateur polyédrique, il peut soit retourner le code optimisé, soit le code source original annoté de la séquence de transformations correspondant à l'optimisation (cette séquence est retrouvée automatiquement grâce à l'algorithme 2).

Vérification de l'optimisation. L'utilisateur peut vérifier cette séquence de transformations et la redonner directement au compilateur polyédrique pour obtenir le code source optimisé.

Modification de l'optimisation. L'utilisateur a aussi la possibilité de modifier cette séquence de transformations afin de raffiner ou modifier l'ordonnancement proposé par l'optimiseur polyédrique. Lorsque l'utilisateur a fini de modifier les transformations, il demande au compilateur de générer le code source correspondant à sa transformation. Dans ce scénario, une " vraie " communication, à double sens, est établie entre le développeur et la plateforme de compilation polyédrique. Le compilateur peut vérifier si la transformation de l'utilisateur est légale.

Création de l'optimisation. L'utilisateur n'est pas obligé de modifier la séquence de transformations proposée par le compilateur polyédrique. Il peut aussi directement créer la sienne en s'inspirant, ou pas, de la séquence de transformations correspondant à celle de l'optimiseur polyédrique. Là aussi, le compilateur peut vérifier si la transformation de l'utilisateur est légale.

L'utilisateur peut donc directement interagir avec le moteur polyédrique qui s'occupe d'appliquer les transformations demandées. Comme la plateforme de transformations fait les modifications nécessaires dans la représentation polyédrique, l'utilisateur n'est jamais confronté à la représentation polyédrique.

En fournissant une interface en mode texte et une API (*Application Programming Interface*, interface de programmation) dans l'outil Clay, on donne accès, simplement, à la plateforme de compilation polyédrique et à sa puissance afin de développer des nouveaux outils pour la compilation semi-automatique assistée par le compilateur. Cela inclut, par exemple, l'optimisation à travers une représentation visuelle du programme [58] ou l'élaboration d'un langage dédié (DSL *Domain Specific language*) dans le but de diriger l'optimiseur vers l'optimisation voulue.

6.5 Travaux associés

Le but de notre approche est d'offrir au développeur un moyen de communiquer avec l'optimiseur de boucles du compilateur. À notre connaissance, aucun travail n'aborde cette problématique. En revanche, plusieurs travaux proposent des approches complémentaires pour interagir avec le compilateur afin d'obtenir de meilleures optimisations. L'infrastructure Paralax [59] autorise le développeur à compléter l'analyse du compilateur grâce à des annotations et utilise le retour du compilateur pour les suggérer. Cette infrastructure permet de paralléliser automatiquement. Elle cible les programmes qui utilisent intensivement l'arithmétique de pointeurs. Larsen et al. [60] proposent un compilateur de production qui explique pourquoi la parallélisation automatique d'une boucle a échoué. Cela peut être dû à plusieurs raisons ; par exemple, les indices et/ou les pas de boucles sont trop compliqués ou alors, des alias de variables ou de

tableaux sont possibles. Plutôt que de demander au développeur d'ajouter des annotations supplémentaires, le programmeur doit modifier ou réécrire le code source pour que ce dernier soit compréhensible par l'optimiseur. Dans ce cas favorable, l'optimiseur pourra alors remplir pleinement son rôle. Avec un fonctionnement similaire, Jensen et al. [61] propose un plugin pour les environnements de développement intégrés (*Integrated Development Environment* en anglais, abrégé IDE) permettant de récupérer le retour des différentes passes des optimisations du compilateur. Un outil intéractif basé sur Polly développé par Göhringer and Tepelmann [62] donne un retour sur l'intensité arithmétique des boucles. Cet outil peut les paralléliser ou les optimiser automatiquement. Prospector [63] permet d'effectuer un profilage de code dynamique afin d'identifer les dépendances de données au niveau des boucles dans le but de les paralléliser. Il y a deux catégories de boucles, celles pouvant être parallélisées facilement avec OpenMP, et celles qui demandent des modifications de code plus avancées. Contrairement à ces outils, notre approche ne demande pas de modification manuelle du code. Elle se base sur une interface haut niveau permettant d'exprimer les différentes transformations et utilise les plateformes de compilation polyédriques pour la génération du code optimisé.

6.6 Conclusion

Grâce à la plateforme de transformations de boucles haut-niveau Clay, l'utilisateur peut développer ses propres séquences de transformations afin d'optimiser son code source et cela, sans le modifier manuellement. Le point fort de Clay est sa capacité à exprimer n'importe quel ordonnancement polyédrique. Cette propriété est assurée en interne par l'utilisation exclusive et unique des relations d'ordonnancement. Cela permet d'assurer la condition de validité globale. Le développeur peut utiliser la même interface pour examiner, comprendre, raffiner et modifier le résultat de l'optimiseur polyédrique. Ces opérations étaient jusqu'à présent impossibles de par la nature du fonctionnement de l'optimiseur. Même si son mode opératoire est connu, en résolvant un système d'inéquations linéaires, l'optimisation obtenue n'est pas accessible aux non experts en modèle polyédrique. Avec l'algorithme élaboré et implémenté dans Chlore par Oleksandr Zinenko, nous pouvons présenter l'optimisation du compilateur polyédrique à n'importe quel développeur, sous la forme de séquence de transformations Clay. Il s'agit d'un nouveau moyen d'intéragir avec la plateforme de compilation polyédrique. Clay et Chlore permettent donc d'ouvrir, virtuellement, la boîte noire du compilateur polyédrique.

Chapitre 7

Conclusion & Perspectives

Conclusion

Les architectures offrent une puissance de calcul crête de plus en plus élevée grâce, principalement, à un nombre de cœurs de plus en plus grand. En contrepartie, programmer ces architectures dans le but d'atteindre une fraction correcte de cette puissance, demande aux programmeurs un haut niveau technique. La prise en compte de la localité des données, du parallélisme de données et du parallélisme de tâches est un bon début pour programmer ces multi-cœurs présents dans l'ensemble des machines grand public. Le rôle des outils permettant l'adéquation logiciel - matériel tels que les langages, les compilateurs ou les bibliothèques est de répondre à ces questions. Cependant, la plupart des développeurs ignorent les limites de ces outils ainsi que le fonctionnement du matériel et il est donc difficile d'obtenir des performances, mêmes lorsque ces dernières sont nécessaires.

Ces dernières années, les compilateurs de production ont fait leur preuve pour les optimisations sur un seul cœur. Ils offrent un excellent compromis entre productivité et performances intra-cœur. Des phases d'optimisation et de parallélisation automatiques basées sur le modèle polyédrique sont présentes dans la plupart de ces compilateurs. Cependant, les fonctionnalités apportées par les plateformes polyédriques sont notées comme expérimentales et ne sont généralement pas encore utilisées par défaut. Les optimisations proposées par ces compilateurs polyédriques automatiques sont soit générales, soit spécialisées. De par le fonctionnement des optimiseurs polyédriques comme Pluto et LeTSeE, les échanges avec l'utilisateur sont limités et sont à sens unique. L'algorithme de Pluto est très largement utilisé par les différentes plateformes de compilation polyédrique. Comme il intègre des heuristiques ciblant les machines les plus courantes, alors les optimisations obtenues sont génériques et ne sont donc pas forcément parfaitement adaptées à l'architecture ciblée par l'utilisateur. À l'inverse, des optimiseurs itératifs comme LeTSeE génèrent des optimisations spécialisées pour une machine ciblée. Ces optimisations ne sont donc pas toujours efficaces sur d'autres architectures ou dans d'autres environnements dynamiques.

Pour résoudre les problèmes de ce type, nous avons proposé deux solutions permettant d'adapter les optimisations proposées par les compilateurs polyédriques. Nos deux solutions utilisent le modèle polyédrique présenté dans le chapitre 2.

La première, entièrement automatique, utilise les versions interchangeables permettant aux programmes à base de boucles imbriquées de s'adapter dynamiquement à leur environnement avec un faible surcoût.

La deuxième, semi-automatique, permet à n'importe quel développeur d'interagir avec l'optimiseur polyédrique pour vérifier, modifier ou raffiner son optimisation ou pour écrire la sienne.

