Extending Polyhedral Techniques towards Parallel Specifications and Approximations

Alexandre Isoard

Numéro National de Thèse : 2016LYSEN011

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale N° 512

École Doctorale en Informatique et Mathématiques de Lyon

Discipline : Informatique

Soutenue publiquement le 05/07/2016, par :

Alexandre ISOARD

Extending Polyhedral Techniques towards Parallel Specifications and Approximations

Extension des techniques polyédriques vers les spécifications parallèles et les approximations

---

Devant le jury composé de :

Albert	Cohen	*Rapporteur*
J. (Ram)	Ramanujam	*Rapporteur*
Samuel	Bayliss	*Examinateur*
François	Irigoin	*Examinateur*
Uday	Reddy Bondhugula	*Examinateur*
Alain	Darte	*Directeur de thèse*

Remerciements

Je tiens à remercier chaleureusement avant tout Alain Darte qui m'a encadré pendant toute cette thèse avec le sérieux d'un directeur, l'enthousiasme d'un collègue, la complicité d'un ami et l'affectif d'un grand frère, et sans qui cette thèse n'aurait jamais pu avoir lieu. Je retiendrai particulièrement son intégrité et sa rigueur scientifique qui font de ses critiques des arguments précis et décisifs qui ont su susciter d'innombrables fois une ingéniosité bienvenue. Je n'aurais jamais pu progresser aussi vite sans nos discussions enflammées au coin d'un tableau, triant entre mes balourdises et mes avancées subtiles. Je crois d'ailleurs que beaucoup des idées novatrices de cette thèse sont issues de nos nombreux *quiproquos* fortuits mais dont la résolution a souvent été source de découvertes.

Je tiens ensuite à remercier mes rapporteurs et plus généralement tous les membres de mon jury : merci Albert, Ram, Samuel, François et Uday d'avoir accepté de voyager jusque là et de relire et juger de mon travail. Je tiens aussi à remercier Paul Feautrier dont l'expertise sans égale du monde polyédrique nous a été d'un recours irremplaçable. De même, mes discussions avec Tomofumi Yuki ont été d'une aide incomparable et nombre de mes présentations ont gagné en clarté grâce à lui.

Je n'oublie pas les personnes qui ont été d'une aide indirecte mais tout autant nécessaire pour ce travail, à savoir : mes deux colocataires, François Gindraud et Clément Lagisquet, dont l'aide logistique et la qualité des repas et des discussions partagés ont fait de ces quelques années passées ensemble un régal ; Maroua Maalej, ma collègue de bureau préférée, mais aussi une amie qui a toujours été là pour me motiver, me donner conseil, et me faire goûter les spécialités tunisiennes, autant culinaires que linguistiques et spirituelles ; Laure Gonnord qui a été d'un support sans faille pour un peu tout mais surtout aussi, avec Nicolas Louvet, pleine d'idées pour l'enseignement. Et je tiens aussi à remercier nos super assistantes, infailliblement de bonne humeur et toujours d'un grand secours : Laetitia Lecot, Evelyne Blesle et Chiraz Benamor.

J'ai beaucoup apprécié mes interactions scientifiques avec nombre de mes collègues au LIP mais aussi plus loin. Je pense particulièrement aux nombreux doctorants et stagiaires, Lucie Martinet, Guillaume Iooss, Aurélien Cavelan, Oguz Kaya, Guillaume Aupy, Aurélie Lagoutte, Julien Herrmann, Bertrand Simon, Loïc Pottier, Yannick Leo, Romain

Labolle... mais aussi aux nombreux chercheurs, Thierry Dumont, Violaine Louvet, Sanjay Rajopadhye, P. Sadayappan, Sven Verdoolaege, Tobias Grosser, Louis-Noël Pouchet, Béatrice Creusillet, Ronan Keryell, Fabrice Rastello, Christophe Alias... Malheureusement, ces listes seraient bien trop longues pour citer tout le monde !

Et bien sûr, je remercie ma famille, qui a toujours été à mes côtés et ce, depuis bien longtemps ; elle, tient la place la plus au chaud.

Et pour finir, merci à Stephen Neuendorffer de m'avoir offert l'opportunité, décisive pour mon futur, de travailler au sein de Xilinx.

Résumé

Guidé par le double objectif de performance et d'énergie, les infrastructures de calcul d'aujourd'hui évoluent vers des architectures d'une complexité croissante, incluant des organisations sophistiquées de la mémoire et l'utilisation d'accélérateurs matériels. Elles exigent que l'utilisateur ou le compilateur soient capables d'extraire le parallélisme, d'optimiser la localité et d'expliciter les mouvements de données. Cette thèse, motivée par le problème pratique du transfert automatique de noyaux de calcul vers des accélérateurs tels que GPUs ou FPGAs, a également eu comme objectif principal d'étendre les techniques polyédriques (adaptées à la manipulation de noyaux de calcul à base de boucles et de tableaux) dans les multiples directions nécessaires pour répondre à un tel problème : paramètres, approximations, parallélisme.

Notre premier résultat est une analyse (exacte ou avec approximation) des ensembles de données à copier vers et depuis un accélérateur pour le tuilage paramétrique quand un noyau est transféré tuile par tuile, éventuellement de manière pipelinée, et en exploitant la réutilisation des données entre tuiles. Notre deuxième résultat, nécessaire pour être en mesure d'allouer les tableaux locaux induits par ces mouvements de données, est de généraliser le concept et la construction des conflits entre éléments d'un tableau aux spécifications parallèles, en particulier aux ordres partiels (par opposition à un ordre total pour un programme séquentiel) qui peuvent prendre en compte certains constructeurs parallèles de langages tels qu'OpenMP ou X10. Notre troisième résultat est, sur la base de cette analyse, de généraliser la contraction de tableaux à base de treillis (réseaux euclidiens), précédemment limitée à un ensemble d'éléments en conflits convexe, au cas où cet ensemble est décrit comme une union d'ensembles convexes, une situation courante lors de l'utilisation de tuilage. Nous combinons tous ces résultats dans une proposition pour la génération automatique de code pour GPUs exploitant leurs différents niveaux de mémoire et de parallélisme. Nous pensons que nos différents résultats contribuent également à l'extension des techniques polyédriques, à l'analyse des programmes parallèles et à la compilation vers d'autres accélérateurs tels que les FPGAs.

Table des matières

Abstract

Guided by both performance and power objectives, today's computing infrastructures evolve toward architectures with an increasing complexity, including sophisticated memory organizations and the use of hardware accelerators. They require the user or the compiler to be able to perform parallelism extraction, locality optimization, and explicit data movements. This thesis, motivated by the practical problem of the automatic offloading of computational kernels on accelerators such as GPU or FPGA, had also as primary objective to extend polyhedral techniques (good for handling loop and array-based kernels) in the multiple directions required to address such a problem: parameters, approximations, parallelism.

Our first result is an analysis (exact or with approximations) of the necessary copy-in and copy-out data for parametric tiling when a kernel is offloaded tile by tile, possibly in a pipelined fashion, and data reuse between tiles is exploited. Our second result, required to be able to allocate local arrays induced by data movements, is to generalize the concept and construction of conflicting array elements to parallel specifications, in particular partial orders (and not a total order as for a sequential program) that can capture parallel constructs of languages such as OpenMP or X10. Our third result is, based on this analysis, to generalize lattice-based array contraction, previously restricted to a convex set of conflicting elements, to the case where this set is described as a union of convex sets, a common situation when using tiling. We combine all these results into a proposal for the automatic code generation for GPUs exploiting their different levels of memory and parallelism. We believe our different results also contribute to the extension of polyhedral techniques, the analysis of parallel programs, and the compilation towards other accelerators such as FPGAs.

Contents

Introduction

À notre époque moderne, les ordinateurs sont omniprésents. Ils ont des applications dans des domaines aussi divers que la médecine, la physique, les mathématiques, l'économie, ainsi que la photographie, le cinéma, la musique, et même la littérature. Ils promettent toujours plus de performances pour une consommation moindre. Cependant, il devient de plus en plus difficile d'utiliser efficacement cette puissance de calcul. Pour comprendre pourquoi, nous avons besoin d'un peu de contexte.

À l'heure actuelle, le parallélisme et l'efficacité énergétique font partie des défis majeurs. Alors que la fabrication des microprocesseurs suit grossièrement la loi de Moore, accumulant toujours plus de transistors sur une même puce, les avantages jadis offerts par ce rétrécissement sont devenus de plus en plus ténus. Jusqu'aux environs de la dernière décennie, des transistors plus petits demandaient une énergie de commutation plus faible, tout en offrant une vitesse plus élevée, et pour une tension plus faible. Dans l'ensemble, ceci conduisait à une fréquence plus élevée et une meilleure performance globale sans augmentation significative de la densité d'énergie et des exigences en refroidissement, suivant ainsi le principe de réduction de Dennard [34, 69]. Mais, comme on a pu le voir sur le marché grand public, la fréquence d'horloge n'a pas augmenté depuis 2006, où elle semble plafonner aux alentours de 3,5 à 4 GHz.

En pratique, parce que la tension de fonctionnement approche la tension de seuil intrinsèque au silicium et que les courants de fuite augmentent avec des transistors si petits, les effets quantiques ont mis un terme au principe de réduction de Dennard, limitant sévèrement toute possibilité d'amélioration significative de la fréquence d'horloge. Pour compenser cette perte, les fabricants de microprocesseurs se sont dirigés vers plus de parallélisme. En effet, puisqu'on a la possibilité de disposer de plus en plus de transistors, en augmentant le nombre d'opérations pouvant être faites en parallèle par cycle, on peut espérer augmenter d'autant les performances et ce, même à fréquence d'horloge constante. Le parallélisme a cependant un coût qui est que les programmes doivent être, de fait, conçus de l'application jusqu'à la machine avec, à l'esprit, parallélisme et concurrence.

Sur un aspect différent mais non moins important, la mémoire est devenue de plus

en plus problématique. Alors que les processeurs et la mémoire ont co-évolués vers un débit plus élevé, ils n'ont pas progressé au même rythme. Il est bien connu qu'un écart de performance s'est creusé entre le processeur et la mémoire, le débit du premier s'étant amélioré plus rapidement que le débit et la latence de la seconde. Notamment, plus grande est la mémoire et plus loin elle est du processeur, plus il faut de temps pour accéder à son contenu. Ceci a motivé l'utilisation d'une hiérarchie mémoire, où une petite mémoire embarquée sur la puce met « en cache » une plus grande mémoire externe, et ainsi de suite. De cette façon, des données qui devaient normalement être lues depuis la mémoire globale peuvent être récupérées plus rapidement depuis la mémoire locale, à la condition qu'elles y soient déjà. La mise en cache vient donc, elle aussi, avec un coût, qui est que les programmes doivent être conçus cette fois-ci avec, à l'esprit, la localité des données et le « pipelining ».

Ceci nous amène à notre problème actuel : comment programmer efficacement ces machines ? En effet, le parallélisme se trouve généralement dans des algorithmes qui travaillent de façon aussi individuelle que possible en termes d'accès mémoire, où deux opérations différentes manipulent des données aussi espacées que possible. D'un autre côté, la localité est, elle, fournie par des algorithmes qui travaillent sur leurs données de façon aussi compacte que possible, où deux opérations différentes manipulent des données aussi proche que possible. Pour concilier ces deux objectifs apparemment antinomiques, il nous faut regarder au plus près de l'architecture, où les concepts tels que ligne de cache et mémoire partagée aident à produire des programmes à la fois parallèles et soucieux de la localité. Mais cela signifie aussi que les algorithmes deviennent de plus en plus dépendants des architectures sous-jacentes à un moment où les langages s'en éloignent pour offrir plus de flexibilité, portabilité et simplicité d'utilisation.

Cela signifie que le compilateur pour la partie la plus « statique » (et l'environnement d'exécution et système d'exploitation pour la partie la plus « dynamique ») est aujourd'hui la pièce centrale du puzzle. En effet, un compilateur idéal transformerait un morceau de code portable, écrit dans un langage de programmation de haut niveau avec principalement la correction de l'algorithme à l'esprit, en un fichier binaire adapté à l'architecture, en se chargeant donc, entre autres, des considérations de localité et de parallélisme de manière entièrement automatique. A cet égard, bien qu'imparfaits dans la pratique, les compilateurs sont, lentement mais sûrement, de plus en plus à même de gérer la grande diversité des problèmes posés par la parallélisation et la vectorisation automatiques.

Le développement de tels compilateurs est difficile car ils ont une forte contrainte à respecter : peu importe la complexité du programme source, le code produit devra calculer le même résultat que l'original. Pour se simplifier la tâche, les compilateurs se concentrent

sur des optimisations à petite échelle où des propriétés locales garantissent la correction de la transformation. Cependant, optimiser pour le parallélisme et la localité nécessite des analyses et des transformations de code de plus grande envergure, mélangeant potentiellement plusieurs boucles ou fonctions, transformations auxquelles les compilateurs ne se risquent que très rarement. Pour aller dans ce sens, un important cadre d'étude, offrant à la fois rigueur mathématique et expressivité, est fourni par les *analyses et optimisations polyédriques*. Construit sur la base de l'arithmétique des inégalités affines, ce « modèle » de programmes et de transformations permet de décrire des morceaux de code avec une précision suffisante pour garantir la correction des transformations et une flexibilité assez grande pour représenter entièrement la plupart des transformations de programmes les plus courantes au sein d'un cadre commun.

Néanmoins, le « modèle polyédrique » central a de fortes limitations. Dans sa forme la plus simple, il nécessite une description exacte des calculs et des accès aux données, un ordonnancement séquentiel (total) des calculs, et, par nature, uniquement des analyses et optimisations qui peuvent être décrites par des inégalités affines portant sur les variables du programme, les paramètres, les accès mémoire, ou les spécifications des ordonnancements. Le but et la portée de cette thèse étaient d'étendre (c'est-à-dire de développer de nouvelles techniques) et d'élargir (c'est-à-dire de répondre à de nouvelles applications et problèmes) les techniques polyédriques vers de nouvelles directions, en particulier le tuilage paramétrique (qui a souvent été considéré comme un problème quadratique), les spécifications parallèles (par exemple, le pipelining et le traitement des ordres partiels), et les approximations (pas juste une description exacte des informations pertinentes).

Avant de présenter l'agencement de cette thèse, évoquons un peu l'histoire cachée qui a conduit aux travaux décrits dans ce manuscrit. Notre point de départ était d'étendre la thèse d'Alexandru Plesco [60], dans laquelle avait été conçue une méthode pour le transfert sur FPGA d'un noyau (petit morceau de code) par blocs de calcul, grâce au tuilage de boucles. Notre objectif était de mieux comprendre la réutilisation de données inter-tuiles et les mouvements de données associés au transfert de noyau, dans un cadre plus conceptuel et plus général, car ceux-ci apparaissent de plus en plus importants à optimiser, non seulement pour les FPGAs mais aussi pour les multicœurs et les GPUs. C'est, par exemple, la motivation première d'OpenAcc [58], même si, en l'occurrence, l'utilisateur est toujours responsable de l'orchestration de ces mouvements de données. De même, les transferts de données dans PPCG [75], un compilateur polyédrique pour GPU, étaient assez similaires, en particulier pour les mouvements de la mémoire globale à la mémoire partagée. Néanmoins, dans PPCG, ainsi que dans les travaux d'A. Plesco, les tailles de tuiles doivent être fixées avant la compilation, ce qui rend difficile la modélisation

statique des performances et de l'influence du choix des tailles de tuile. Pour le cas des FPGAs, ce travail était même d'autant plus pénible que la technique, mise en œuvre en amont d'un outil de synthèse de haut niveau (HLS), n'était que semi-automatique et nécessitait des modifications manuelles du code ainsi généré. Pour évaluer les performances des différentes tailles de tuiles, il était donc nécessaire d'effectuer, manuellement, toutes ces modifications, pour chaque taille de tuiles !

Nous avons donc d'abord travaillé sur la façon de rendre la technique précédente **paramétrique en la taille des tuiles**, avec l'objectif double de rendre la génération de code plus générique et de permettre, potentiellement, de concevoir des modèles de sélection de taille de tuiles. Nous avons également considéré l'impact des **approximations** sur l'analyse et les optimisations associées. Il s'est avéré que, lorsque les tuiles (contenant éventuellement du parallélisme) sont exécutées en séquence (le long des axes qui les définissent), c'est-à-dire d'une façon localement parallèle mais globalement séquentielle (LPGS), le problème peut être résolu de manière entièrement paramétrique en dépit de son caractère intrinsèquement quadratique.

Nous souhaitions ensuite appliquer les techniques standard de contraction mémoire pour définir l'allocation des données transférées en mémoire locale. Ces techniques requièrent une analyse de la durée de vie des éléments de tableaux (c'est-à-dire, savoir quand un élément est « mort » et son emplacement réutilisable) et, plus précisément, une analyse d'interférences qui décrit les éléments d'un tableau qui peuvent partager le même emplacement mémoire. Cependant, pour superposer communications et calculs, nous utilisions une spécification pipelinée pour exprimer l'ordre des chargements, déchargements, et tâches de calcul, donc une spécification exprimant un certain parallélisme. Il nous est alors apparu que, même sous cette forme restreinte de parallélisme, certaines propriétés intuitives, vraies pour des calculs totalement ordonnés, ne pouvaient plus s'appliquer. Ceci nous a amenés à revoir et à étendre les techniques polyédriques standard pour l'**analyse des durées de vie** et la **construction des interférences** dans le cas d'**ordonnancements parallèles**.

Enfin, en utilisant cette analyse, il restait à appliquer les techniques standard de contraction de tableaux pour allouer les données transférées dans la mémoire locale. Pour beaucoup d'exemples, une technique à base de modulos successifs [54] était suffisante pour obtenir une bonne allocation. Cependant, lors de discussions en marge du colloque IMPACT'14, Uday Bondhugula et son élève S. G. Bhaskaracharya nous ont montré quelques exemples tuilés où la théorie d'allocation de mémoire basée sur les treillis (réseaux euclidiens) [32] nécessitait une sélection adéquate de base pour produire une bonne allocation. Nous avons alors travaillé indépendamment d'eux, et même en concurrence avec eux, sur

le problème de l'**extension de l'allocation mémoire basée sur les treillis** au cas où les interférences sont décrites par une **union non-convexe de polyèdres**, un cas qui arrive en effet fréquemment dans les situations parallèles et/ou tuilées. Ceci nous a conduits à une solution différente, avec quelques découvertes communes, comme le fait que la recherche de directions d'allocation est similaire à la recherche de directions d'ordonnancement, mais avec des contraintes (d'interférence au lieu de dépendance) qui sont non orientées.

Cette thèse raconte les détails de cette histoire. Elle est organisée de la façon suivante :

Contexte et travaux connexes où nous dressons une esquisse des architectures matérielles actuelles et donnons une vue générale de leurs capacités, de leurs limites, et des considérations associées. Nous passons aussi brièvement en revue ce qu'est la représentation polyédrique et quelques applications courantes auxquelles nous nous référerons dans la suite de la thèse.

Réutilisation inter-tuiles pour le tuilage paramétrique où nous motivons, décrivons et traitons le tuilage avec des tailles de tuiles paramétriques et réutilisation des données inter-tuiles, c'est-à-dire comment exploiter la réutilisation des données non seulement dans une tuile, mais aussi entre des tuiles successives et ce, d'une manière paramétrique. L'ordonnancement parallèle spécifique que nous avons choisi pour orchestrer les communications et les calculs, une forme de pipeline logiciel des tuiles, est ce qui a motivé notre analyse des ordonnancements parallèles décrits par une relation « happens-before » (ordre de précédence dans toute exécution).

Analyse de durée de vie pour spécifications parallèles où nous décrivons comment traiter de programmes parallèles dont la liberté d'ordonnancement sous-jacente (pouvant correspondre à un ordre partiel) est capturée par une relation « happens-before ». Nous montrons comment représenter les formes classiques de parallélisme et comment reconstruire des analyses polyédriques sur une telle description, comme l'analyse de dépendance de données, de durée de vie, ou de conflit mémoire.

Allocation mémoire où nous étendons l'allocation mémoire à base de treillis par un mécanisme permettant de choisir des directions de réutilisation, de manière à contracter des tableaux intermédiaires à partir d'une description générique des différences entre conflits mémoire. Combinée avec l'analyse précédente, cette technique devrait permettre la contraction de tableaux pour de nombreux langages exprimant des comportements parallèles.

Transfert de noyaux de calcul où nous combinons tous les chapitres précédents dans le but de produire du code optimisé pour CPU et GPU, et peut-être adapté également aux FPGAs. Nous montrons les avantages de l'analyse paramétrique, qui devrait aider à résoudre le problème de la sélection de taille de tuiles pour une architecture spécifique.

Nous concluons ce manuscrit par une synthèse de notre travail et quelques perspectives.

Introduction

In our modern life, computers are ubiquitous. They have applications in domains as diverse as medicine, physics, mathematics, economics, as well as photography, cinema, music, and even literature. They promise forevermore increasing performance for less energy. But it is becoming more and more difficult to efficiently use this processing power. To understand why, we need a little bit of context.

Today, some major challenges are parallelism and energy efficiency. While microprocessor manufacture roughly follows Moore's law by packing forever more and more transistors on the same die, the benefits associated with the shrinkage have grown increasingly tenuous. Until the last decade, smaller transistors meant smaller energy to switch them, higher switching speed, and lower voltage. All in all, this provided higher frequency and better global performance without significant increase in energy density or cooling requirements, following the scaling principles known as Dennard scaling [34, 69]. But as can be seen in the consumer market, clock frequency increased only until around 2006, where it seemingly got stuck around 3.5 to 4 GHz.

Actually, as operating voltage approached silicon intrinsic threshold voltage, and smaller transistor sizes mean bigger leakage current, quantum effects put Dennard scaling on hold severely limiting further significant increase in clock frequency. To compensate for this loss, microprocessor manufacturers focused on parallelism. Indeed, an increase in the number of parallel operations per cycle will potentially increase performance even at constant clock rate, and the availability of more and more transistors makes it practicable. Parallelism comes at a cost however, which is that programs need to be designed all the way down with parallelism and concurrency in mind.

On a different but not less important aspect, memory has become more and more problematic. While processors and memory co-evolved towards higher throughput, they did not do so at the same rate. There is a so called processor-memory performance gap, where the throughput of the former improved faster than the throughput and latency of the latter. Notably, the bigger is the memory and the further it is from the processor, the longer it takes to access its content. This motivated the use of memory hierarchy, where a small on-chip memory caches a bigger off-chip memory and so on. In this way,

data that would normally be read from global memory could be cheaply and quickly read from local memory at the condition that it is already there. Caching thus also came at a cost, which is that programs need to be conceived all the way down with data locality and pipelining in mind.

This brings us to our current problem: how to efficiently program those machines? Indeed, parallelism is usually offered by algorithms that work in a manner that is as separable as possible in terms of data accesses, where two operations manipulate data as far as possible. On the other hand, locality is offered by algorithms that work on data in a manner that is as packed as possible, where two operations manipulate data that are as close as possible. To conciliate these two seemingly antipodal objectives, a closer look at the architectures is needed, where concepts such as cache lines and shared memory help producing parallelism and locality-aware programs. But this also means that algorithms need to be more and more dependent on the underlying architectures at a time where languages are evolving further apart for more flexibility, portability, and ease of use.

This means that compilers for the more "static" part (and runtimes/operating systems for the more "dynamic" part) are nowadays the central piece of the puzzle. Indeed, an ideal compiler would transform a portable piece of code, written by the user in a high-level programming language and with mostly correctness in mind, into a binary form tailored to the architecture, handling, among others, locality and parallelism considerations in an automated way. In this respect, although imperfect in practice, compilers are slowly but surely becoming better at handling the wide diversity of the problems posed by automatic parallelization and vectorization.

Developing such compilers is difficult as they have a strong constraint to respect: no matter how complex the input program is, the produced code should compute the same result as the original. To simplify the task, compilers focus on small scale optimizations, where local properties guarantee the correctness of code transformations. But parallelism and locality optimizations require a wider analysis and manipulation of the code, potentially mixing multiple loops or functions, transformations that compilers rarely risk themselves into. To go in this direction, an important tool that provides both the soundness of mathematics and a quite wide expressiveness is the framework of *polyhedral code analyses and optimizations*. Built on top of the arithmetic of affine inequalities, this "model" of programs and transformations is able to describe pieces of code with enough precision to guarantee correct transformations and enough flexibility to enable most common program transformations to be entirely described into the same framework.

Nevertheless, the central "polyhedral model" has strong limitations. In its simplest form, it requires an exact description of computations and data accesses, a sequential

(total) order of computations, and, by nature, only analyses and optimizations that can be addressed with affine inequalities of program variables, parameters, memory accesses, specifications of orders. The goal and the scope of this thesis were to extend (i.e., develop new techniques) and expand (i.e., address new applications and problems) polyhedral techniques towards new directions, in particular parametric tiling (which was often seen as a quadratic problem), parallel specifications (e.g., pipelining and handling of partial orders), and approximations (not just an exact description of all relevant information).

Before giving the general organization of the thesis, let us tell the hidden story of the work described in this document. Our starting point was to extend the thesis of Alexandru Plesco [60], in which a method for offloading to FPGA a kernel of blocked computations, thanks to loop tiling, was designed. Our goal was to better understand inter-tile data reuse and data movements for kernel offloading in a more conceptual and general setting, as they appeared more and more relevant, not just for FPGAs, but also for multicores and GPUs. For example, this is what OpenAcc [58] was designed for, even if the user is still responsible for orchestrating such data movements. Data movements in PPCG [75], a polyhedral compiler for GPU, were also quite similar, especially for movements from the global to the shared memory. In PPCG as well as in A. Plesco work, tile sizes were however fixed before compilation, which makes tile size selection and static performance models more difficult to design. For the work on FPGA, this was even more painful as the technique, implemented on top of a HLS (high level synthesis) tool, was only semi-automatic and required some manual modifications of the generated code. To evaluate the performance of different tile sizes, it thus required manual changes for each tile size!

We thus first worked on how to make the previous technique **parametric with respect to tile sizes**, in the double objective of making code generation more generic and of possibly designing tile size selection models. We also considered how **approximations** impact the analyses and related optimizations. It turned out that when tiles (possibly containing parallelism) are to be offloaded in sequence (along the axes that define them), in a LPGS (locally parallel globally sequential) fashion, the parametric problem can be solved, despite its intrinsic quadratic nature.

We then wanted to apply standard memory contraction techniques so as to map the offloaded data to local memory. These require an analysis of the liveness of array elements (i.e., when an element is dead and its location can be reused) and, more precisely, an interference analysis specifying which array elements can share the same memory location. However, to overlap communications and computations, we used a pipelined specification of the schedule of loads, stores, and computation tasks, i.e., a specification expressing some parallelism. Even in this restricted form of parallelism, it appeared that some

intuitive properties true for a total order of computations do not apply anymore. This led us to revisit and extend standard polyhedral techniques for **liveness analysis** and the **construction of interferences** to the case of **parallel specifications**.

Finally, using this analysis, it remained to use standard array contraction technique to map offloaded data to local memory. For many examples, a basic successive modulo technique [54] was enough to get a good allocation. However, through side discussions of the IMPACT'14 workshop, Uday Bondhugula and his student S. G. Bhaskaracharya showed us some tiling examples where the theory of lattice-based memory allocation [32] required an adequate selection of basis to find a good allocation. Independently of them, even in competition with them, we thus worked on the problem of **extending lattice-based memory allocation** to a description of interferences as a **non-convex union of polyhedra**, a case that indeed happened more often in parallel specifications and tiling situations. And we ended up with a different solution, with some common discovery such as the fact that looking for mapping directions is almost the same as looking for scheduling directions, except that constraints (interferences instead of dependences) are undirected.

This thesis tells the details of this story. It is organized into the following chapters:

Background and related work where we draw a sketch of current hardware architectures and give a high-level overview of their capabilities, their limitations, and the associated considerations. We also briefly survey the polyhedral representation and some common applications to which we will refer to in the remaining of the thesis.

Inter-tile reuse for parametric tiling where we motivate, describe, and address tiling with parametric tile sizes with inter-tile data reuse, i.e., how to exploit data reuse not only in a tile, but between successive tiles, in a parametric fashion. The specific parallel schedule we choose to orchestrate communications and computations, a form of software pipelining of tiles, is what motivated our analysis of parallel schedules based on an happens-before relation.

Liveness analysis over parallel specifications where we describe how to handle parallel programs whose underlying schedule freedom (possibly corresponding to a partial order) is captured by an happens-before relation. We show how to represent classic forms of parallelism and rebuild polyhedral analyses over such a description, such as data dependences, liveness, memory conflict analysis.

Memory allocation where we extend lattice-based memory allocation with a mechanism to choose reuse directions, so as to contract intermediate arrays using a generic description of memory conflict differences. Combined with the previous analysis,

this technique should enable array contraction for many languages expressing parallel behaviors.

Kernel offloading where we combine all of the previous chapters in order to produce optimized code for CPU and GPU, and possibly suitable for FPGA too. We show the advantages of parametric analysis, which should help addressing the problem of tile sizes selection for a specific architecture.

We conclude this manuscript with a summary of our work and some perspectives.