Versions Interchangeables

Dans le chapitre 3 nous avons présenté les limites des optimiseurs statiques. Pour répondre à ces limites, nous avons défini les versions interchangeables. L'utilisation des versions interchangeables autorise la création d'un programme multi-versions qui a la capacité de changer de version du code original durant son exécution afin d'exécuter celle qui est la plus performantes pour le contexte d'exécution courant. Notre technique hybride associe optimisations statiques et dynamiques. Durant la phase statique, on se base sur le modèle polyédrique pour générer différentes versions interchangeables. Nous les évaluons et

les sélectionnons afin de construire le programme multi-versions final. Durant la phase dynamique, le programme trouve la version la plus performante pour le contexte d'exécution courant. Grâce à l'échantillonnage des versions disponibles, comme l'éxécution des différentes versions durant l'échantillonnage se fait sur les données du problème à résoudre, le surcoût à l'exécution de notre technique est faible.

L'utilisation des versions interchangeables est particulièrement adaptée lorsqu'il faut prendre en compte de nombreux paramètres dont les valeurs sont inconnues. C'est le cas, par exemple, lorsque le programme doit être exécuté sur différentes architectures allant du téléphone portable au super-calculateur, parfois seul sur le système, parfois sur un système déjà largement occupé par d'autres processus. Avec les machines de plus en plus hétérogènes et avec la démocratisation du *cloud computing*, les techniques de ce type devraient être de plus en plus nécessaires.

Plateforme de transformations haut-niveau

Nous avons proposé le premier jeu de transformations haut-niveau basé sur le modèle polyédrique complet. En effet, Clay permet de spécifier n'importe quel ordonnancement polyédrique.

Clay peut être utilisé directement par le développeur qui a la possibilité d'écrire des séquences de transformations correspondant aux optimisations désirées. Clay est également conçu pour être utilisé par d'autres outils pour qu'ils communiquent avec la plateforme de compilation polyédrique.

Polyhedral White Boxing

Nous avons proposé le premier algorithme qui traduit le résultat de l'optimiseur polyédrique en une séquence de transformations Clay. Comme ces transformations sont principalement basées sur le vocabulaire classiquement utilisé pour les transformations de boucles, nous offrons à n'importe quel développeur la possibilité d'interagir avec l'optimiseur polyédrique.

L'utilisation des outils de compilation permet aux développeurs d'optimiser leur code sans sacrifier leur productivité. En présentant le résultat de l'optimiseur polyédrique sous la forme d'une séquence de transformations Clay, nous ouvrons, virtuellement, la boîte noire du compilateur polyédrique et nous proposons ainsi une approche semi-automatique qui autorise le développeur à adapter l'optimisation du compilateur polyédrique.

Perspectives

Ces deux approches contribuent à l'adaptation des optimisations de programme. L'approche automatique adapte les optimisations statiques à des contextes dynamiques alors que l'approche semi-automatique offre à l'utilisateur la possibilité d'adapter l'optimisation polyédrique à ses besoins.

Face aux nouvelles architectures hybrides qui semblent se diffuser dans l'ensemble des machines grand public, les optimisations obtenues de manière entièrement automatiques sont limitées face à celles des experts humains. Les outils semi-automatiques sont utiles à l'ensemble des programmeurs. Ils permettent de simplifier le travail d'optimisation qui devient alors accessible aux développeurs qui débutent en optimisation de programmes, et ils augmentent la productivité des experts. Cependant, de nouvelles avancées de nos solutions seront utiles pour les rendre plus performantes :

Introspection des paramètres d'optimisation. Nos versions interchangeables ciblent actuellement les noyaux de calculs dont le temps d'exécution est d'au moins de l'ordre de la seconde. Comme dans les travaux de Pradelle et al. [40], des tests pourraient être effectués durant la phase statique pour déterminer quelle est la version à exécuter en fonction de paramètres dynamiques. Par exemple, pour les très petites tailles, la version séquentielle naïve est souvent la meilleure.

Continuité de l'échantillonnage. Nos versions interchangeables ciblent actuellement les noyaux de calculs composés de plusieurs centaines d'itérations. Si le noyau de calcul est court, l'échantillonnage peut résoudre entièrement le calcul et on ne profite pas des avantages offerts par la meilleure version disponible car cette dernière n'est pas exécutée suffisamment longtemps. Dans le cas où le noyau de calcul est appelé plusieurs fois, le résultat de l'échantillonnage pourrait servir directement au prochain appel. Cette solution permettrait de réduire encore plus le surcoût de notre technique.

Génération agressive des versions. Durant nos expérimentations sur les versions interchangeables, notre difficulté principale venait de notre capacité à générer des versions dont le comportement est significativement différent selon les contextes d'exécution. Le nouvel algorithme Pluto+ permet d'explorer de nouvelles optimisations. Des optimiseurs itératifs comme LeTSeE ciblant différents critères de performance, pourraient être intéressants.

Sémantique des transformations. Notre algorithme traduisant le résultat de l'optimiseur polyédrique en une séquence de transformations Clay donne une solution. Il est évident que d'autres séquences existent. Comme ces transformations sont destinées à l'utilisateur, il est important de trouver celle qui lui est la plus compréhensible. Des travaux futurs porteront sur la définition de métriques permettant de savoir si la séquence de transformations est plus ou moins adaptée à un humain. À partir de cette métrique, on pourra alors raffiner la séquence de transformations afin de l'améliorer.

Interface pour les optimisations. Comme nous pensons que le développeur doit pouvoir rester au centre des optimisations, il est nécessaire de proposer des interfaces haut-niveau qui facilitent sa compréhension des optimisations et qui simplifient les modifications qu'il veut apporter. Zinenko propose déjà le prototype d'une interface graphique [58] pour traiter ces problématiques.

Annexe A

Programmation en langage C

Sommaire

Le C est un langage de programmation inventé en 1972 par Dennis Ritchie et Ken Thompson. Ce langage a été normalisé en 1989. On parle du C89 ou ANSI C. Différentes évolutions ont vu le jour et ont été normalisées en 1999 (C99) et 2011 (C11).

Les forces du langage C sont principalement ses règles simples et sa proximité avec la machine. Grâce à ses règles simples, il est facile de créer un compilateur C. Pour cette première caractéristique, le langage C est l'un des plus portables notamment sur les architectures embarquées sans système d'exploitation. Grâce à sa proximité avec la machine, il est possible d'écrire du code bas niveau très performant. Pour cette seconde caractéristique, le langage C est l'un des plus rapides et son utilisation est toute indiquée lorsque la performance importe, notamment sur les architectures embarquées.

Le code suivant correspond au traditionnel programme « Hello world » permettant d'afficher sur la sortie standard (l'écran par défaut) le texte `Hello world!` avec un saut de ligne :

```c
#include <stdio.h>

int main()
{
    printf("Hello world!\n"); // Hello world!

    return 0;
}
```

Le style et la syntaxe de C ont été repris dans de nombreux langages tels que C++, Java, C# ou encore PHP et JavaScript pour ne citer qu'eux. Selon l'index TIOBE de juin 2016, ces 6 langages ont une popularité cumulée de presque 50%. Même si cette métrique peut être controversée, il est indéniable que ces langages de programmation, influencés par le C, sont les plus utilisés. De nombreux développements, disponibles sous forme de bibliothèque, sont écrit en C. Afin de réutiliser le code existant, de nombreux langages de programmation sont plus ou moins compatibles avec le C. Par exemple, le code compilé en C peut être utilisé en C++, Java, C# et PHP. On peut aussi compiler/transcompiler le code C en code JavaScript.

Le C est un langage impératif : on déclare des variables et on les modifie comme dans cet exemple. On déclare la variable `pi` de type `double` à la ligne 2. Cette variable est incrémentée 1000×1000 fois à la ligne 5 grâce à la boucle de la ligne 3. On multiplie la variable `pi` par 4 puis on affiche la valeur de π.

```c
// Calcul de π (formule de Leibniz, James Gregory et Madhava de Sangamagrama)
double pi = 0.0;
for (size_t i = 0; i < 1000 * 1000; ++i)
{
    pi += pow(-1, (double)i) / (double)((2 * i) + 1);
}
pi *= 4;

printf("π = %f\n", pi); // π = 3.141592
```

La plupart des programmes C déclarent et modifient des variables en utilisant différentes structures de contrôle et différentes boucles.