Chapter 1

Background and Related Work

Summary

In this chapter, we lay the theoretical foundations used throughout this thesis. Section 1.1 provides an abstract view of current architectures. It explains some of the similarities and differences between CPU and GPU, and provides the different technical specificities of each one such as superscalar behavior, vector operations, warp scheduling, cache hierarchy, scratchpad programming. Section 1.2 describes the polyhedral model and some associated techniques. It provides the mathematical framework that will be extensively used in this thesis. This includes iteration domain, access functions, partial orders, scheduling functions, and their associated transformations.

1.1 Processor Architecture and Processing Model

> Any discrepancy between model
> and reality is most certainly
> due to reality being inaccurate.
>
> Douglas Adams

The purpose of this section is to provide the reader with the necessary knowledge about modern hardware architectures. It is not meant to be an exact depiction of all architectures, but to provide the reader with enough information so as to explain the motivation and reasoning behind certain choices, notably those involving caching and prefetching mechanisms, vectorized instructions, and multicore considerations. These features are common, although in different forms, to both CPU and GPU, and might apply to other architectures. FPGAs have a different processing model, closer to logic circuit, but nevertheless share some similarities when programmed through HLS (High Level Synthesis) tools, and will be only succinctly discussed as they need special considerations.

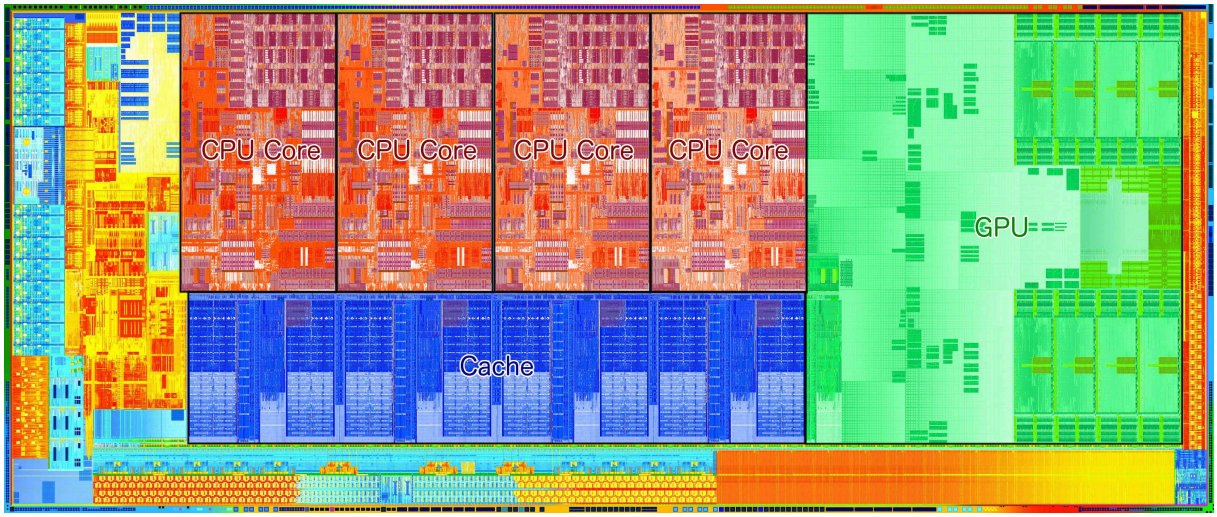


Figure 1.1 – Die shot of an Intel 3rd Generation Core: Intel i7 3770
Displayed: 4 x86_64 cores, L3 Cache of 8 MB, HD Graphics 4000 (16 execution units).

1.1.1 CPU

Figure 1.1 is a picture of the silicon die of a modern quad-core Intel processor. The parts that are of interest to us are the CPU cores and the cache. The integrated GPU can mostly be viewed as a separate chip and will be discussed in Section 1.1.2.

A CPU core is where most of the computation takes place. It reads instructions from the main memory, decode them, reorder them (if useful), and then execute them. For a given instruction, these steps happen in sequential order, but as multiple instructions are processed, current architectures pipeline the steps of different instructions to offer a higher frequency of execution. For example, while some instructions such as multiplications can take more than one cycle to compute, the pipelined computation unit can execute an instruction at every cycle assuming sufficiently many parallel instructions (not writing on registers of others) are provided. Holes in this pipeline (when other instructions cannot be executed due to dependence or resource constraints) are a possible source of inefficiency and have to be mitigated by increasing the instruction-level parallelism at compile time if possible.

The cache is here to mitigate another source of inefficiency, which is due to memory accesses. While compute instructions take one cycle to execute, they most of the time require the data to be in the register file. As there are relatively few registers compared to the main memory capacity, it is sometimes necessary to transfer data back and forth between the memory and the registers. The problem is that the main memory accesses are really slow as can be seen on Table 1.1. To alleviate the problem, data accesses are done through a cache that copies an area around the accessed data (a cache line) so as

Table 1.1 – Performance for Intel Core i7 Processor [55], all cache line are 64 bytes in size.

Cache hit	Size	Associativity	Latency
L1 instruction cache	32 KB	4 ways	4 cycles
L1 data cache	32 KB	8 ways	4 cycles
L2 cache	256 KB	8 ways	10 cycles
L3 cache (unshared)	8 MB	16 ways	$\sim$ 40 cycles
L3 cache (shared)	8 MB	16 ways	$\sim$ 65 cycles
L3 cache (modified)	8 MB	16 ways	$\sim$ 75 cycles
Main memory			60–100 ns*

* Results in ns do not scale with the frequency. At 3 GHz there are 3 cycles per ns.

to make the next accesses to the same data, or data next to it, faster. The cache is also capable of prefetching data that it expects to be needed in the future instructions, so as to not even pay the latency for the first access. It does so by analyzing previous accesses to predict the future accesses (usually assuming a constant stride).

To understand how this influences the performance of algorithms, let us consider the ubiquitous matrix multiplication:

```
for(int i = 0; i < n; ++i)
    for(int j = 0; j < n; ++j) {
S:      C[i][j] = 0;
        for(int k = 0; k < n; ++k)
T:          C[i][j] += A[i][k] * B[k][j];
    }
```

where A and B are in row-major order (consecutive cells on a row are consecutive in memory). This code seems good, as it shows ideal locality (here temporal locality, i.e., successive reuse of the same data) for C accesses (the innermost loop iterates on the same element) and good locality (here spatial locality, i.e., successive use of data close in memory w.r.t. the cache) for A accesses. While B shows poor locality (access in column-major order), the symmetry of the situation (one matrix is accessed in rows and the other in columns), we could think that this is the best we can do.

In reality, the transformed code given hereafter shows better performance. While locality on C accesses has been downgraded to a spatial locality, A accesses got upgraded to a temporal locality and, more importantly, the ones on B now display spatial locality too, which means that most data accesses will now successfully find the data in the cache.

```
for(int i = 0; i < n; ++i) {
    for(int j = 0; j < n; ++j)
S:      C[i][j] = 0;
    for(int k = 0; k < n; ++k)
        for(int j = 0; j < n; ++j)
T:          C[i][j] += A[i][k] * B[k][j];
}
```

A side effect of loosing temporal locality for C is that the privatization of C (computing the k loop in a scalar then only writing it on C at the end) is no longer possible. On the other hand, assuming there is no aliasing between A, B, and C (i.e., arrays are disjoint in memory), the loop on j is parallel which should enable vectorization of the code if the alignment (vectorized data accesses are required to be at addresses multiple of a specific power of two) of B and C in memory is suitable, and should also provide good instruction-level parallelism.

To finish with the CPU, as can be seen on the die shot of Figure 1.1, they harbor multiple cores and these cores are mostly autonomous and loosely coupled, unlike for GPUs as we will see later. This has advantages, as these cores can execute completely different pieces of code in parallel at different rates, but this also means that synchronizations are sometimes necessary at the software level. On our example (second version), using the parallelism on the innermost j loop to split the work among the different cores is a bad idea as the surrounding k loop is sequential. This means a potential synchronization between each iteration of the k loop which can happen at a high frequency, especially when the j loop has been well optimized with vectorization. On the other hand, the i loop is also parallel and does not require any synchronization except at the end of the whole computation. A rule of thumb is that innermost parallelism is good for vectorization, and outermost parallelism is best for multi-threading.

The cache the most visible on the die shot of Figure 1.1 is actually the last level of cache (or L3), it is shared between the cores (although each core has its privileged section). There are other cache levels as can be seen on Table 1.1 that are smaller and private to each core. Cache associativity is the number of cache line address collisions that the cache can handle before evicting one of the concerned cache lines (caches behave like hash maps where the key is the address, only a limited number of collisions are possible due to hardware constraints).

Many further considerations can affect performance, such as branch prediction, page table and address translation, but they are outside the scope of this thesis.

1.1.2 GPU

Graphical Processing Units (GPU) have been introduced to improve the performance of computations for a specific kind of programs, those typically involving image synthesis. Pushed by the increasing need of visual quality in video games, they were optimized for tasks where manipulation of a huge number (millions) of simple independent objects (triangles or pixels) were needed at a really high rate (tens of frames per seconds). Their unique parallel design made them suited for more tasks than initially envisioned, and they are now the accelerator of choice when raw performance is needed and sufficient parallelism is available.

General Purpose GPU (GPGPU) programming became so prevalent in the last decade that most consumer devices now offer some generic programming, outside the simple framework provided by fragment and geometry shaders of graphic drivers. The specificities of these massively parallel architectures required domain-specific languages. OpenCL and CUDA are the most successful ones, thanks to their compatibility with the classic C/C++, and are therefore the privileged way to exploit GPGPU capabilities. They share some common design principles that we will describe shortly. Also, as they match more or less the GPU hardware itself, this description will serve as a good abstraction model of modern GPU.

GPU are well suited for programs that offer a high degree of parallelism as they commonly provide thousands of cores. Figure 1.2 is a die shot of a GPU main microchip. It does not include the main device memory as it is usually on a separate chip on the GPU board. This GPU contains Graphic Processing Clusters (GPC), each one containing Streaming Multiprocessors (SMX or SM). These streaming multiprocessors (in the same GPC or not) are independent of each other and usually cannot be synchronized. The cache that can be seen in the figure is the last level of cache (L2) and is shared between all the SM. Figure 1.4 shows an abstract view of the hardware, the GPC are not represented as, from a programming point of view, they do not seem significant for GPGPU purposes. What is remarkable is the number of cores that a GPU provides: this model has 2880 cores. Each of these cores is fully pipelined like a CPU, but they usually run slower at around 1 GHz. They do not provide vector operations but are instead grouped by packs of 32 cores that execute, in lock step, the same instructions, albeit on different registers.

CUDA abstracts the architecture in the following way: the basic unit of computation is the thread, which corresponds to the execution of one CUDA core. Threads are implicitly grouped into warps, i.e., packs of 32 threads. A warp is therefore equivalent to a thread of vector operations with a vector width of 32. This brings limitations on the kind of codes that can be efficiently executed on a GPU. Thread divergence, where different threads

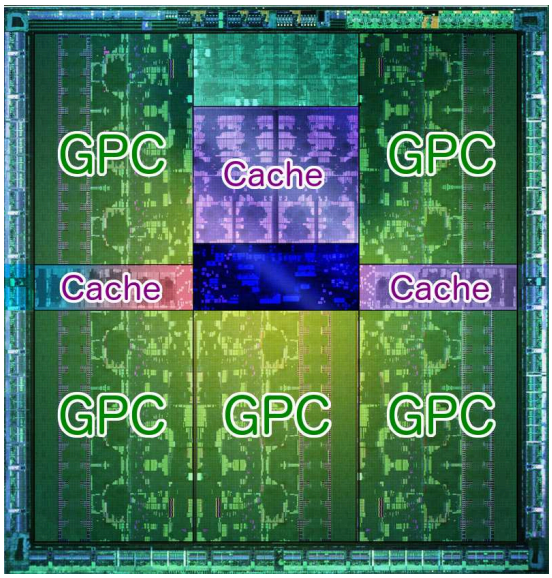


Figure 1.2 – Die shot of a NVIDIA Kepler GK110: Tesla K40
Displayed: 5 GPC (of 3 SMX each), L2 Cache of 1.5 MB (2880 CUDA cores).

Figure 1.3 – SMX Block Diagram of a NVIDIA Kepler GK110.

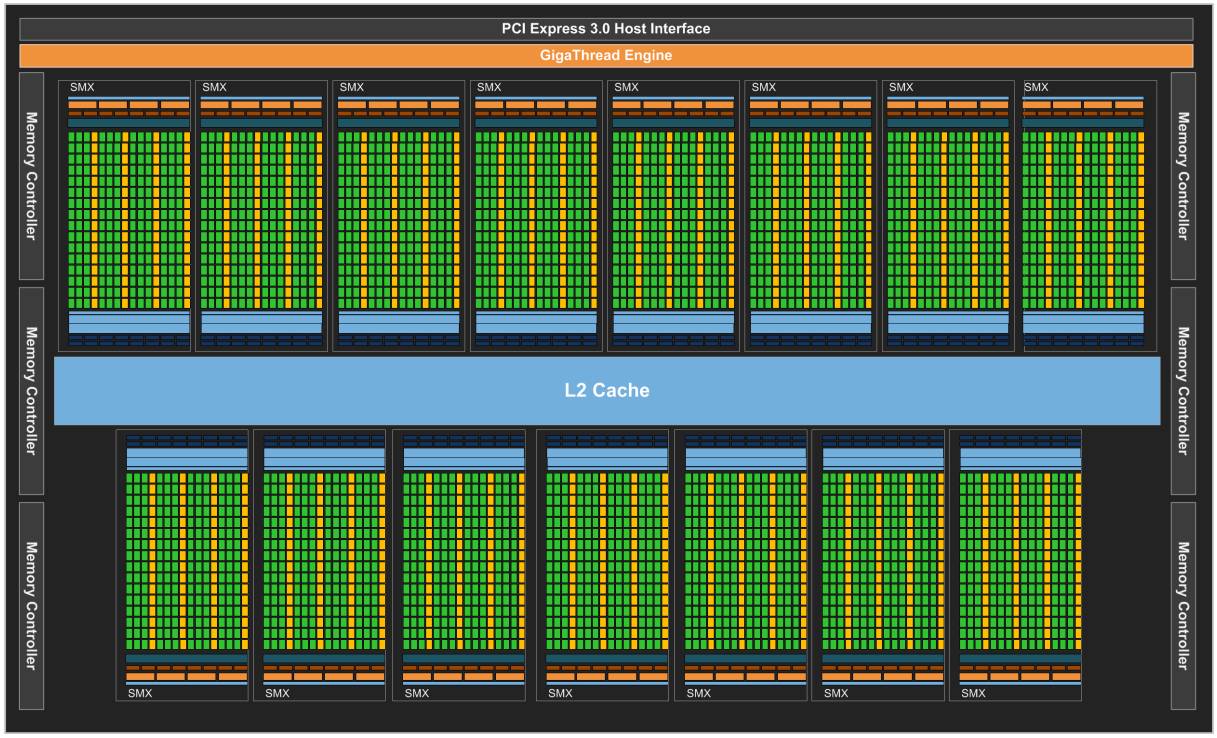


Figure 1.4 – Full Block Diagram of a NVIDIA Kepler GK110.

take different control flow paths, causes a loss of performance, as a warp can only take one path at a time. In case of thread divergence in a warp, it will successively take one path, then another, inhibiting cores that do not actually participate in each branch. This usually leads to an under-utilization of computing resources and must generally be avoided as much as possible. GPU can contain dedicated double precision units, special function units (for special mathematical function such as sinus, cosines, or logarithms), and load/store units (for main memory and shared memory transfer).