A.1 Structures de contrôle

A.1.1 Branchements

Le programmeur peut exécuter des instructions selon plusieurs conditions à l'aide des mots clés `if` et `else`. Ces structures de contrôle permettent donc d'effectuer des tests afin d'exécuter conditionnellement certaines instructions. On parle aussi de branchements conditionnels.

Dans cet exemple, l'affichage dépendra des valeurs des variables `a` et `b` :

```c
int a = 0;
int b = 1;
// Modification de a et de b
if (a == b)     { printf("a est égal à b\n"); }
else if (a > b) { printf("a est strictement plus grand que b\n"); }
else            { printf("a est strictement plus petit que b\n"); }
```

D'autres branchements conditionnels existent mais ceux-ci sont, soit des cas particuliers par rapport à l'exemple ci-dessus, soit du *sucre syntaxique* (syntaxe équivalente considérée comme plus agréable à lire) comme `switch` avec `case` et `default`.

Il existe également des branchements inconditionnels que nous n'aborderons pas ici. Ces branchements sont aussi appelés sauts. Ces derniers sont accessibles grâce aux mots clés `break` (qui permet de sortir de la boucle en cours), `continue` (qui permet de passer à l'*itération* suivante de la boucle en cours), `return` (qui permet de retourner la valeur d'une fonction) et `goto` (qui permet de sauter à une étiquette).

D'autres structures de contrôle sont disponibles, il s'agit des boucles.

A.1.2 Boucles

Le programmeur traite souvent un ensemble de données de la même façon. Pour cela, il peut utiliser les boucles `for`, `while` et `do while`. Les instructions à l'intérieur de la boucle peuvent être exécutées plusieurs fois. On appelle itération une exécution de la boucle. Généralement, lorsque le nombre d'itérations est (plus ou moins) connu, on utilise la boucle `for`. Le nombre d'itérations exact est rarement connu à l'écriture du programme mais dépend souvent d'un paramètre, par exemple, la taille des données.

Dans cet exemple, à la ligne 1, on déclare un tableau de 3 éléments valant respectivement $11,38$, $5,24$ et $19,89$. Le nom de la variable correspondant au tableau est `tab`. On affiche la valeur de chaque élément grâce à l'instruction présente dans la boucle `for` de la ligne 4. Les lignes 7 et 8 permettent de calculer la somme des éléments du tableau `tab` dans la variable `somme` en utilisant la même boucle `for` :

```c
double tab[3] = { 11.38, 5.24, 19.89 };

printf("Valeurs = ");
```

```
for (size_t i = 0; i < 3; ++i) { printf("%f ", tab[i]); }
printf("\n"); // Valeurs = 11.380000 5.240000 19.890000

double somme = 0.0;
for (size_t i = 0; i < 3; ++i) { somme += tab[i]; }
printf("Somme = %f\n", somme); // Somme = 36.510000
```

Les deux autres boucles sont : la boucle `while`, qui permet d'exécuter des instructions tant que la condition donnée est vraie, et la boucle `do while`, qui permet de re-exécuter les instructions tant que la condition donnée est vraie.

Ces différentes structures de contrôle (branchements conditionnels, branchements inconditionnels, boucles) permettent de choisir quelles instructions et le nombre de fois où elle doivent être exécutées. La plupart des instructions permettent de modifier les variables, et donc modifier la mémoire.

A.2 Accès à la mémoire

Programmer en C revient à jouer avec la mémoire. Un programmeur C est habitué à utiliser deux types de mémoire : la pile et le tas. Par défaut, le C utilise la pile et gère lui-même la durée de vie de la mémoire allouée : une variable créée dans le bloc existera uniquement dans ce bloc et ses sous blocs On parle d'allocation statique.

Dans cet exemple, le tableau `tab` existe à partir du début du bloc et est détruit automatiquement à la fin de ce même bloc. Il est alloué dans la pile :

```
{ // Début bloc
    int tab[128]; // Tableau statique contenant 128 entiers
    // ... Accès au ième élément avec tab[i]
} // tab est automatiquement détruite par le compilateur
```

Comme cette mémoire, la pile, est relativement petite (quelques mégaoctets), le programmeur est souvent amené à utiliser la seconde mémoire : le tas. Le tas est beaucoup plus grand (plusieurs gigaoctets). L'allocation mémoire sur le tas se fait manuellement. C'est donc le programmeur qui est responsable de la durée de vie des variables allouées sur le tas. On parle d'allocation dynamique. Les allocations sur le tas se font avec la fonction `malloc` qui, retourne une adresse que l'on peut récupérer dans un pointeur. La fonction `free` permet de libérer la mémoire.

Dans cet exemple, à la ligne 1, on alloue un tableau dynamique `tab` contenant 128 entiers non initialisés. Lorsque le programmeur n'a plus besoin du tableau, il libére la mémoire avec la fonction `free` à la ligne 3. L'opérateur `[]` est disponible sur les pointeurs (et fait " *la bonne chose* "). Cela permet d'accéder à un élément du tableau avec la même notation que les tableaux alloués statiquement :

```
int * tab = malloc(128 * sizeof(int)); // Tableau dynamique
// ... Accès au ième élément avec tab[i]
free(tab); // Libération explicite de la mémoire prise par tab
```

L'accès à une valeur du tableau se fait classiquement via l'opérateur `[]` qui prend un indice. L'utilisation des tableaux avec l'opérateur `[]` est indépendante de son mode d'allocation (statique ou dynamique.)

Dans cet exemple, à la ligne 1, on déclare le tableau à deux dimensions nommé `tab` avec 8 lignes et 8 colonnes. Chaque valeur est initialisée à 0. La ligne 4 affecte la valeur 1 à la première valeur de `tab` (indice de ligne égal à 0 et indice de colonne égal à 0). Les lignes 5 à 13 parcourent le tableau avec deux boucles `for` imbriquées et l'instruction de la ligne 10 permet d'affecter une valeur à l'élément `i`, `j` de `tab`. Les boucles imbriquées des lignes 15 à 22 permettent d'afficher le tableau.

```
int tab[8][8] = { { 0 } };
```

```
// Triangle de Pascal                                                                3
tab[0][0] = 1;                                                                       4
for (size_t i = 1; i < 8; ++i)                                                       5
{                                                                                    6
    tab[i][0] = 1;                                                                   7
    for (size_t j = 1; j < i; ++j)                                                   8
    {                                                                                9
        tab[i][j] = tab[i - 1][j - 1] + tab[i - 1][j];                               10
    }                                                                                11
    tab[i][i] = 1;                                                                   12
}                                                                                    13
                                                                                     14
for (size_t i = 0; i < 8; ++i)     //  1  0  0  0  0  0  0  0                        15
{                                  //  1  1  0  0  0  0  0  0                        16
    for (size_t j = 0; j < 8; ++j) //  1  2  1  0  0  0  0  0                        17
    {                              //  1  3  3  1  0  0  0  0                        18
        printf("%2d ", tab[i][j]); //  1  4  6  4  1  0  0  0                        19
    }                              //  1  5 10 10  5  1  0  0                        20
    printf("\n");                  //  1  6 15 20 15  6  1  0                        21
}                                  //  1  7 21 35 35 21  7  1                        22
```

Une autre utilisation est possible : l'arithmétique des pointeurs. L'accès à l'élément `i` du tableau `tab` avec l'instruction `tab[i]` est équivalent à cette instruction : `*(tab + i)`. On décale le pointeur `tab` de `i` puis on accède à l'élément pointé avec l'opérateur unaire `*`. L'arithmétique des pointeurs fait " la bonne chose " dans le sens où les opérations arithmétiques sur les pointeurs prennent en compte la taille des éléments pointés. Il s'agit de deux styles d'écriture différents mais équivalents.

Pour des raisons de fragmentation de la mémoire, il est légitime de linéariser les tableaux (pour ne pas avoir de " trou " et profiter pleinement des mémoires cache). Par exemple, au lieu de déclarer un tableau deux dimensions `N` lignes, `M` colonnes ; on peut créer un tableau une dimension $N \times M$ éléments. L'accès `tab2D[i][j]` est alors remplacé par `tab1D[i * M + j]`.