On the software side, threads or warps are grouped into blocks. Each block has its dedicated shared memory, cache, and register file. They are shared among the different threads of the block. The shared memory is fully programmable and has a higher throughput and lower latency than the device main memory. Its utilization is essential to achieve the highest performance offered by the architecture. The caches benefit from the same advantages but they are duplicating data from higher levels of cache, or from the device memory, and they cannot hold data that is not already allocated into the device memory. They are programmed by the hardware, which saves the need for explicit load and store instructions (this eases programming and also saves cycles) but this behavior is harder to predict and cannot easily be controlled. There are two different caches, one is generic and also used as an instruction cache, the other one is specialized for texture as it is a read-only memory with a good locality in both vertical and horizontal accesses. The later can be used for GPGPU purposes only on latest architectures. Registers are the most efficient memory level as read and write accesses can be made at each cycle by each CUDA core. Variables are local to a thread, and usually mapped to registers (unless spilled). Finally, threads of a same block can share registers by declaring small arrays of constant sizes, which are usually mapped to registers when possible.

On the hardware side, a block is executed on a single SM, but a SM can execute multiple blocks. The shared memory and registers of a block are allocated in the shared memory and register files of the SM. This usually dictates how many blocks can be alive at the same time in a given SM. These SM have additional limits on the number of blocks and the number of threads they can have active at the same time, which can further limit the number of blocks running concurrently on the same SM. It is usually important to maximize the use of registers as this is the memory level with the lowest latency but also the highest throughput. It is also sometimes advisable to use fewer threads per block so as to have more registers per thread, and keep some instruction-level parallelism for each thread. Also, on recent architectures, a SM can schedule multiple warps in parallel, but also dispatch multiple instructions of the same warp at the same cycle. Preserving some parallelism for instruction-level parallelism is therefore a good idea as it can be used both

to fill the pipeline of the CUDA cores and to utilize this dispatch capability.

At a GPU-wide level, computations are regrouped into kernels. Modern GPU can execute multiple kernels concurrently. They also provide copy engines that can transfer data from the main memory of the host to the main memory of the device and vice versa, concurrently. These communications happen significantly faster if the data on the host is allocated in page locked memory, that is if the memory is mapped, on the host, without standard paging mechanisms. With this, the GPU can execute the transfers with minimal search in the page table. Some GPUs provide sufficiently many copy engine to perform bidirectional communication, some can only perform one way at a time. In case of multi-GPUs, direct device-to-device communications are also possible and they operate through a dedicated faster link. All these communications and computations need to be synchronized. This is handled using streams and events. A stream is a list of kernels, transfers, and/or events that are executed sequentially. Streams are potentially executed in parallel, depending on the capabilities of the hardware, and can be synchronized by waiting for events of concurrent streams. There are some subtleties to fill these streams properly so as to obtain the desired overlap.

A kernel is composed of a grid of blocks, these blocks are parallel and **cannot** easily be synchronized with respect to each other. There is also no guarantee (and it is usually not the case) that these blocks are all live at the same time at some point. Indeed, each block is usually executed on its own SM, and each SM can only have a small number of live blocks as they reserve registers and shared memory to run. This means that attempting to execute a barrier between multiple blocks is prone to deadlocks. Barriers over the whole grid is instead implemented by splitting the computations into multiple kernels at each barrier.

All in all, this makes the GPU an architecture of choice for programs with a large amount of parallelism. The simplicity of the architecture makes it relatively easy to achieve good performance for programs that fit well into the CUDA model. Programs exhibiting a lot of irregular control are however a poor fit as they tend to create thread divergence. It is also difficult to balance the computations during the execution of a kernel and programs with regular parallelism or workload are to be preferred.

1.2 Polyhedral Techniques

The polyhedral model provides a precise symbolic representation particularly suited to describe nested loops. The name comes from the fact that it is based on the use of polyhedra in arbitrary (but finite) multidimensional spaces. The mathematics behind the

model are often complex, the intent of this section is to convey the intuition of the model itself and give an overview of the tools it provides. We will discuss its typical use in the context of analysis and optimization of imperative programs.

1.2.1 Affine Static Control Part

We are interested in pieces of codes in which nested loops intervene. Those are typical of computationally-intensive applications, such as image/sound/video manipulation but also a lot of scientific applications. Feautrier [36] showed that under some assumptions on the nested loops of the program or fragment of program, it is possible to represent its execution using polyhedra over multidimensional integer spaces. This is of particular interest because under such assumptions, the model fits in the logic of Presburger arithmetic—the first-order theory over integer numbers with equalities, inequalities, and addition (but no multiplication)—which is a decidable theory. What it means is that most of the operations we want to perform, even on a symbolic (i.e., with parameters) representation, can be done algorithmically. There is however no guarantee on the time these operations take and most known algorithms have worst-case complexity above exponential. Luckily, in our context, the formulas we manipulate are usually relatively simple, making the approach practicable on examples of moderate size.

The affine form of the formulas induces strong constraints on the kind of codes that can be handled precisely. Feautrier [36] provides a way to describe a class of programs composed exclusively of nested/successive `for` loops and `if` conditionals, in which memory accesses are array accesses (possibly multi-dimensional) that never alias each other. Additionally, the following constraints are required:

- The control shall be static: predicates, loop bounds, loop increments, and array subscripts shall only depend on literal constants or a finite number of variables that are either constant inside the whole part or iterators of enclosing loops.
- The control shall be affine: predicates, loop bounds, and array subscripts shall be multi-dimensional affine functions with regards to the variables. Loop increments shall be literal constants.

These constraints guarantee that loop bounds and memory accesses can be represented into Presburger arithmetic. We will abbreviate the concept into Static Control Parts (SCoP) as it is common in the literature, even if the name is not necessarily well chosen.

Some languages such as C can implement multi-dimensional arrays as arrays of pointers to arrays, recursively. As the polyhedral model is usually oblivious to these pointers,

provided that none of them share any cell (no aliasing), there is no particular contraindication for such structures except that this can lead to poor performance. Pointer arithmetic should be avoided, or converted to array accesses.

As an example, the following C code:

```
float A[m][n], B[n][p], C[m][p];
for(int i = 0; i < m; ++i)
    for(int j = 0; j < p; ++j) {
        C[i][j] = 0;
        for(int k = 0; k < n; ++k)
            C[i][j] += A[i][k] * B[k][j];
    }
```

is a valid SCoP. It is important to note that the following equivalent code:

```
float A[m*n], B[n*p], C[m*p];
for(int i = 0; i < m; ++i)
    for(int j = 0; j < p; ++j) {
        C[i*p+j] = 0;
        for(int k = 0; k < n; ++k)
            C[i*p+j] += A[i*n+k] * B[k*p+j];
    }
```

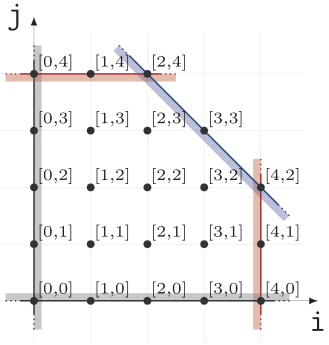
is **not** a valid SCoP, due to the polynomial access functions. There are two ways of handling such a common problem: via delinearization of the access functions into equivalent access functions of higher dimension [56, 20, 43], but this is a hard problem and the solution may not be unique, or via approximations with affine functions, but this can give poor results as it may lead to assuming potential accesses anywhere in each array.

1.2.2 Constraint Representation

There are two main representations for polyhedra: constraint vs vertex. The first one describes a polyhedron as an intersection of half-spaces, each described by a direction vector (orthogonal to the delimiting hyperplane) and a constant (which shifts the delimiting hyperplane away from the origin). This corresponds to the logical `and` of affine inequalities. The second one describes a polyhedron as the convex set delimited by its vertices (for a bounded polyhedron). Both representations have their respective advantages and drawbacks. The first one is harder to visualize, there are multiple representations for the

same polyhedron (especially when it is flat), and it is not trivial to detect if it is empty. On the other hand, it is closely related to loop bounds, and most polyhedral libraries rely on it. The second one looks more intuitive and geometric, but an infinite polyhedron requires rays (infinitely far vertices), and the number of vertices usually grows exponentially with the dimension of the space.

As we will exclusively use the first representation, an easy-to-read syntax is required. We will rely on the syntax introduced by `Omega` and its calculator [51] and later extended by `isl` [73], which is designed to be fairly readable compared to the matrix representation of constraints used internally. As an example, the following integer polyhedron[1], represented for `n` = 5 and `m` = 6 on the right (black dots), is written in `isl` syntax as follows:




```
[n,m] -> { S[i,j] : 0 <= i,j < n and i+j <= m }
```

where `n` and `m` are parameters, and this set is defined over a 2D integer space[2] named `S` and delimited by 5 constraints: $0 \leq i$, $0 \leq j$ (in black), $i < n$, $j < n$ (in red), and $i + j \leq m$ (in blue).

The strength of the polyhedral model comes from the abstraction it provides. These sets can be treated as mathematical sets from which we can compute the intersection (`and`), the union (`or`), the complement (`not`), project out some dimensions ($\exists$), and much more. Some of these operations are more or less costly in the worst case, just to mention the size of their results as shown in Table 1.2. Luckily, in practice the complexity remains acceptable. There are exceptions, for example the cross-product of unions of polyhedra usually corresponds to the worst-case for intersection, when rewritten into a union of polyhedra (disjunction of conjunctions). In general, operations that increase the number of dimensions are expensive.

Projection (elimination) of variables is defined as follows:

Definition 1 (Projection)**.** *Let S be an integer set of dimension n. The integer set P built from S by projecting out the d last dimensions is defined by:*

$$P = \left\{ \vec{x} \in \mathbb{Z}^{n-d} \;\middle|\; \exists \vec{y} \in \mathbb{Z}^d, [\vec{x}\ \ \vec{y}] \in S \right\}$$

where $[\vec{x}\ \ \vec{y}]$ is the vector obtained by concatenating $\vec{x}$ (of size $n-d$) and $\vec{y}$ (of size d).

[1] We use the term "integer polyhedron" as a shorthand for the set of integer points in a polyhedron.

[2] The dimensions of this space are named `i` and `j`, but it is only relevant for defining the constraints because space dimensions are actually identified by their position and not by their name.

Operation	Disjunctions	Conjunctions	Worst-case sizes
Intersection	$\mathcal{O}(n \times n')$	$\mathcal{O}(p + p')$	quadratic
Union	$\mathcal{O}(n + n')$	$\max(p, p')$	linear
Complement	$\mathcal{O}(p^n)$	$\mathcal{O}(n)$	exponential
Projection of d variables	$\mathcal{O}(n)$	$\mathcal{O}(p^{2^d})$	super-exponential

Table 1.2 – Worst-case sizes of the result of common operations (worst-case computational cost can be higher): n is the number of disjunctions (polyhedra) of the input and p the number of conjunctions (faces) in each disjunction.

Projecting out dimensions is one of the most powerful operations of the polyhedral model: it enables, among others, to compute a parametric emptiness test (by projecting out all dimensions except parameters) or to find the set of minima with respect to a partial order, and much more. It is a complex operation, which can in general be solved by Fourier-Motzkin elimination, or by keeping, thanks to parametric linear programming, the projected variables as existential variables expressed in terms of the remaining ones.

1.2.3 SCoP Representation

The objective is to represent SCoP symbolically using the polyhedral model. The central concept is the concept of iteration. Indeed, the strength of the polyhedral model is to reason about loops as if they were symbolically unrolled. We associate a unique identifier to each operation, which is usually the name of the instruction and a point into a space (an iteration vector) with as many dimensions than enclosing loops.

For example, the following code (polynomial product):

```
for(int k = 0; k < 2*n-1; ++k)
S:  C[k] = 0;
for(int i = 0; i < n; ++i)
    for(int j = 0; j < n; ++j)
T:      C[i+j] += A[i] * B[j];
```

can be described over the following set of iterations, called domain:

$$\text{Domain} = \{\text{S}[k] \mid 0 \leq k \leq 2n-2\} \cup \{\text{T}[i,j] \mid 0 \leq i,j < n\}$$

where S$[k]$ describes one instance of the instruction S, and T$[i,j]$ one instance of the instruction T. The data accessed at each iteration can then be described with the following

access relations:

$$\text{Read(A)} = \{\text{T}[i,j] \mapsto \text{A}[i]\} \qquad \text{Read(B)} = \{\text{T}[i,j] \mapsto \text{B}[j]\}$$
$$\text{Read(C)} = \text{Write(C)} = \{\text{S}[k] \mapsto \text{C}[i+j]\} \cup \{\text{T}[i,j] \mapsto \text{C}[i+j]\}$$

where Read is a binary relation over Domain × Array and where Array represents the domains of the array subscripts. Finally, the order of execution can be described with the following scattering function:

$$\text{Sched} = \{\text{S}[k] \mapsto [0,k,0]\} \cup \{\text{T}[i,j] \mapsto [1,i,j]\}$$

where Sched is a binary relation over Domain × Time and where Time represents a virtual multi-dimensional space over which instructions are executed in the lexicographic order (the order of the dictionary used for vectors as words), i.e., here the total order: $[0,0,0]$, $[0,1,0]$, $[0,2,0]$, …, $[1,0,0]$, $[1,0,1]$, $[1,0,2]$, …, $[1,1,0]$, …. Notice how the sequential execution of the two sets of nested loops (loop k for S, loops i and j for T) is represented with an additional dimension (the leftmost here, with value 0 for S and 1 for T). This is because a sequential execution can be seen as the unrolled version of the code:

```
for(int a = 0; a < 2; ++a)
    if(a == 0)
        for(int k = 0; k < 2*n-1; ++k)
S:          C[k] = 0;
    else
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j)
T:              C[i+j] += A[i] * B[j];
```

Another remark is that `if` conditions are not encoded in the scattering function, but instead by constraining the iteration domain. For example, the following code:

```
for(int i = 0; i < n; ++i)
    for(int j = 0; j < n; ++j)
        if(i != j)
S:          C[i][j] -= C[j][i];
        else
T:          C[i][j] = C[i][j] * 2;
```

is encoded as:

$$\begin{aligned}
\text{Domain} &= \{S[i,j] \mid 0 \leq i,j < n \wedge i \neq j\} \cup \{T[i,j] \mid 0 \leq i,j < n \wedge i = j\} \\
\text{Read} &= \{S[i,j] \mapsto C[i,j]\} \cup \{S[i,j] \mapsto C[j,i]\} \cup \{T[i,j] \mapsto C[i,j]\} \\
\text{Write} &= \{S[i,j] \mapsto C[i,j]\} \cup \{T[i,j] \mapsto C[i,j]\} \\
\text{Sched} &= \{S[i,j] \mapsto [i,j]\} \cup \{T[i,j] \mapsto [i,j]\}
\end{aligned}$$

1.2.4 Relations

As shown earlier, the polyhedral model describes some properties using relations (maps). Some polyhedral libraries provide such an abstraction and they are accompanied by useful operations such as taking the domain (project out the right hand side), taking the range (project out the left hand side), applying a map to a set (intersect the domain by the set, then take the range), and joining two maps (described below).