Cet exemple est équivalent au précédent mais utilise l'arithmétique des pointeurs et linéarise le tableau deux dimensions. De plus, il utilise l'allocation dynamique ; il ne faut donc pas oublier de libérer la mémoire, ce qui est fait à la ligne 24.

```
int * tab = calloc(8 * 8, sizeof(int));                                              1
                                                                                     2
// Triangle de Pascal                                                                3
*tab = 1;                                                                            4
for (size_t i = 1; i < 8; ++i)                                                       5
{                                                                                    6
    *(tab + i * 8 + 0) = 1;                                                          7
    for (size_t j = 1; j < i; ++j)                                                   8
    {                                                                                9
        *(tab + i * 8 + j) =                                                         10
            *(tab + (i - 1) * 8 + (j - 1)) + *(tab + (i - 1) * 8 + j);               11
    }                                                                                12
    *(tab + i * 8 + i) = 1;                                                          13
}                                                                                    14
                                                                                     15
for (size_t i = 0; i < 8; ++i)                 //  1  0  0  0  0  0  0  0           16
{                                              //  1  1  0  0  0  0  0  0           17
    for (size_t j = 0; j < 8; ++j)             //  1  2  1  0  0  0  0  0           18
    {                                          //  1  3  3  1  0  0  0  0           19
        printf("%2d ", *(tab + i * 8 + j)); //  1  4  6  4  1  0  0  0           20
    }                                          //  1  5 10 10  5  1  0  0           21
    printf("\n");                              //  1  6 15 20 15  6  1  0           22
```

```
}                                                                    23
                                                                     24
free(tab);                                                           25
```

L'utilisation de l'arithmétique des pointeurs associée à la linéarisation des accès en deux dimensions rend le code source moins lisible, voire même incompréhensible pour les débutants.

Le *modèle polyédrique* n'est pas lié au langage C mais il permet de représenter une sous-partie du langage. Avec certaines contraintes sur la forme du code source, on peut alors le traduire dans cette représentation mathématique. Généralement, on s'intéresse au noyau de calcul. Il s'agit de codes relativement courts qui prennent la majorité du temps d'exécution du programme. Les *plateformes de compilation polyédriques* nous autorisent à optimiser ces noyaux de façon agressive.

Annexe B

PeriScop

FIGURE B.1 – Code source en langage C du noyau de calcul `jacobi-2d-imper` venant des PolyBench/C 3.2, généré par CLooG 0.18.1, après optimisation par Pluto 0.11.4 avec l'option `--parallel`

```
if ((N >= 3) && (T_STEPS >= 1)) {
  for (t3=1; t3<=N-2; t3++) {
    B[1][t3] = 0.2 * (A[1][t3] + A[1][t3-1] + A[1][1+t3] + A[1+1][t3] + A[1 -1][t3]);;
  }
  for (t1=2; t1<=min(N-2,3*T_STEPS-2); t1++) {
    if ((2*t1+1)%3 == 0) {
      for (t3=ceild(2*t1+1,3); t3<=floord(2*t1+3*N-8,3); t3++) {
        B[1][((-2*t1+3*t3+2)/3)] = 0.2 * (A[1][((-2*t1+3*t3+2)/3)] + A[1][((-2*t1+3*t3+2)/3)-1] +
     A[1][1+((-2*t1+3*t3+2)/3)] + A[1+1][((-2*t1+3*t3+2)/3)] + A[1 -1][((-2*t1+3*t3+2)/3)]);;
      }
    }
    lbp=ceild(2*t1+2,3);
    ubp=t1;
#pragma omp parallel for private(lbv,ubv,t3)
    for (t2=lbp; t2<=ubp; t2++) {
      B[(-2*t1+3*t2)][1] = 0.2 * (A[(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)][1 -1] + A[(-2*t1+3*t2)][1+1] +
     A[1+(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)-1][1]);;
      for (t3=2*t1-2*t2+2; t3<=2*t1-2*t2+N-2; t3++) {
        B[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] = 0.2 * (A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)-1] +
     A[(-2*t1+3*t2)][1+(-2*t1+2*t2+t3)] + A[1+(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)-1][(-2*t1+2*t2+t3)]);;
        A[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)] = B[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)];;
      }
      A[(-2*t1+3*t2-1)][(N-2)] = B[(-2*t1+3*t2-1)][(N-2)];;
    }
  }
  if (N == 3) {
    for (t1=2; t1<=3*T_STEPS-2; t1++) {
      if ((2*t1+1)%3 == 0) {
        B[1][1] = 0.2 * (A[1][1] + A[1][1 -1] + A[1][1+1] + A[1+1][1] + A[1 -1][1]);;
      }
      if ((2*t1+2)%3 == 0) {
        A[1][1] = B[1][1];;
      }
    }
  }
  for (t1=3*T_STEPS-1; t1<=N-2; t1++) {
    lbp=t1-T_STEPS+1;
    ubp=t1;
#pragma omp parallel for private(lbv,ubv,t3)
    for (t2=lbp; t2<=ubp; t2++) {
      B[(-2*t1+3*t2)][1] = 0.2 * (A[(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)][1 -1] + A[(-2*t1+3*t2)][1+1] +
     A[1+(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)-1][1]);;
      for (t3=2*t1-2*t2+2; t3<=2*t1-2*t2+N-2; t3++) {
        B[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] = 0.2 * (A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)-1] +
     A[(-2*t1+3*t2)][1+(-2*t1+2*t2+t3)] + A[1+(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)-1][(-2*t1+2*t2+t3)]);;
        A[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)] = B[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)];;
      }
      A[(-2*t1+3*t2-1)][(N-2)] = B[(-2*t1+3*t2-1)][(N-2)];;
    }
  }
  if (N >= 4) {
    for (t1=N-1; t1<=3*T_STEPS-2; t1++) {
      if ((2*t1+1)%3 == 0) {
        for (t3=ceild(2*t1+1,3); t3<=floord(2*t1+3*N-8,3); t3++) {
          B[1][((-2*t1+3*t3+2)/3)] = 0.2 * (A[1][((-2*t1+3*t3+2)/3)] + A[1][((-2*t1+3*t3+2)/3)-1] +
     A[1][1+((-2*t1+3*t3+2)/3)] + A[1+1][((-2*t1+3*t3+2)/3)] + A[1 -1][((-2*t1+3*t3+2)/3)]);;
        }
      }
      lbp=ceild(2*t1+2,3);
      ubp=floord(2*t1+N-2,3);
```

```
#pragma omp parallel for private(lbv,ubv,t3)
      for (t2=lbp; t2<=ubp; t2++) {
        B[(-2*t1+3*t2)][1] = 0.2 * (A[(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)][1 -1] + A[(-2*t1+3*t2)][1+1] +
     A[1+(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)-1][1]);;
        for (t3=2*t1-2*t2+2; t3<=2*t1-2*t2+N-2; t3++) {
          B[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] = 0.2 * (A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)-1]
     + A[(-2*t1+3*t2)][1+(-2*t1+2*t2+t3)] + A[1+(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)-1][(-2*t1+2*t2+t3)]);;
          A[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)] = B[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)];;
        }
        A[(-2*t1+3*t2-1)][(N-2)] = B[(-2*t1+3*t2-1)][(N-2)];;
       }
       if ((2*t1+N+2)%3 == 0) {
        for (t3=ceild(2*t1-2*N+8,3); t3<=floord(2*t1+N-1,3); t3++) {
          A[(N-2)][((-2*t1+3*t3+3*t2+N-5)/3)] = B[(N-2)][((-2*t1+3*t3+3*t2+N-5)/3)];;
        }
       }
      }
    }
    for (t1=max(N-1,3*T_STEPS-1); t1<=3*T_STEPS+N-5; t1++) {
      lbp=t1-T_STEPS+1;
      ubp=floord(2*t1+N-2,3);
#pragma omp parallel for private(lbv,ubv,t3)
      for (t2=lbp; t2<=ubp; t2++) {
        B[(-2*t1+3*t2)][1] = 0.2 * (A[(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)][1 -1] + A[(-2*t1+3*t2)][1+1] +
       A[1+(-2*t1+3*t2)][1] + A[(-2*t1+3*t2)-1][1]);;
        for (t3=2*t1-2*t2+2; t3<=2*t1-2*t2+N-2; t3++) {
          B[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] = 0.2 * (A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)][(-2*t1+2*t2+t3)-1] +
       A[(-2*t1+3*t2)][1+(-2*t1+2*t2+t3)] + A[1+(-2*t1+3*t2)][(-2*t1+2*t2+t3)] + A[(-2*t1+3*t2)-1][(-2*t1+2*t2+t3)]);;
          A[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)] = B[(-2*t1+3*t2-1)][(-2*t1+2*t2+t3-1)];;
        }
        A[(-2*t1+3*t2-1)][(N-2)] = B[(-2*t1+3*t2-1)][(N-2)];;
      }
      if ((2*t1+N+2)%3 == 0) {
        for (t3=ceild(2*t1-2*N+8,3); t3<=floord(2*t1+N-1,3); t3++) {
          A[(N-2)][((-2*t1+3*t3+3*t2+N-5)/3)] = B[(N-2)][((-2*t1+3*t3+3*t2+N-5)/3)];;
        }
      }
    }
    for (t3=2*T_STEPS; t3<=2*T_STEPS+N-3; t3++) {
      A[(N-2)][(t3-2*T_STEPS+1)] = B[(N-2)][(t3-2*T_STEPS+1)];;
    }
  }
}
```