The last one is by far one of the most powerful operations we will use. It corresponds to the composition of relations. It is formally described by:

Definition 2 (Composition of relations)**.** *Let $S : \mathcal{L} \mapsto \mathcal{M}$ and $T : \mathcal{M} \mapsto \mathcal{R}$ be two relations. The* composition *$S.T : \mathcal{L} \mapsto \mathcal{R}$ of S and T is defined by*

$$(l,r) \in S.T \iff \exists m \in \mathcal{M}, (l,m) \in S \wedge (m,r) \in T$$

This can be seen as intersecting the constraints from both sides on the intermediate space, then projecting the intermediate variable out. This composition can be used, for example, to easily express the computation memory-based dependence analysis (and even dataflow analysis with additional set differences), or to compose schedule transformations.

1.2.5 Dependences

Most optimization techniques require a preliminary dependence analysis. One of the strengths of the polyhedral model is that it can be used to compute an exact dependence analysis in nested loops, even when the dependences are not uniform (i.e., not just translation of iteration vectors) or the loops not perfectly nested (some instructions are at different depths or even in different loops).

To illustrate this construction, we can compute and represent a memory-based flow-dependence analysis as a relation that maps an instance of an instruction, writing to a given memory cell, to another instance of (potentially another) instruction, reading to the same memory cell, and where the latter instruction is executed after the former (RaW

dependence, i.e., read after write). This analysis can easily be done using the Sched, Read, Write, and relation composition defined earlier:

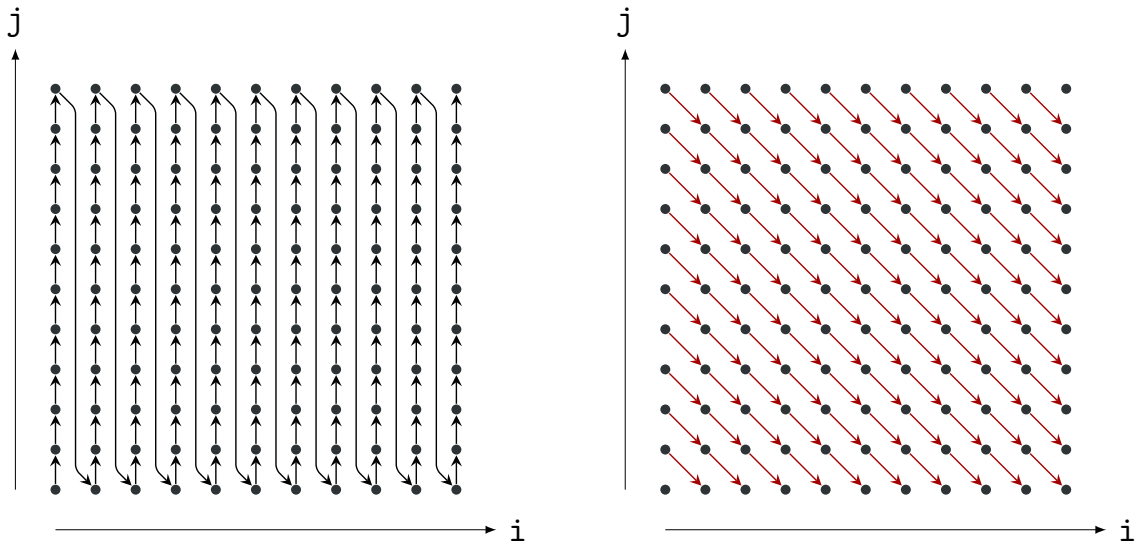
$$\text{RaW} = (\text{Read}^{-1}.\text{Write}) \cap (\text{Sched}.\text{After}.\text{Sched}^{-1})$$

where Read^{-1} (resp. Sched^{-1}) is the reverse relation (simple syntactic permutation of the domain and range of the map) of Read (resp. Sched), and After is the relation representing the lexicographic order over the Time space. In 3D, it is defined by:

$$\text{After} = \{[t_0, t_1, t_2] \mapsto [t'_0, t'_1, t'_2] \mid t_0 > t'_0 \vee (t_0 = t'_0 \wedge (t_1 > t'_1 \vee (t_1 = t'_1 \wedge t_2 > t'_2)))\}$$

The idea here is to compute the set of pairs of instruction instances where one reads and the other writes, on at least one common array cell. This is done by joining the two relations on the Array side (hence the need to reverse one of them). This produces a relation that maps two instances of instructions together when the left one reads (i.e., the map Read) a cell that the right one writes (i.e., the map Write). We then keep only the ones in the Read-after-Write order, which is done by the intersection. For that, we consider the pairs of instructions for which the left one is executed after the right one, i.e., those such that the time (given by the Sched map) at which the left one is scheduled happens after (in the lexicographic order) the time at which the right one is scheduled.

Figure 1.5a represents the `After` relation (or more exactly, its transitive reduction), and Figure 1.5b represents the exact value-based flow dependences that can be computed



(a) Sequential order (no transitive edges). (b) Exact value-based flow dependences.

Figure 1.5 – Geometric representation of the iteration space of instruction T of the polynomial product example, and related relations over it.

with polyhedral techniques, for the previous polynomial product example. We did not explain how to compute these exact flow dependences (memory-based dependences from a write to a read, with an intervening write that kills the previous value, need to be removed). Most polyhedral libraries provide built-in dependence analyses that are possibly faster than this computation because here we first enumerate all pairs of reads and writes even those in the wrong order, before removing them with an intersection. But this was just for the sake of illustration as we will make plenty use of this kind of operations in this manuscript, in particular in Chapter 3 where we generalize such dependences to partial (non-lexicographic) orders.

1.2.6 Loop Transformations

One of the strengths of the polyhedral model is its ability to describe and compose most of the usual loop transformations. Indeed, by simply applying a new schedule relation, we can produce a new loop structure, provided the desired loop structure fits the polyhedral model. Furthermore, if the transformations themselves can be described as affine relations, mapping the original time space to the new time space, then we can compose them using the join operation.

Most common loop transformations are actually affine transformations of the time space. This includes loop reversal, loop striding (multiply a time dimension by a numerical constant), loop skewing (linear combination of time dimensions), loop splitting/peeling, loop sectioning/strip-mining of constant width (splitting a time dimension into two, as computing by slices), loop fission/distribution, loop fusion/combining, loop interchange/permutation (swapping two time dimensions), etc. For example, to produce the following code (equivalent to the polynomial product):

```
for(int x = 0; x < 2*n-1; ++x) {
S:  C[x] = 0;
    for(int y = max(0, x-n+1); y <= min(n-1, x); ++y)
T:      C[x] += A[y] * B[x-y];
}
```

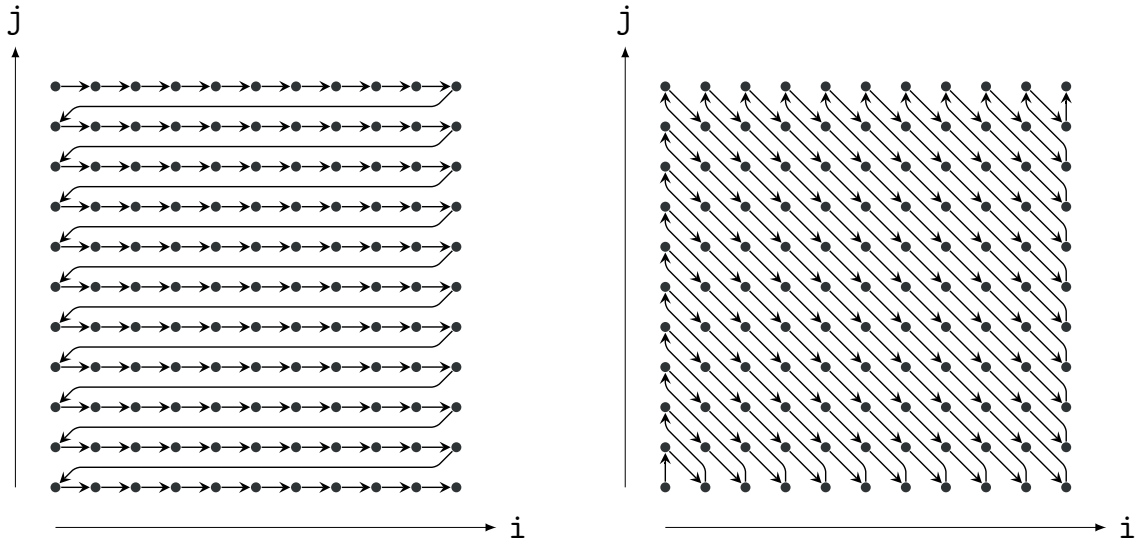
we simply need to produce the following schedule:

$$\text{Sched'} = \{\text{S}[k] \mapsto [k, 0, 0]\} \cup \{\text{T}[i, j] \mapsto [i + j, i, 1]\}$$

This corresponds to applying for T a skew of the innermost loop by the outermost one $\{[1, i, j] \mapsto [1, i, i + j]\}$, a loop permutation of the two loops $\{[1, i, i + j] \mapsto [1, i + j, i]\}$,

then a loop fusion of the loops for both S and T, pushing inside the additional dimension used for sequentiality $\{[0, k, 0] \mapsto [k, 0, 0]\}$ for S, and $\{[1, i+j, i] \mapsto [i+j, i, 1]\}$ for T. The iteration for S in the inner dimension was then gracefully separated from the innermost loop by the code generator (to avoid the need of a guard), making the loop fusion of the innermost loop useless.

Figure 1.6 shows two valid schedules (only depicting the instances of T), in a graphical way so that we can quickly see that the schedules are valid (remember that for this example, flow dependences are along the diagonal directed by the vector (1,-1)). These two schedules will be used later on for tiling this example.



(a) Loop reversal with loop interchange.

(b) Loop skewing with loop interchange.

Figure 1.6 – Two different valid schedules for the polynomial product example (only instances of T are shown).

As we can see, there are multiple difficulties: we first need to define a valid (and better) schedule, then to be able to generate the code (with the correct loop structure and loop bounds). There are numerous approaches for finding good schedules in the polyhedral model, and they usually involve searching the space of valid schedules (schedules that respect a set of dependences) for one that is optimum for a given criteria (usually in the form of a linear function, or a combination of linear functions).

For the scheduling problem, many algorithms have been developed. Two of them stand out: the seminal scheduling method by Feautrier [38], which is efficient at finding innermost parallelism, critical for efficient vectorization, but provides codes with relatively poor locality, and the Pluto [15] scheduler, which optimizes for both locality and parallelism. The latter one looks for schedules that offer good tileability and relatively

good coarse-grained parallelism. Tiling considerations will be described in more details in Section 1.2.7 as this is one of the main subjects of this thesis.

For the code generation problem, the CLooG [10] code generator is widely accepted as producing good quality code. The generator is influenced by many parameters, the most important ones being loop separation and loop unrolling. Loop separation will split loops into several pieces if it avoids the generation of branching in its body or in the body of its inner loops. This diminishes control overhead (by reducing the number of branches inside loops), but might duplicate code (in case of splitting at multiple levels). Loop unrolling will simply fully unroll a loop, while partial unrolling requires to strip-mine (by schedule transformation, see above), and then unroll the inner loop.

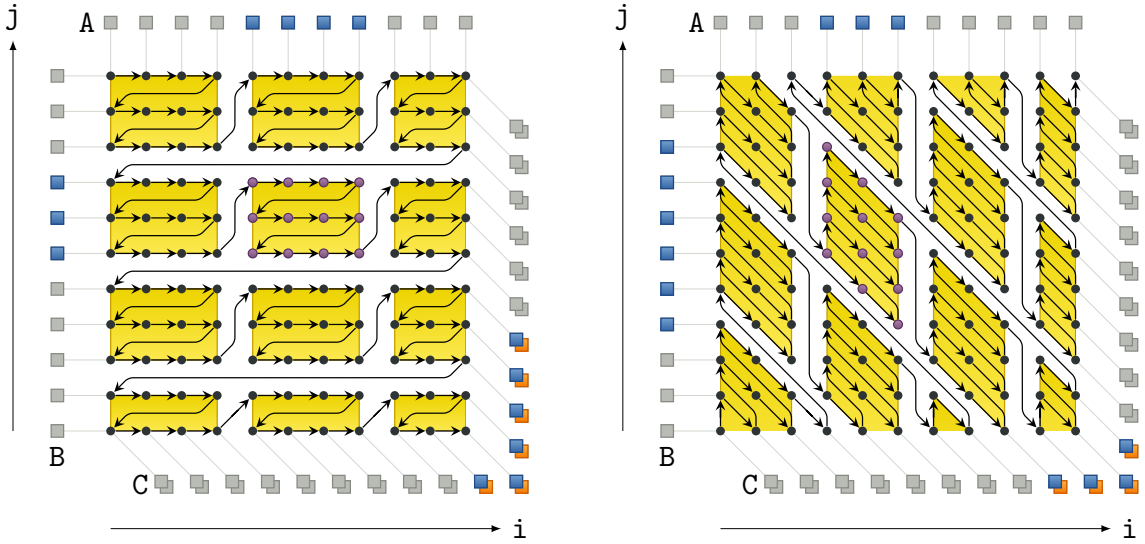
Both scheduling and code generation techniques are implemented with slight variations within the integer set `isl` library [73].

1.2.7 Tiling

Tiling is the flagship optimization of the polyhedral model as it improves locality, facilitates memory access coalescing, provides coarse-grained parallelism, and allows the programmer or compiler to modulate arithmetic intensity (the ratio between computation and memory transfer) through tile size selection. The idea is to split a set of loops into multi-dimensional chunks, which can each be fully executed, in an atomic fashion, before going to the next one. The resulting code is a set of nested loops that iterate inside the tiles (point loops), enclosed in a set of nested loops that iterate over the tiles (tile loops).

We are interested in regular tiling (where all chunks have the same shape, except maybe on the border of the iteration space), and more specifically in rectangular tiling (where tiles are boxes with edges parallel to the axes of the basis used, and aligned into a grid). Rectangular tiling, and tiling in general, is not always a legal transformation (except in 1D, where it is equivalent to strip-mining). Indeed, grouping instructions blindly can create cycles between two groups, making their scheduling impossible. A preliminary transformation is usually applied on the code to enable rectangular tiling. This is illustrated on Figure 1.7 for the polynomial product example, where the loop transformations of the previous section (Figure 1.6) were applied before tiling.

Figure 1.7 displays the memory elements accessed by one of the tiles. As we can see, in the most favorable case, there is a considerable amount of data reuse when executing the points of a tile. Here, the amount of computations grows quadratically with the size of the tile (area), while the amount of communication grows linearly (projection). This is generally the case for codes whose access patterns show a good multi-dimensional locality.



(a) Rectangular tiles corresponding to the schedule: `[i,j] -> [-j,i]`.

(b) Parallelepipedic tiles, which are rectangular for the schedule: `[i,j] -> [i+j,i]`.

Figure 1.7 – Different tilings of the polynomial product example (using different schedules). The points represent the iterations (purple, as opposed to black, for the points of the current tile), the arrows represent the sequential execution order, the yellow shapes represent the atomic tiles, the squares represent the data (blue is a load, orange is a store).

The Pluto [15] scheduling algorithm is designed to provide schedules that exhibit tileability. More specifically, it tries to produce a set of nested loops where multiple consecutive loops can safely be interchanged, which guarantees the legality of rectangular tiling. The generated schedule is also optimized to minimize the number of dependences crossing tile boundaries, the objective being to minimize the amount of communication between tiles, and, in the extreme case, to exhibit parallelism between tiles (when no communications due to flow dependences occur). For the polynomial product kernel, if RaW distances are minimized, `isl` produces $\{\mathrm{S}[k] \mapsto [k,0,0]\} \cup \{\mathrm{T}[i,j] \mapsto [i+j,i,1]\}$, which is the schedule Sched' mentioned before, leading to the tiling of Figure 1.7b,

Validity

Loop permutability is a sufficient condition for rectangular tiling to be legal for any tile size. Indeed, rectangular tiling can be seen as a combination of strip-mining (always legal), and loop permutation. It might be necessary to initially apply a skew transformation and/or a loop reversal to obtain a set of nested loops in which loops can be freely permuted. Finding such transformations is done by polyhedral schedulers. An easy way of visualizing the permutability of loops is by checking that dependence distances (differences of the two

iteration vectors of the operations in dependence) are nonnegative along each axis of the schedule, i.e., they point to the first orthant. In Figure 1.6a, the dependences are indeed nonnegative when projected along $\vec{i}$ and along $-\vec{j}$, therefore the loops can be permuted. This is the same in Figure 1.6b along $\vec{i}+\vec{j}$ and $\vec{i}$.

Full loop permutability is however not a necessary condition in general, in particular for specific tile sizes (in particular small sizes or sizes as large as the domain). But as we are interested in parametric tile sizes for any size, we will only consider tiling on a permutable band of loops, i.e., successively nested loops.

Tile sizes

Selecting tile sizes (and shape) is one of the most critical steps for optimizing code via tiling. A poor choice of tile sizes may produce worse results than without tiling due to the control overhead introduced. However, good tile sizes can enable vectorization, can provide the adequate middle ground between instruction-level parallelism and thread-level parallelism, and can drastically improve cache utilization. It is also a privileged way of introducing intermediate buffers for scratchpad programming or for decomposing kernels into smaller pieces ready to be exported/offloaded to an external accelerator such as a GPU or a FPGA. Then, it is also possible to implement some pipelining of the transfers of consecutive tiles, in the same way advanced programmers implement double buffering.

Tile sizes selection is therefore guided by many hardware specific constraints. The local buffers for the tiles have to fit in the local memory while the communication time between the local and the remote memory must be as small as possible. It is most of the time advised to increase the tile sizes until the data necessary for the tile fills the local memory, as increasing the tile sizes is more likely to increase memory reuse and therefore decrease memory transfer. Even when the computation is compute bound, increasing the tile sizes can still decrease communications and therefore energy consumption, even if performance is not increased. When using tiling for vectorization purposes, choosing the tile size to be a multiple of the vector size is advised.

With a parametric analysis with regard to tile sizes, many of these characteristics can be expressed with a closed formula. This helps to guide the selection of tile sizes but also allows the actual selection to be done at runtime, when the hardware characteristics are known. The performance for different tile sizes can also be evaluated (through auto-tuning) without the need to recompile. It might also generate codes that can adapt the tile sizes to better fit the input data. This is what motivated us to extend different analyses and optimizations related to tiling towards parametric tiling, i.e., where tile sizes are handled as parameters and not as numerical constants known at compile time.

Chapter 2

Inter-Tile Reuse for Parametric Tiling

Summary

As briefly exposed in Section 1.2.7, loop tiling is a loop transformation widely used to improve spatial and temporal data locality, to increase computation granularity, and to enable blocking algorithms, which are particularly useful when offloading kernels on computing units with smaller memories. When caches are not available or used, data transfers and local storage must be software-managed, and some useless remote communications can be avoided by exploiting data reuse between tiles. An important parameter of tiling is the sizes of the tiles, which impact the size of the required local memory. However, for most analyzes involving several tiles, which is the case for inter-tile data reuse, the tile sizes induce non-linear constraints, unless they are numerical constants. This complicates or prevents a parametric analysis with polyhedral optimization techniques.

This chapter shows that, when tiles are executed in sequence along tile axes, the parametric (with respect to tile sizes) analysis for inter-tile data reuse is nevertheless possible, i.e., one can determine, at compile-time and in a parametric fashion, the copy-in and copy-out data sets for all tiles, with inter-tile reuse, as well as sizes for the induced local memories. When approximations of transfers are performed, the situation is much more complex, and involves a careful analysis to guarantee correctness when data are both read and written. We provide the mathematical foundations to make such approximations possible. Combined with hierarchical tiling, this result opens perspectives for the automatic generation of blocking algorithms, guided by parametric cost models, where blocks can be pipelined and/or can contain parallelism. Previous work on FPGAs and GPUs already showed the interest and feasibility of such automation with tiling, but in a non-parametric fashion.

2.1 Motivation

Todays' hardware diversity increases the need for optimizing compilers and runtime systems. As we sketched in Section 1.1, a difficulty when using hardware accelerators (FPGA, GPU, dedicated boards) is to automatically perform kernel/function offloading (a.k.a. outlining as opposed to inlining) between the host and the accelerator, and to organize data transfers between the different memory layers (e.g., in a GPU, from remote to global memory, and from global to shared memory, or even registers). This requires static analysis to identify the kernel input (data read) and output (data produced), and code generation for transfers, synchronizations, and computations. In general, such tasks are done by the programmer who has to express the communications, to allocate and size the intermediate buffers, and to decompose the kernel into fitting chunks of computation. When each kernel is offloaded in a three-phase process (i.e., upload, compute, store back), such programming remains feasible. For GPUs, developers can use OpenCL or CUDA, or they can rely on higher-level abstractions (e.g., compilation directives as in OpenACC or garbage collector mechanisms as in [19]), static analysis as in OpenMPC [53], runtime approaches as in [52], or mixed compile/runtime optimizations as in [59]. These approaches mainly work at the granularity of variable names, still defined by the programmer, but they can be used to optimize remote transfers when several kernels are successively launched. Things get more complicated when a given kernel is decomposed into smaller kernels (and the initial arrays into array regions) to get blocking algorithms, thanks to *loop tiling*. Indeed, iteration-wise loop analysis and element-wise array analysis are needed to enable intra- and inter-tile data reuse. Moreover, the choice of tile sizes is driven by hardware capabilities such as memory bandwidth, size, and organization, computational power, and such codes are very hard to obtain without automation and some cost model. With this objective, our contribution is a **parametric** (w.r.t. tile sizes) polyhedral analysis technique for **inter-tile data reuse** and a mathematical framework to reason with **approximations** of data accesses and transfers.

Loop tiling is a well-known transformation used to improve data locality [79], increase computation granularity, and control the use and size of local memories for out-of-core computations (we refer to [81] for details on semantics, validity conditions, and code generation). It was first introduced as "supernode partitioning" [48], for a set of perfectly nested loops, as a grouping of iterations into *supernodes* [48], which are atomic (i.e., can be executed without any communication/synchronization with other supernodes except for live-in/live-out data at beginning/end of a tile execution), identical by translation, bounded, and form a partition of the whole iteration space. Validity conditions were

given in terms of dependence cones and hyperplane partitioning, which define tiles, when the number of hyperplanes equals the space dimension, as hyper-rectangles (after some possible change of basis) and establish a link with affine scheduling and the generation of permutable loops. Now, tiling is also used for non-perfectly nested loops [16], thanks to multi-dimensional affine loop transformations: as in the perfectly nested case, some permutable dimensions can be used to perform tiling, even if not all instructions have the same iteration domain, as long as they are all mapped (by a scattering or scheduling function as recalled in Section 1.2) into a common space. Analysis and code generation may involve more complex sets, but the principles are similar. Today, loop tiling is still a key loop transformation for performance (speed, memory, locality) and the subject of many new advanced developments, including non-rectangular tiling.

Let us recall here and detail a bit more some of the explanations already provided in Section 1.2.7. Loop tiling can be viewed as a composition of strip-mining and loop interchange, after a preliminary change of basis. It transforms n nested loops into n *tile loops* iterating over the tiles, surrounding n *intra-tile loops* (or *point loops*) iterating within a tile. Dependence analysis and code generation for loop tiling is well-established in the polyhedral model [40], i.e., for a set of nested `for` loops, writing and reading multi-dimensional arrays and scalar variables, where loop bounds, `if` conditions, and array access functions are affine expressions of surrounding loop counters and structure parameters. In this case, loop iterations can be represented by a *polyhedral iteration domain*. When tile sizes are numerical constants, parametric (w.r.t. program counters and structural parameters) polyhedral optimizations (e.g., linear programming) can be used although loop tiling transforms n loops into $2n$ loops. Indeed, the image by tiling of an n-dimensional polyhedral iteration domain can be expressed as a $2n$-dimensional polyhedral iteration domain, because the set of points after tiling with fixed sizes can be described by affine inequalities.[1] In general, **parametric tiling** refers to the case where tile sizes are parameters too. Parametric analysis within a tile is in general feasible as the set of points in a tile is defined with affine constraints from the tile sizes and the *tile origin* (first corner of the tile). However, when an analysis involves several tiles, it becomes more intricate, if not unsolvable, as *a priori* expressing the tiled space with tile sizes as parameters induces quadratic constraints. For example, the tiling theory developed in [80], the code generation schemes of [48, 41, 16], the data movement and scratchpad optimizations of [50, 49, 9, 5, 62, 75] are not parametric. Recently, efficient code generation for parametric tiling [65, 47] as well as some forms of symbolic scheduling for tiled codes [17] have been developed.

[1] However, difficulties due to large coefficients are possible.

In the context of high-level synthesis (HLS), inter-tile data reuse was proposed [3] (then automated [5]) as a source-to-source process on top of Altera C2H HLS tool, to offload small computation kernels to FPGAs while optimizing communications from a remote (in this case external) DDR memory. Similar results with data reuse between two successive tiles only were then demonstrated for AutoESL Xilinx tool [62]. Different (and more restricted) forms of inter-tile data reuse were also designed for programmable accelerators such as GPUs [7, 45, 75]. However, none of these approaches are parametric with respect to tile sizes.

In this chapter, we show that maximal inter-tile data reuse can be expressed in the parametric case, even in an approximated situation. The trick to get around a quadratic formulation is to work with all possible tiles – not just the tiles that are part of the iteration space partitioning and whose origins belong to a lattice – but the difficulty is to make sure that exactness and correctness are maintained. Our contributions, mostly at the level of code analysis, are the following:

- When read/write accesses can be described in an exact way using polyhedral representations, we show how to derive, thanks to manipulations of integer sets, the copy-in and copy-out sets for each tile, with parametric tile sizes. This gives a full parametric generalization of the inter-tile data reuse of [5].

- We extend this parametric analysis to handle approximations, which make the analysis more complex when some data may be both read and written by the tiles, as loading too much may not be safe. We introduce the concept of *pointwise functions* for which no additional loss of accuracy is induced.

- Using similar principles, a parametric analysis can be done in the following steps of the compilation too, in particular to perform a parametric array contraction for the definition of local arrays. This will be demonstrated in Chapter 5, using the analysis of conflicts between array elements developed in Chapter 3 for parallel specifications and, if needed, the parametric memory allocation scheme developed in Chapter 4. The reason why we need a specific analysis of conflicts is twofold: because we want to consider the case of pipelining (which is a particular parallel specification) too and because our way of dealing with all tiles, not just those aligned with a lattice, requires some deeper understanding to guarantee that the technique is indeed correct (because it is conservative). This will be detailed later on.

2.2 Prerequisites

2.2.1 Notations and Definitions

We write all vectors with a letter topped by an arrow such as $\vec{i}$, whose components are denoted $i_1, \ldots, i_n$. The vector $\vec{0}$ (resp. $\vec{1}$) has all components equal to 0 (resp. 1) and $\vec{a} \circ \vec{b}$ is the product (component-wise) of $\vec{a}$ and $\vec{b}$. We denote by $\preceq$ the lexicographic total order on vectors of arbitrary size and by $\leq$ the component-wise partial order on vectors with same size, defined by $\vec{i} \leq \vec{j}$ if and only if (iff) $i_k \leq j_k$ for all k.

We will not elaborate on how to build and interpret the different affine functions for tiling non-perfectly nested loops. To simplify the discussion and notations, we only focus on the n dimensions to be tiled. We assume that each statement S with polyhedral iteration domain $\mathcal{D}_S$ (scanned with the iteration vector $\vec{i}$) is tiled, after a first affine mapping $\vec{i} \mapsto \vec{i'} = \theta(S, \vec{i})$, by canonical tiles whose sizes are specified by a vector $\vec{s}$. In other words, a point $\vec{i}$ is mapped to the tile indexed by $\vec{T}$ where $T_k = \lfloor \frac{i'_k}{s_k} \rfloor$, or equivalently $s_k T_k \leq (\theta(S, \vec{i}))_k < s_k(T_k{+}1)$, for $k \in [1..n]$, i.e., $0 \leq \theta(S, \vec{i}) - \vec{s} \circ \vec{T} \leq \vec{s} - \vec{1}$. Also, we restrict to the case where the original and the tiled programs are both executed sequentially.[2] Several orders of iterations in the tiled program are possible, we consider that the tiled code is executed following the lexicographic order on the $2n$-dimensional vectors $(\vec{T}, \vec{i'})$. The tiled iteration domain for statement S is then:

$$\mathcal{T}_S = \{(\vec{T}, \vec{i'}) \mid \exists \vec{i} \in \mathcal{D}_S,\ \vec{i'} = \theta(S, \vec{i}),\ \vec{0} \leq \vec{i'} - \vec{s} \circ \vec{T} \leq \vec{s} - \vec{1}\}$$

If θ is a one-to-one mapping and $\mathcal{D}_S$ the set of integer points in a polyhedron, then $\vec{i}$ can be eliminated and $\mathcal{T}_S$ is also the set of integer points in a polyhedron.