Figure B.2 – Code source en langage C du noyau de calcul `jacobi-2d-imper` venant des PolyBench/C 3.2, généré par CLooG 0.18.1, après optimisation par Pluto 0.11.4 avec les options `--tile` et `--parallel`

```
  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) && (T_STEPS >= 1)) {
  for (t1=0; t1<=floord(3*T_STEPS+N-4,32); t1++) {
    lbp=max(ceild(2*t1,3),ceild(32*t1-T_STEPS+1,32));
    ubp=min(min(floord(2*T_STEPS+N-3,32),floord(64*t1+N+61,96)),t1);
#pragma omp parallel for private(lbv,ubv,t3,t4,t5,t6)
    for (t2=lbp; t2<=ubp; t2++) {
      for (t3=max(ceild(32*t2-N-28,32),2*t1-2*t2);
     t3<=min(min(floord(2*T_STEPS+N-3,32),floord(32*t2+N+28,32)),floord(64*t1-64*t2+N+61,32)); t3++) {
        if ((t1 <= floord(64*t2+32*t3-N+1,64)) && (t2 <= t3-1)) {
          if ((N+1)%2 == 0) {
            for (t5=max(32*t2,32*t3-N+3); t5<=32*t2+31; t5++) {
              A[(-32*t3+t5+N-2)][(N-2)] = B[(-32*t3+t5+N-2)][(N-2)];;
            }
          }
        }
        if ((t1 <= floord(96*t2-N+1,64)) && (t2 >= t3)) {
          if ((N+1)%2 == 0) {
            for (t6=max(32*t3,32*t2-N+3); t6<=min(32*t2,32*t3+31); t6++) {
              A[(N-2)][(-32*t2+t6+N-2)] = B[(N-2)][(-32*t2+t6+N-2)];;
            }
          }
        }
        if ((N == 3) && (t2 == t3)) {
          for (t4=16*t2; t4<=min(min(T_STEPS-1,16*t2+14),32*t1-32*t2+31); t4++) {
            B[1][1] = 0.2 * (A[1][1] + A[1][1 -1] + A[1][1+1] + A[1+1][1] + A[1 -1][1]);;
            A[1][1] = B[1][1];;
          }
        }
        if (t2 == t3) {
          for (t4=max(ceild(32*t2-N+2,2),32*t1-32*t2);
       t4<=min(min(min(floord(32*t2-N+32,2),T_STEPS-1),16*t2-1),32*t1-32*t2+31); t4++) {
            for (t5=32*t2; t5<=2*t4+N-2; t5++) {
              for (t6=32*t2; t6<=2*t4+N-2; t6++) {
```

```
          B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
          A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
        }
        A[(-2*t4+t5-1)][(N-2)] = B[(-2*t4+t5-1)][(N-2)];;
      }
      for (t6=32*t2; t6<=2*t4+N-1; t6++) {
        A[(N-2)][(-2*t4+t6-1)] = B[(N-2)][(-2*t4+t6-1)];;
      }
    }
  }
  for (t4=max(max(ceild(32*t2-N+33,2),ceild(32*t3-N+2,2)),32*t1-32*t2);
t4<=min(min(min(floord(32*t3-N+32,2),T_STEPS-1),16*t2-1),32*t1-32*t2+31); t4++) {
    for (t5=32*t2; t5<=32*t2+31; t5++) {
      for (t6=32*t3; t6<=2*t4+N-2; t6++) {
        B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
        A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
      }
      A[(-2*t4+t5-1)][(N-2)] = B[(-2*t4+t5-1)][(N-2)];;
    }
  }
  for (t4=max(max(ceild(32*t2-N+2,2),ceild(32*t3-N+33,2)),32*t1-32*t2);
t4<=min(min(min(floord(32*t2-N+32,2),T_STEPS-1),16*t3-1),32*t1-32*t2+31); t4++) {
    for (t5=32*t2; t5<=2*t4+N-2; t5++) {
      for (t6=32*t3; t6<=32*t3+31; t6++) {
        B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
        A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
      }
    }
    for (t6=32*t3; t6<=32*t3+31; t6++) {
      A[(N-2)][(-2*t4+t6-1)] = B[(N-2)][(-2*t4+t6-1)];;
    }
  }
  for (t4=max(max(ceild(32*t2-N+33,2),ceild(32*t3-N+33,2)),32*t1-32*t2);
t4<=min(min(min(T_STEPS-1,16*t2-1),16*t3-1),32*t1-32*t2+31); t4++) {
    for (t5=32*t2; t5<=32*t2+31; t5++) {
      for (t6=32*t3; t6<=32*t3+31; t6++) {
        B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
        A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
      }
    }
  }
  if ((N >= 4) && (t2 == t3)) {
    for (t4=16*t2; t4<=min(min(floord(32*t2-N+32,2),T_STEPS-1),32*t1-32*t2+31); t4++) {
      for (t6=2*t4+1; t6<=2*t4+N-2; t6++) {
        B[1][(-2*t4+t6)] = 0.2 * (A[1][(-2*t4+t6)] + A[1][(-2*t4+t6)-1] + A[1][1+(-2*t4+t6)] + A[1+1][(-2*t4+t6)] +
A[1 -1][(-2*t4+t6)]);;
      }
      for (t5=2*t4+2; t5<=2*t4+N-2; t5++) {
        B[(-2*t4+t5)][1] = 0.2 * (A[(-2*t4+t5)][1] + A[(-2*t4+t5)][1 -1] + A[(-2*t4+t5)][1+1] + A[1+(-2*t4+t5)][1]
+ A[(-2*t4+t5)-1][1]);;
        for (t6=2*t4+2; t6<=2*t4+N-2; t6++) {
          B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
          A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
        }
        A[(-2*t4+t5-1)][(N-2)] = B[(-2*t4+t5-1)][(N-2)];;
      }
      for (t6=2*t4+2; t6<=2*t4+N-1; t6++) {
        A[(N-2)][(-2*t4+t6-1)] = B[(N-2)][(-2*t4+t6-1)];;
      }
    }
  }
  if (t2 == t3) {
    for (t4=max(ceild(32*t2-N+33,2),16*t2); t4<=min(min(T_STEPS-1,16*t2+14),32*t1-32*t2+31); t4++) {
      for (t6=2*t4+1; t6<=32*t2+31; t6++) {
        B[1][(-2*t4+t6)] = 0.2 * (A[1][(-2*t4+t6)] + A[1][(-2*t4+t6)-1] + A[1][1+(-2*t4+t6)] + A[1+1][(-2*t4+t6)] +
A[1 -1][(-2*t4+t6)]);;
      }
      for (t5=2*t4+2; t5<=32*t2+31; t5++) {
        B[(-2*t4+t5)][1] = 0.2 * (A[(-2*t4+t5)][1] + A[(-2*t4+t5)][1 -1] + A[(-2*t4+t5)][1+1] + A[1+(-2*t4+t5)][1]
+ A[(-2*t4+t5)-1][1]);;
        for (t6=2*t4+2; t6<=32*t2+31; t6++) {
          B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
          A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
        }
      }
    }
  }
  for (t4=max(ceild(32*t3-N+2,2),16*t2);
t4<=min(min(min(min(floord(32*t3-N+32,2),T_STEPS-1),16*t2+14),16*t3-1),32*t1-32*t2+31); t4++) {
    for (t6=32*t3; t6<=2*t4+N-2; t6++) {
      B[1][(-2*t4+t6)] = 0.2 * (A[1][(-2*t4+t6)] + A[1][(-2*t4+t6)-1] + A[1][1+(-2*t4+t6)] + A[1+1][(-2*t4+t6)] +
```

```
    A[1 -1][(-2*t4+t6)]);;
        }
        for (t5=2*t4+2; t5<=32*t2+31; t5++) {
          for (t6=32*t3; t6<=2*t4+N-2; t6++) {
            B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
    A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
            A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
          }
          A[(-2*t4+t5-1)][(N-2)] = B[(-2*t4+t5-1)][(N-2)];;
        }
      }
      for (t4=max(ceild(32*t3-N+33,2),16*t2); t4<=min(min(min(T_STEPS-1,16*t2+14),16*t3-1),32*t1-32*t2+31); t4++) {
        for (t6=32*t3; t6<=32*t3+31; t6++) {
          B[1][(-2*t4+t6)] = 0.2 * (A[1][(-2*t4+t6)] + A[1][(-2*t4+t6)-1] + A[1][1+(-2*t4+t6)] + A[1+1][(-2*t4+t6)] +
    A[1 -1][(-2*t4+t6)]);;
        }
        for (t5=2*t4+2; t5<=32*t2+31; t5++) {
          for (t6=32*t3; t6<=32*t3+31; t6++) {
            B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
    A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
            A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
          }
        }
      }
      for (t4=max(ceild(32*t2-N+2,2),16*t3);
    t4<=min(min(min(min(floord(32*t2-N+32,2),T_STEPS-1),16*t2-1),16*t3+14),32*t1-32*t2+31); t4++) {
        for (t5=32*t2; t5<=2*t4+N-2; t5++) {
          B[(-2*t4+t5)][1] = 0.2 * (A[(-2*t4+t5)][1] + A[(-2*t4+t5)][1 -1] + A[(-2*t4+t5)][1+1] + A[1+(-2*t4+t5)][1] +
    A[(-2*t4+t5)-1][1]);;
          for (t6=2*t4+2; t6<=32*t3+31; t6++) {
            B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
    A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
            A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
          }
        }
        for (t6=2*t4+2; t6<=32*t3+31; t6++) {
          A[(N-2)][(-2*t4+t6-1)] = B[(N-2)][(-2*t4+t6-1)];;
        }
      }
      for (t4=max(ceild(32*t2-N+33,2),16*t3); t4<=min(min(min(T_STEPS-1,16*t2-1),16*t3+14),32*t1-32*t2+31); t4++) {
        for (t5=32*t2; t5<=32*t2+31; t5++) {
          B[(-2*t4+t5)][1] = 0.2 * (A[(-2*t4+t5)][1] + A[(-2*t4+t5)][1 -1] + A[(-2*t4+t5)][1+1] + A[1+(-2*t4+t5)][1] +
    A[(-2*t4+t5)-1][1]);;
          for (t6=2*t4+2; t6<=32*t3+31; t6++) {
            B[(-2*t4+t5)][(-2*t4+t6)] = 0.2 * (A[(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)][(-2*t4+t6)-1] +
    A[(-2*t4+t5)][1+(-2*t4+t6)] + A[1+(-2*t4+t5)][(-2*t4+t6)] + A[(-2*t4+t5)-1][(-2*t4+t6)]);;
            A[(-2*t4+t5-1)][(-2*t4+t6-1)] = B[(-2*t4+t5-1)][(-2*t4+t6-1)];;
          }
        }
      }
      if ((t1 >= ceild(3*t2-1,2)) && (t2 <= min(floord(T_STEPS-16,16),t3-1))) {
        for (t6=32*t3; t6<=min(32*t3+31,32*t2+N+28); t6++) {
          B[1][(-32*t2+t6-30)] = 0.2 * (A[1][(-32*t2+t6-30)] + A[1][(-32*t2+t6-30)-1] + A[1][1+(-32*t2+t6-30)] +
    A[1+1][(-32*t2+t6-30)] + A[1 -1][(-32*t2+t6-30)]);;
        }
      }
      if ((t1 >= ceild(2*t2+t3-1,2)) && (t2 >= t3) && (t3 <= floord(T_STEPS-16,16))) {
        for (t5=max(32*t2,32*t3+31); t5<=min(32*t2+31,32*t3+N+28); t5++) {
          B[(-32*t3+t5-30)][1] = 0.2 * (A[(-32*t3+t5-30)][1] + A[(-32*t3+t5-30)][1 -1] + A[(-32*t3+t5-30)][1+1] +
    A[1+(-32*t3+t5-30)][1] + A[(-32*t3+t5-30)-1][1]);;
        }
      }
    }
  }
}
/* End of CLooG code */
```