Example We illustrate the concepts and steps of our technique with the kernel from PolyBench [63] named `jacobi_1d_imper`, with a time loop, and tiled in 2D. For the code in Figure 2.1, the Pluto compiler [61] generates the following mapping:

$$\theta(S_1, (t, i)) = (t, 2t + i, 0) \quad \theta(S_2, (t, j)) = (t, 2t + j + 1, 1)$$
$$\mathcal{D}_{S_1} = \mathcal{D}_{S_2} = \{(t, i) \mid 0 \leq t \leq M - 1,\ 0 \leq i \leq N - 2\}$$

This means shifting S_2 by 1 in the j loop, fusing the i and j loops, then skewing by 2 the inner loop, to get the code of Figure 2.2. Then, several tiled code generations are

[2] However, parallelism inside a tile is possible, as well as hierarchical tiling, which enables to play with the extent of the tiled domain. Parallel execution are also possible by defining a partial execution order, if execution follows the axes defining tiles. It seems possible to handle other situations but with additional complications and approximations, and not in all cases.

```
for (t = 0; t < M; t++) {
  for (i = 1; i < N - 1; i++)
    S1: B[i] =
        (A[i-1] + A[i] + A[i+1])/3;
  for (j = 1; j < N - 1; j++)
    S2: A[j] = B[j];
}
```

Figure 2.1 – Original kernel.

```
for (t = 0; t < M; t++)
  for (i' = 2t+1; i' < 2t + N; i'++) {
    S0: i = i'-2t;
    S1: if (i<N-1) B[i] =
        (A[i-1] + A[i] + A[i+1])/3;
    S2: if (i>1) A[i-1] = B[i-1];
  }
```

Figure 2.2 – Transformed kernel.

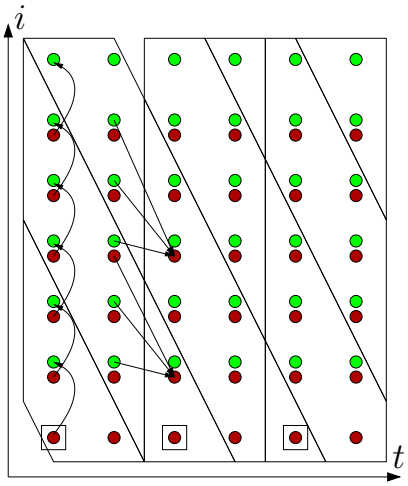
possible depending on how iterators are defined and how tiles are aligned, i.e., what the underlying lattice of the tiling is. With the relation $T_k = \lfloor \frac{i_k}{s_k} \rfloor$, tiles are aligned with the canonical basis obtained after the transformation θ (see Figure 2.3 for tiles of size 2×3, drawn in the original basis to save space). With the "outset" code generation scheme of [65], for tile sizes $s_1 \times s_2$, we get:

```
for (T1 = 0; T1 < M; T1+=s1) {
  lb = 2T1+1-(s2-1); lb = s2*ceiling(lb/s2);
  for (T2 = lb; T2 < 2T1 + N + 2(s1 - 1); T2+=s2)
    for (t=max(0,T1); t<min(M,T1+s1); t++)
      for (i'=max(2t+1,T2); i'<min(2t+N,T2+s2); i'++) {
        S0: i = i'-2t;
        S1: if (i<N-1) B[i] = (A[i-1] + A[i] + A[i+1])/3;
        S2: if (i>1) A[i-1] = B[i-1];
      }
}
```

For our scheme, it would also be valid to shift, after tiling, the inner tile-loop w.r.t. the outer tile-loop, i.e., to move up or down each column in Figure 2.3. □

2.2.2 Inter-Tile Data Reuse

The inter-tile reuse problem we formalize here is the kernel offloading with optimized remote accesses presented in [3, 5], even if other variations are possible. A kernel is tiled and offloaded, tile by tile, to a computing accelerator (a FPGA in [3, 5]). Initially, all data are in remote memory, while all computations are performed on the accelerator. Each tile $\vec{T}$ consists of three *successive* phases: a *loading* phase where data are copied from remote memory to local memory, enabling burst communications, then a *compute* phase where the original computations corresponding to the tile are performed on the local memory, and finally a *storing* phase where data are copied to remote memory. In addition, all compute (resp. loading and storing) phases are performed in sequence, following the



Non-empty 2×3 tiles drawn with respect to the original space. Instruction S_1 is in red. Instruction S_2 is in green.

Are also shown some flow dependences, due to reads of `B`, at distance $(0, 1)$, and reads of `A`, at distance $(1, 0)$, $(1, -1)$, $(1, -2)$ in the (t, i) space.

Figure 2.3 – Kernel `jacobi1d` and skewed tiling.

lexicographic order on tile indices. Nevertheless, loads and stores can be done concurrently with the computations of other tiles, enabling pipelining, computation/communication overlapping, and execution similar to double buffering. *Inter-tile reuse* makes this possible even when data are both read and written.[3]

Then, the "maximal inter-tile data reuse problem" is to define the loading and storing sets Load($\vec{T}$) and Store($\vec{T}$) for each tile $\vec{T}$ so that a data element is never loaded from remote memory if it is already available in local memory, i.e., if it has already been loaded or computed (as, in this latter case, the remote memory is not necessarily up-to-date). This inter-tile reuse is performed for each *tile strip* (subspace of tiles corresponding to inner tile dimensions). In [5], a tile strip is one-dimensional, but the technique can be applied to multi-dimensional strips. This choice however impacts the size of the local memory. We want to do this analysis independently of how loads, computations, and stores will actually be scheduled/pipelined at runtime, in other words, we want to perform this analysis for any schedule that respects the dependence task graph of Figure 2.4.

The problem we want to address has some similarities with the reuse analysis of Größlinger [44], but with fundamental differences. Given a "sliding window" of iterations, this analysis identifies the data that each iteration needs to bring because they were not already present due to previous iterations in the sliding window. But the communications are not coalesced out of the tile, they are still at the iteration level. In other words, this is a reuse analysis at constant (possibly parametric) distance (the sliding window), but with

[3]Without inter-tile reuse, full pipelining of tiles is not always possible if a data is locally written, then read in a subsequent tile. Indeed, one would then need to wait for the data to be stored in remote memory before loading it again. Inter-tile reuse enables to break such a cycle of synchronizations and avoid considering latencies. This will be illustrated in more details in Chapter 5.

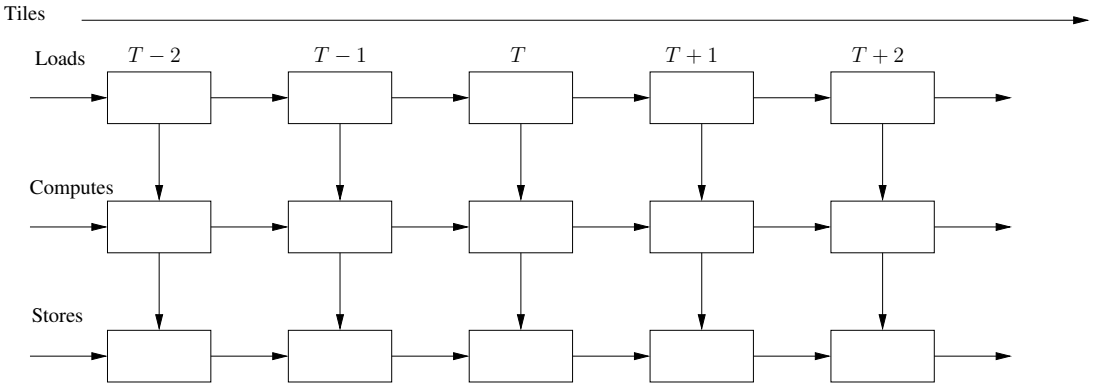


Figure 2.4 – Minimal dependence task graph for loads, computations, stores.

no granularity or scheduling (through tiling) reorganization, which makes the problem different and, actually, simpler.

Let us first recall the approach of [5], which is based on parametric linear programming [35] (we will use a different approach to make it parametric w.r.t. tile sizes). It consists in performing loads (resp. stores) as late (resp. as soon) as possible, i.e., a data element is loaded just before the first tile that accesses it, if this access is a read, and is stored just after the last tile that writes it. Among all schemes that exploit a full inter-tile reuse in a strip, this tends to reduce the size of the local memory. We illustrate this technique again on the `jacobi_1d_imper` example.

Example (cont'd) For the tiling of Figure 2.3, a 1D tile strip is vertical, indexed by $T_1 = \lfloor \frac{t}{s_1} \rfloor$. To simplify explanations, we only consider the array `A` (the array `B` is not live-in of a tile strip). We compute the first operation (following the order defined by the tiling) that accesses `A[m]`. This means computing, with $(i_1, i_2) = (t, i)$ and the four parameters M, N, m, and T_1, the lexicographic minimum of $(T_2, i'_1, i'_2, k, i_1, i_2)$ in a set defined by a disjunction of two conjunctions of affine inequalities derived from the program (iteration domains and access functions):

$$\begin{cases} -1 \leq m - i_2 \leq 1,\ 0 \leq i_1 \leq M-1,\ 1 \leq i_2 \leq N-2,\ k = 0, \\ i'_1 = i_1,\ i'_2 = 2i_1 + i_2,\ 0 \leq i'_1 - 2T_1 \leq 1,\ 0 \leq i'_2 - 3I_2 \leq 2 \end{cases}$$
$$\vee$$
$$\begin{cases} m = i_2,\ 0 \leq i_1 \leq M-1,\ 1 \leq i_2 \leq N-2,\ k = 1, i'_1 = i_1, \\ i'_2 = 2i_1 + i_2 + 1,\ 0 \leq i'_1 - 2T_1 \leq 1,\ 0 \leq i'_2 - 3T_2 \leq 2 \end{cases}$$

The first set of constraints corresponds to reads in S_1 and specifies that `A[m]` is `A[i-1]`, `A[i]`, or `A[i+1]`, that iterations in tiles are valid ($(T_1, T_2, i'_1, i'_2) \in \mathcal{T}_S$), and $k = 0$ is the third component of $\theta(S_1, (t, i))$ (i.e., S_1 is the first executed statement in the loop body). The second set of constraints corresponds to writes in S_2 (with $k = 1$, i.e., second executed

statement in the loop body). The lexicographic minimum is expressed as a disjunction of cases (a QUAST or quasi affine solution tree [35]). Then, all solutions (i.e., leaves of the tree) that correspond to a write operation are removed. Here, all first accesses are reads, no simplification is needed. It remains to project out the variables i'_1, i'_2, i_1, i_2, k, to get a relation between tile index $\vec{T}$ and array element m, which describes $\text{Load}(\vec{T})$ as a union:

$$\text{Load}(\vec{T}) = \begin{array}{l} \{m \mid 0 \leq 2T_1 \leq M-1,\ 2 \leq m \leq N-1,\ 1 \leq m + 4T_1 - 3T_2 \leq 3\} \\ \qquad\qquad \cup \\ \{m \mid 0 \leq m \leq 1,\ 3 \leq N,\ 0 \leq 2T_1 \leq M-1,\ -1 \leq 4T_1 - 3T_2 \leq 1\} \end{array}$$

The second set loads the additional `A[0]` and `A[1]` for the unique tile in the strip that contains an iteration $(t, 1)$ on its first column (squares in Figure 2.3).

As can be seen from the inequalities involved in the previous example with $\vec{s} = (2, 3)$ (and in the definition of $\mathcal{T}_S$), considering the components of the size vector $\vec{s}$ as parameters generates **quadratic constraints**. In other words, this formulation is inherently not linear in the tile sizes. The goal of this chapter is to show that, surprisingly, the problem can nevertheless be solved, both for exact inter-tile reuse (as in the previous example) and with approximations.

2.3 Dealing with Unaligned Tiles

The first key idea to break the non-linearity constraint is to represent each tile not with its tile index $\vec{T}$ defined earlier, but with the index $\vec{I}$ of its *origin* (first element in the tile in the lexicographic order). The first difference is that tiles are scanned with loops with increments equal to $\vec{1}$ when $\vec{T}$ is used and equal to $\vec{s}$ when $\vec{I}$ is used. The second difference is that, when $\vec{I}$ is used instead of $\vec{T}$, the set of elements $\vec{i}$ in a tile is affine in $\vec{s}$: this is the set of all $\vec{i}$ such that $\vec{I} \leq \vec{i} \leq \vec{I} + \vec{s} - \vec{1}$. In other words, parametric analysis inside a tile is possible. This representation is not new, it is used for the analysis of tile footprint in PIPS [46, Fig. 6] and for the parametric code generation [65] used for the tiled code of Section 2.2.1. However, when reasoning with different tiles, the non-linearity is coming back. Indeed, in a given execution, the tile origins $\vec{I}$ are restricted to the lattice $\mathcal{L}$ defined by $\vec{I} \in \mathcal{L}$ iff $\vec{I} = \vec{s} \circ \vec{J}$ for some integer vector $\vec{J}$. The second key idea is to show how these quadratic constraints can nevertheless be ignored, by reasoning on the set of all tiles of size $\vec{s}$, not just those restricted to $\mathcal{L}$. The inter-tile reuse problem then becomes (piece-wise) affine in $\vec{s}$ as we will show.

Note that, with standard conditions for tiling (i.e., when all dependence distances are non-negative along the dimensions being tiled [48]), if a tiling is valid, any translation of it is valid too. In other words, considering all tile origins $\vec{I} = \vec{s} \circ \vec{J} + \vec{I}_0$ for some vector $\vec{I}_0$

defines a valid tiling too. This has the same effect as defining the tiling from the shifted mapping $\vec{i} \mapsto \sigma(S, \vec{i}) - \vec{I}_0$ for all S. Hereafter, we say that two tiles are *aligned* if they belong to the same tiling.