Figure B.3 – SCoP utilisant le format OpenScop, extrait depuis le code source C `jacobi-2d-imper` venant des PolyBench/C 3.2 par Clan 0.7.1

```
[Clan] Info: parsing file #1 (jacobi-2d-imper.cpp)
# [Generated by Clan 0.7.1]
# [File generated by the OpenScop Library 0.8.4]

<OpenScop>

# ============================================== Global
# Language
C

# Context
CONTEXT
0 4 0 0 0 2

# Parameters are provided
1
<strings>
T_STEPS N
</strings>

# Number of statements
2

# ============================================== Statement 1
# Number of relations describing the statement:
8

# ----------------------------------------------  1.1 Domain
DOMAIN
8 7 3 0 0 2
# e/i|  t    i    j |T_S.   N |  1
   1    1    0    0    0    0    0    ## t >= 0
   1   -1    0    0    1    0   -1    ## -t+T_STEPS-1 >= 0
   1    0    0    0    1    0   -1    ## T_STEPS-1 >= 0
   1    0    1    0    0    0   -1    ## i-1 >= 0
   1    0   -1    0    0    1   -2    ## -i+N-2 >= 0
   1    0    0    0    0    1   -3    ## N-3 >= 0
   1    0    0    1    0    0   -1    ## j-1 >= 0
   1    0    0   -1    0    1   -2    ## -j+N-2 >= 0

# ----------------------------------------------  1.2 Scattering
SCATTERING
7 14 7 3 0 2
# e/i| c1   c2   c3   c4   c5   c6   c7 |  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0 |  0    0    0    0    0    0    ## c1 == 0
   0    0   -1    0    0    0    0    0 |  1    0    0    0    0    0    ## c2 == t
   0    0    0   -1    0    0    0    0 |  0    0    0    0    0    0    ## c3 == 0
   0    0    0    0   -1    0    0    0 |  0    1    0    0    0    0    ## c4 == i
   0    0    0    0    0   -1    0    0 |  0    0    0    0    0    0    ## c5 == 0
   0    0    0    0    0    0   -1    0 |  0    0    1    0    0    0    ## c6 == j
   0    0    0    0    0    0    0   -1 |  0    0    0    0    0    0    ## c7 == 0

# ----------------------------------------------  1.3 Access
WRITE
3 10 3 3 0 2
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    6    ## Arr == B
   0    0   -1    0    0    1    0    0    0    ## [1] == i
   0    0    0   -1    0    0    1    0    0    ## [2] == j

READ
3 10 3 3 0 2
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    7    ## Arr == A
   0    0   -1    0    0    1    0    0    0    ## [1] == i
   0    0    0   -1    0    0    1    0    0    ## [2] == j

READ
3 10 3 3 0 2
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    7    ## Arr == A
   0    0   -1    0    0    1    0    0    0    ## [1] == i
   0    0    0   -1    0    0    1    0    0   -1    ## [2] == j-1

READ
3 10 3 3 0 2
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    7    ## Arr == A
   0    0   -1    0    0    1    0    0    0    ## [1] == i
   0    0    0   -1    0    0    1    0    0    1    ## [2] == j+1

READ
3 10 3 3 0 2
```

```
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    7    ## Arr == A
   0    0   -1    0    0    1    0    0    0    ## [1] == i+1
   0    0    0   -1    0    0    1    0    0    ## [2] == j

READ
3 10 3 3 0 2
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    7    ## Arr == A
   0    0   -1    0    0    1    0    0   -1    ## [1] == i-1
   0    0    0   -1    0    0    1    0    0    ## [2] == j

# ------------------------------------------------  1.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
3
# List of original iterators
t i j
# Statement body expression
B[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][1+j] + A[1+i][j] + A[i-1][j]);
</body>

# ================================================ Statement 2
# Number of relations describing the statement:
4

# ------------------------------------------------  2.1 Domain
DOMAIN
8 7 3 0 0 2
# e/i|  t    i    j |T_S.   N |  1
   1    1    0    0    0    0    0    ## t >= 0
   1   -1    0    0    1    0   -1    ## -t+T_STEPS-1 >= 0
   1    0    0    0    1    0   -1    ## T_STEPS-1 >= 0
   1    0    1    0    0    0   -1    ## i-1 >= 0
   1    0   -1    0    0    1   -2    ## -i+N-2 >= 0
   1    0    0    0    0    1   -3    ## N-3 >= 0
   1    0    0    1    0    0   -1    ## j-1 >= 0
   1    0    0   -1    0    1   -2    ## -j+N-2 >= 0

# ------------------------------------------------  2.2 Scattering
SCATTERING
7 14 7 3 0 2
# e/i| c1   c2   c3   c4   c5   c6   c7 |  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    0    0    0    0    0    0    ## c1 == 0
   0    0   -1    0    0    0    0    0    1    0    0    0    0    0    ## c2 == t
   0    0    0   -1    0    0    0    0    0    0    0    0    0    1    ## c3 == 1
   0    0    0    0   -1    0    0    0    0    1    0    0    0    0    ## c4 == i
   0    0    0    0    0   -1    0    0    0    0    0    0    0    0    ## c5 == 0
   0    0    0    0    0    0   -1    0    0    0    1    0    0    0    ## c6 == j
   0    0    0    0    0    0    0   -1    0    0    0    0    0    0    ## c7 == 0

# ------------------------------------------------  2.3 Access
WRITE
3 10 3 3 0 2
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    7    ## Arr == A
   0    0   -1    0    0    1    0    0    0    ## [1] == i
   0    0    0   -1    0    0    1    0    0    ## [2] == j

READ
3 10 3 3 0 2
# e/i| Arr  [1]  [2]|  t    i    j |T_S.   N |  1
   0   -1    0    0    0    0    0    0    6    ## Arr == B
   0    0   -1    0    0    1    0    0    0    ## [1] == i
   0    0    0   -1    0    0    1    0    0    ## [2] == j

# ------------------------------------------------  2.4 Statement Extensions
# Number of Statement Extensions
1
<body>
# Number of original iterators
3
# List of original iterators
t i j
# Statement body expression
A[i][j] = B[i][j];
</body>

# ================================================ Extensions
<scatnames>
b0 t b1 i b2 j b3
</scatnames>

<arrays>
# Number of arrays
```

```
7
# Mapping array-identifiers/array-names
1 t
2 T_STEPS
3 i
4 N
5 j
6 B
7 A
</arrays>

<coordinates>
# File name
jacobi-2d-imper.cpp
# Starting line and column
54 0
# Ending line and column
74 0
# Indentation
0
</coordinates>

</OpenScop>
```

Bibliographie

[1] Lénaïc Bagnères and Cédric Bastoul. Switchable scheduling for runtime adaptation of optimization. In *Euro-Par*, pages 222–233, 2014.

[2] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Opening Polyhedral Compiler's Black Box. In *CGO 2016 - 14th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Barcelona, Spain, March 2016.

[3] Cédric Bastoul. *Improving Data Locality in Static Control Programs.* PhD thesis, University Paris 6, Pierre et Marie Curie, France, December 2004.

[4] Amy Wingmui Lim. *Improving Parallelism and Data Locality with Affine Partitioning.* PhD thesis, 2001. AAI3028136.

[5] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 ACM Conf. on Programming language design and implementation*, Tucson, USA, June 2008.

[6] Qingda Lu, Christophe Alias, Uday Bondhugula, Thomas Henretty, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, P. Sadayappan, Yongjian Chen, Haibo Lin, and Tin-Fook Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 348–357, Sept 2009.

[7] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[8] Nicolas Vasilache, Benoît Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT-2 : 2nd International Workshop on Polyhedral Compilation Techniques, Paris, France, January*, Paris, France, Jan 2012.

[9] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*, Seattle, WA, June 2013. ACM Press.

[10] Konrad Trifunovic, Konrad Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 327–337, Sept 2009.

[11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Grösslinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *IMPACT 2011 First International Workshop on Polyhedral Compilation Techniques*, Chamonix, France, 2011.

[12] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John 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.

[13] Cédric Bastoul. Extracting polyhedral representation from high level languages. Technical report, LRI, Paris-Sud University, France, 2008. Related to the Clan tool.

[14] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *IMPACT 2012 Second International Workshop on Polyhedral Compilation Techniques*, Paris, France, 2012.

[15] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools.* Pearson Education, Inc, 1986.

[16] Cédric Bastoul. *Contributions to High-Level Program Optimization.* Habilitation Thesis. Paris-Sud University, France, December 2012.

[17] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[18] Sven Verdoolaege. Integer set library : Manual. Technical report, January 2016.

[19] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4) :12 :1–12 :50, July 2015.

[20] Sébastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, P. Jouvelot, G.-A. Silber, and N. Vasilache. GRAPHITE : Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developper's Summit*, pages 179–198, Ottawa, Canada, June 2006.

[21] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT'10, pages 343–352, Vienna, Austria, 2010.

[22] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. Graphite two years after : First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, 2010.

[23] Konrad Trifunovic and Albert Cohen. Enabling more optimizations in GRAPHITE : ignoring memory-based dependences. In *Proceedings of the 8th GCC Developper's Summit*, Ottawa, Canada, October 2010.

[24] Sven Verdoolaege. *isl* : An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software*, pages 299–302, Kobe, Japan, September 2010.

[25] Aditya Kumar and Sebastian Pop. Scop detection : A fast algorithm for industrial compilers. In *IMPACT 2016 Sixth International Workshop on Polyhedral Compilation Techniques*, Prague, Czech Republic, 2016.

[26] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II : multidimensional time. *Int. J. of Parallel Programming*, 21(6) :389–420, December 1992.

[27] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL : a framework for composing high-level loop transformations. Technical Report 08-897, USC Computer Science, June 2008.

[28] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. of Parallel Programming*, 34(3) :261–317, June 2006.

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

[30] Paul Feautrier. The Power of Polynomials. In Alexandra Jimborean and Alain Darte, editors, *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, Amsterdam, Netherlands, January 2015.

[31] Imen Fassi and Philippe Clauss. XFOR : Filling the Gap between Automatic Loop Optimization and Peak Performance. In *4th International Symposium on Parallel and Distributed Computing*, Limassol, Cyprus, June 2015.

[32] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3) :243–268, 1988.

[33] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model : Part I, one-dimensional time. In *IEEE/ACM Fifth International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156, San Jose, California, March 2007. IEEE Computer Society press.

[34] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *SC'10*, New Orleans, USA, November 2010.

[35] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley Publishing Company, 1995.

[36] F. Bodin, T. Kisuki, P. M. W. Knijnenburg, M. F. P. O'Boyle, and E. Rohou. Iterative compilation in a non-linear optimisation space. In *W. on Profile and Feedback Directed Compilation*, Paris, October 1998.

[37] Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2) :3–35, 2000.

[38] Ramakrishna Upadrasta and Albert Cohen. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *ACM Symposium on Principles of Programming Languages*, POPL '13, pages 483–496, Rome, Italy, 2013.

[39] Alexandra Jimborean, Luis Mastrangelo, Vincent Loechner, and Philippe Clauss. VMAD : an Advanced Dynamic Program Analysis & Instrumentation Framework. In *CC - 21st International Conference on Compiler Construction*, volume 7210 of *LNCS*, pages 220–237, Tallinn, Estonia, March 2012.

[40] Benoit Pradelle, Philippe Clauss, and Vincent Loechner. Adaptive Runtime Selection of Parallel Schedules in the Polytope Model. In *19th High Performance Computing Symposium - HPC 2011*, Boston, United States, April 2011.

[41] M. Byler, J. R. B. Davies, C. Huson, B. Leasure, and M. Wolfe. Multiple version loops. In *International Conference on Parallel Processing*, August 1987.

[42] M.J. Voss and R. Eigenmann. ADAPT : Automated de-coupled adaptive program transformation. In *Int. Conf. on Parallel Processing*, pages 163–170, 2000.

[43] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin : Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO-42. 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 45–55, Dec 2009.

[44] M.K. Emani, Zheng Wang, and M.F.P. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–10, 2013.

[45] Lawrence Rauchwerger and David Padua. The LRPD test : speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232, New York, NY, USA, 1995. ACM.

[46] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3) :253–300, August 2005.

[47] Sanket Tavarageri, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Dynamic selection of tile sizes. In *18th IEEE Int. Conf. on High Performance Computing (HiPC'11)*, Bangalore, India, December 2011.

[48] Jean-François Dollinger and Vincent Loechner. Adaptive Runtime Selection for GPU. In *42nd International Conference on Parallel Processing*, pages 70–79, Lyon, France, 2013. IEEE.

[49] Jean-François Dollinger. *A framework for efficient execution on GPU and CPU+GPU systems*. Theses, Université de Strasbourg, July 2015.

[50] Jithendra Srinivas, Wei Ding, and Mahmut Kandemir. Reactive tiling. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 91–102, Washington, DC, USA, 2015. IEEE Computer Society.

[51] Grigori Fursin. Collective Tuning Initiative : automating and accelerating development and optimization of computing systems. In *GCC Developers' Summit*, Montreal, Canada, June 2009.

[52] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126.1, University of Maryland Institute for Advanced Computer Studies, 1993.

[53] W. Kelly. *Optimization within a Unified Transformation Framework*. doctoral thesis, University of Maryland, 1996.

[54] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 136–150, Houston, TX, 2010.

[55] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, Maria Garzaran, David Padua, and Keshav Pingali. A language for the compact representation of

multiple program versions. In *Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 4339 of *Lect. Notes in Computer Science*, pages 136–151, Hawthorne, New York, October 2005.

[56] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. Poet : Parameterized optimizations for empirical tuning. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[57] R. Müller-Pfefferkorn, W. Nagel, and B. Trenkler. Optimizing cache access : A tool for source-to-source transformations and real-life compiler tests. In *Euro-Par 2004, 10th International Euro-Par Conference*, pages 72–81, Pisa, august 2004.

[58] Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Clint : A direct manipulation tool for parallelizing compute-intensive program parts. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 109–112, Melbourne, VIC, Australia, 2014. IEEE.

[59] Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure : automatic parallelization with a helping hand. In *PACT'19 IEEE Intl. Conference on Parallel Architectures and Compilation Techniques*, pages 389–400, 2010.

[60] P. Larsen, R. Ladelsky, J. Lidman, S.A. McKee, S. Karlsson, and A. Zaks. Parallelizing more loops with compiler guided refactoring. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 410–419, Sept 2012.

[61] Nicklas Bo Jensen, Sven Karlsson, and Christian W. Probst. Compiler feedback using continuous dynamic compilation during development. In *International Workshop on Dynamic Compilation Everywhere*, Vienna, January 2014.

[62] Diana Göhringer and Jan Tepelmann. An interactive tool based on polly for detection and parallelization of loops. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '14, Vienna, Austria, 2014.

[63] Minjang Kim, Hyesoon Kim, and Chi-Keung Luk. Prospector : A dynamic data-dependence profiler to help parallel programming. In *HotPar'10 : Proceedings of the USENIX workshop on Hot Topics in parallelism*, 2010.

Résumé

Cette thèse porte sur l'adaptation automatique et semi-automatique des optimisations de programmes en utilisant le modèle polyédrique, un modèle mathématique permettant de représenter une certaine classe de programme. Ces optimisations peuvent être statiques, c'est-à-dire faites à la compilation, et dynamiques, c'est-à-dire faites à l'exécution. Une première approche automatique, statique et dynamique se concentre sur l'exploration, la sélection, la combinaison et l'évaluation de différentes versions issues du programme. Celles-ci sont générées par le compilateur polyédrique à partir du programme à optimiser afin d'exécuter la version la plus pertinente au bon moment. Une seconde approche semi-automatique et statique présente les optimisations faites par le compilateur polyédrique sous une forme compréhensible, rejouable et modifiable par n'importe quel développeur qui pourra alors raffiner l'optimisation afin de l'adapter à son cas.

Les architectures matérielles offrent une puissance de calcul de plus en plus élevée pour une consommation d'énergie relative de plus en plus faible mais deviennent de plus en plus complexes. C'est le rôle du programmeur expert de prendre en compte les différentes hiérarchies de mémoire et les multiples niveaux de parallélisme afin d'obtenir de bonnes performances. Malheureusement, pour la grande majorité des développeurs, il est difficile voire impossible, d'atteindre ces performances. Si la rapidité d'exécution est vue comme une fonctionnalité alors, dans de nombreux cas, celle-ci n'est pas toujours nécessaire. Généralement, l'utilisateur a besoin d'une certaine performance qui peut être éloignée des performances maximales. Il est raisonnable de laisser cette tâche aux outils automatiques ou semi-automatiques accessibles par n'importe quel programmeur non expert en techniques d'optimisation.

Les compilateurs offrent un excellent compromis entre le temps de développement et les performances de l'application. Actuellement l'efficacité de leurs optimisations reste limitée lorsque les architectures cibles sont des multi-cœurs ou si les applications demandent des calculs intensifs. Il est difficile de prendre en compte les nombreuses configurations existantes et les nombreux paramètres inconnus à la compilation et donc disponibles uniquement pendant l'exécution. En se basant sur les techniques de compilation polyédrique, nous proposons deux solutions complémentaires pour contribuer au traitement de ces problèmes.

Dans une première partie, nous présentons une technique automatique à la fois statique et dynamique permettant d'optimiser les boucles des programmes en utilisant les possibilités offertes par l'*auto-tuning* dynamique. Cette solution entièrement automatique explore de nombreuses versions et sélectionne les plus pertinentes à la compilation. Le choix de la version à exécuter se fait dynamiquement avec un faible surcoût grâce à la génération de versions interchangeables : un ensemble de transformations autorisant le passage d'une version à une autre du programme tout en faisant du calcul utile.

Dans une seconde partie, nous offrons à l'utilisateur une nouvelle façon d'interagir avec le compilateur polyédrique. Afin de comprendre et de modifier les transformations faites par l'optimiseur, nous traduisons depuis la représentation polyédrique utilisée en interne n'importe quelle transformation de boucles impactant l'ordonnancement des itérations en une séquence de transformations syntaxiques équivalente. Celle-ci est compréhensible et modifiable par les programmeurs. En offrant la possibilité au développeur d'examiner, modifier, améliorer, rejouer et de construire des optimisations complexes avec ces outils de compilation semi-automatiques, nous ouvrons une boîte noire du compilateur : celle de la plateforme de compilation polyédrique.

Ces deux approches complémentaires contribuent à l'élaboration d'optimisations automatiques et semi-automatiques plus robustes. L'utilisation des versions interchangeables permet d'exécuter la meilleure optimisation dynamiquement et automatiquement. Pour les cas où le développeur désire plus de contrôle, on offre un moyen simple d'interagir avec le compilateur polyédrique. Le développeur peut alors profiter de l'analyse précise et des transformations de code agressives du compilateur.