2.3.1 Exact Approach with Set Equations

In Section 2.2.2, maximal inter-tile data reuse was expressed as a linear programming optimization, following [5]. It can be equivalently formulated with set equations [4], expressed in terms of $\mathrm{In}(\vec{T})$ and $\mathrm{Out}(\vec{T})$, the standard *live-in* and *live-out* sets for tile $\vec{T}$, as defined for example for array region analysis [24]:

$$\begin{aligned}\mathrm{Load}(\vec{T}) &= \mathrm{In}(\vec{T}) \setminus \bigcup_{\vec{T}' \prec \vec{T}} (\mathrm{In} \cup \mathrm{Out})(\vec{T}') = \mathrm{In}(\vec{T}) \setminus (\mathrm{In} \cup \mathrm{Out})(\vec{T}' \prec \vec{T})\} \\ \mathrm{Store}(\vec{T}) &= \mathrm{Out}(\vec{T}) \setminus \bigcup_{\vec{T}' \succ \vec{T}} \mathrm{Out}(\vec{T}') = \mathrm{Out}(\vec{T}) \setminus \mathrm{Out}(\vec{T}' \succ \vec{T})\end{aligned}$$

Here, as indicated in the previous formulas, $X(\vec{T}' \prec \vec{T})$ is a shortcut to denote the union of all sets $X(\vec{T}')$ for all tiles $\vec{T}'$ executed before $\vec{T}$ (lexicographic order) in the same tile strip as $\vec{T}$. Expressing $X(\vec{T}' \prec \vec{T})$ from $X(\vec{T}')$ is done simply by adding the constraint $\vec{T}' \prec \vec{T}$ and specifying that $\vec{T}'$ is in the strip where reuse is exploited. The previous set equations state that we load what is live-in for $\vec{T}$ and not previously live-in (redundant load) or live-out (defined locally), and we store what is live-out, but not again live-out later (redundant store). One could expect to rather subtract $\mathrm{Load}(\vec{T}' \prec \vec{T})$ from $\mathrm{Load}(\vec{T})$ and $\mathrm{Store}(\vec{T}' \succ \vec{T})$ from $\mathrm{Store}(\vec{T})$, but such recursive implicit definitions are not usable, and an explicit formulation is preferable.

We now rephrase these equations when tiles $\vec{T}$ are represented by their tile origins $\vec{I}$ as previously explained. We also consider *all* tiles with size $\vec{s}$, not just those whose origins belong to the lattice $\mathcal{L}$, i.e., even those that will not be executed in a given tiling. These tiles contain valid iterations (which will be executed as part of an *aligned* tile), but their Load and Store sets will not generate transfers during the execution. We define two relations on tiles:

- $\vec{I}' \sqsubset_{\vec{s}} \vec{I}$ iff $\vec{I}' \prec \vec{I}$ and $\vec{I} - \vec{I}' \in \mathcal{L}$. This is equivalent to the lexicographic order $\vec{T}' \prec \vec{T}$ for the corresponding tile indices.
- $\vec{I}' \prec_{\vec{s}} \vec{I}$ iff, for some $k \in [1..n]$, $I'_i \leq I_i$ for all $i < k$ and $I'_k \leq I_k - s_k$ where n is the dimension of $\vec{I}$ and $\vec{I}'$. This is a variation of the lexicographic order.

The standard reflexive extensions $\sqsubseteq_{\vec{s}}$ and $\preceq_{\vec{s}}$ of these relations are clearly partial orders. Figure 2.5 shows all tile origins $\vec{I}'$ strictly smaller (in blue) or strictly larger (in red) than

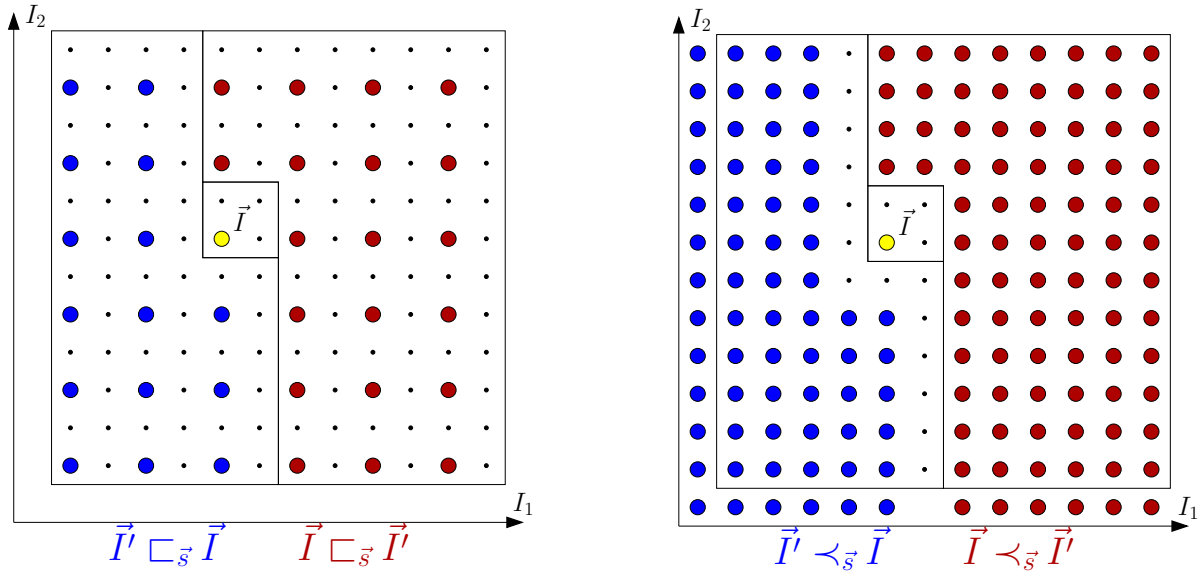


Figure 2.5 – Orders $\sqsubseteq_{\vec{s}}$ and $\preceq_{\vec{s}}$. Points are tile origins. Here $\vec{s} = (2, 2)$.

the tile origin $\vec{I}$ (in yellow), for the orders $\sqsubseteq_{\vec{s}}$ and $\preceq_{\vec{s}}$. Note that tiles comparable for $\sqsubseteq_{\vec{s}}$ are always aligned with each other. An alternate, maybe more intuitive, definition of $\prec_{\vec{s}}$ is as follows: $\vec{I'} \prec_{\vec{s}} \vec{I}$ iff, in the tiling induced by $\vec{I}$ (the same is true with $\vec{I'}$, this is symmetric), every point in the tile $\vec{I'}$ is executed before any point in the tile $\vec{I}$ (but $\vec{I}$ and $\vec{I'}$ may not be aligned, i.e., they may not be both executed at runtime).

With tile origins, the previous Load/Store equations can be rewritten as:

$$\text{Load}(\vec{I}) = \text{In}(\vec{I}) \setminus (\text{In} \cup \text{Out})(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) \tag{2.1}$$

$$\text{Store}(\vec{I}) = \text{Out}(\vec{I}) \setminus \text{Out}(\vec{I'} \sqsupset_{\vec{s}} \vec{I}) \tag{2.2}$$

The key is now to show that these sets can also be defined equivalently as:

$$\text{Load}(\vec{I}) = \text{In}(\vec{I}) \setminus (\text{In} \cup \text{Out})(\vec{I'} \prec_{\vec{s}} \vec{I}) \tag{2.3}$$

$$\text{Store}(\vec{I}) = \text{Out}(\vec{I}) \setminus \text{Out}(\vec{I'} \succ_{\vec{s}} \vec{I}) \tag{2.4}$$

This is not obvious as the contribution of unaligned tiles (i.e., not in the same tiling as $\vec{I}$) is also subtracted, thus the Load/Store sets could now be too small. Nicely, these sets **only involve affine constraints** as the relation $\prec_{\vec{s}}$ is, by definition, piece-wise affine (this is also the case for a similar "happens-before" relation defined on iteration points). They can thus be computed with a library such as `isl` [73]. Before proving these formulas, we first illustrate their use.

Example (cont'd) The following sets were computed thanks to the `isl` calculator `iscc` [74] with the generic script of Figure 2.6, for `jacobi_1d_imper` (see Figure 2.3).

$$\begin{aligned}
\text{Load}(\vec{I}) \;=\; & \{\text{A}(m) \mid 1 \le m + 2I_1 - I_2 \le s_2,\ s_1 \ge 1,\ I_1 \ge 0,\ m \ge 1,\ I_1 \le -1 + M,\\
& \qquad I_2 \ge 2 - s_2 + 2I_1,\ m \le -1 + N,\ N \ge 3\}\\
\cup\ & \{\text{A}(m) \mid m \ge 1 + I_2,\ m \ge 1,\ M \ge 1,\ m \le -1 + N,\ I_1 \le -1,\\
& \qquad I_1 \ge 1 - s_1,\ I_2 \ge 2 - s_2,\ N \ge 3,\ m \le s_2 + I_2\}\\
\cup\ & \left\{\text{A}(1) \mid\ I_2 = 1 + 2I_1 \wedge 0 \le I_1 \le -1 + M,\ N \ge 3,\ s_1 \ge 1,\ s_2 \ge 1\ \right\}\\
\cup\ & \left\{\text{A}(m) \mid\ 0 \le m \le 1,\ I_2 = 1 \le s_2,\ 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3\ \right\}\\
\cup\ & \left\{\text{A}(0) \mid\ 0 \le I_1 \le M - 1,\ N \ge 3,\ s_1 \ge 1,\ 1 \le I_2 - 2I_1 \ge 2 - s_2\ \right\}\\
\cup\ & \left\{\text{A}(0) \mid\ 1 - s_1 \le I_1 \le -1,\ M \ge 1,\ N \ge 3,\ I_2 \ge 2 - s_2,\ I_2 \le 0\ \right\}
\end{aligned}$$

$$\begin{aligned}
\text{Store}(\vec{I}) \;=\; & \{\text{B}(m) \mid m \ge 1,\ m \ge 2 - 2M + s_2 + I_2,\ m \le -2 + N,\\
& \qquad I_1 \ge 1 - s_1,\ 2 \le m + 2s_1 + 2I_1 - I_2 \le 1 + s_2,\ s_1 \ge 1\}\\
\cup\ & \{\text{B}(m) \mid m \ge 1,\ s_1 \ge 1,\ m \le -2 + N,\ I_1 \le -1 + M,\ m \le 1 - 2M + s_2 + I_2,\\
& \qquad m \ge 2 - 2s_1 - 2I_1 + I_2,\ I_1 \ge 1 - s_1,\ M \ge 1,\ m \ge 2 - 2M + I_2\}\\
\cup\ & \{\text{A}(m) \mid m \ge 1,\ m \ge 1 - 2M + s_2 + I_2,\ m \le -2 + N,\\
& \qquad I_1 \ge 1 - s_1,\ 1 \le m + 2s_1 + 2I_1 - I_2 \le s_2,\ s_1 \ge 1\}\\
\cup\ & \{\text{A}(m) \mid m \ge 1,\ s_1 \ge 1,\ m \le -2 + N,\ I_1 \le -1 + M,\ m \le -2M + s_2 + I_2,\\
& \qquad m \ge 1 - 2s_1 - 2I_1 + I_2,\ I_1 \ge 1 - s_1,\ M \ge 1,\ m \ge 1 - 2M + I_2\}
\end{aligned}$$

The fact that the array `B` appears in the Store set may be surprising as `B` is recomputed in each tile strip (this is why it does not appear in the Load set). This is because the script of Figure 2.6 considers each tile strip in isolation. To be able to remove `B` from the Store set, one would need a similar analysis on tile strips to discover that `B` is actually overwritten by subsequent tile strips. Then, only the last tile strip should store `B`, in case it is live-out of the program.

It can be checked (e.g., with `iscc`) that the set Load($\vec{I}$) above is indeed a generalization of the set Load($\vec{T}$) derived earlier for the canonical tiling with $\vec{s} = (2, 3)$. It is the complete expression, parameterized by $\vec{s}$, of all cases, including incomplete tiles, and even tilings obtained by translation of $\mathcal{L}$. Note that simply changing the object `Strip` (see Figure 2.6) from `{[I_1,I_2]->[I_1,I_2']}` to `{[I_1,I_2]->[I_1',I_2']}` gives 2D inter-tile reuse, i.e., in the whole space, as the first dimension is not a fixed parameter anymore. The strict order $\prec_{\vec{s}}$ is defined by `TiledPrev` while `Load` and `Store`, at the end of the script, express Equations (2.3) and (2.4). Constraints on parameters or on $\vec{I}$ can be added in `Params`, e.g., to get simplified Load/Store sets for complete tiles, for large tiles, etc. Note however that `isl` uses coalescing heuristics to simplify expressions and, depending on the constraints, the outcome can be simpler or more complicated (although equivalent). Here, replacing $s_1 \ge 0$ by $s_1 > 0$ changes the final expression.

```
# Inputs
Params := [M, N, s_1, s_2] -> { : s_1 >= 0 and s_2 >= 0 };
Domain := [M, N] -> { # Iteration domains
  S_1[i_1, i_2] : 1 <= i_2 <= N-2 and 0 <= i_1 <= M-1;
  S_2[i_1, i_2] : 1 <= i_2 <= N-2 and 0 <= i_1 <= M-1; } * Params;
Read := [M, N] -> { # Read access functions
  S_1[i_1, i_2] -> A[m] : -1 + i_2 <= m <= 1 + i_2;
  S_2[i_1, i_2] -> B[i_2]; } * Domain;
Write := [M, N] -> { # Write access functions
  S_1[i_1, i_2] -> B[i_2];
  S_2[i_1, i_2] -> A[i_2]; } * Domain;
Theta := [M, N] -> { # Preliminary mapping
  S_1[i_1, i_2] -> [i_1, 2 i_1 + i_2, 0];
  S_2[i_1, i_2] -> [i_1, 1 + 2 i_1 + i_2, 1]; };

# Tools for set manipulations
Tiling := [s_1, s_2] -> { # Two dimensional tiling
  [I_1, I_2, i_1, i_2, k] -> [i_1, i_2, k] :
      I_1 <= i_1 < I_1 + s_1 and I_2 <= i_2 < I_2 + s_2 };
Coalesce := { [I_1, I_2] -> [I_1, I_2, i_1, i_2, k] };
Strip := { [I_1, I_2] -> [I_1, I_2'] };
Prev := { # Lexicographic order
  [I_1, I_2, i_1, i_2, k] -> [I_1, I_2, i_1', i_2', k'] :
      i_1' <= i_1 - 1 or (i_1' <= i_1 and i_2' <= i_2 - 1)
      or (i_1' <= i_1 and i_2' <= i_2 and k' <= k - 1) };
TiledPrev := [s_1, s_2] -> { # Special ``lexicographic'' order
  [I_1, I_2] -> [I_1', I_2'] : I_1' <= I_1 - s_1 or
      (I_1' <= I_1 and I_2' <= I_2 - s_2) } * Strip;
TiledNext := TiledPrev^-1;
TiledRead  := Tiling.(Theta^-1).Read;
TiledWrite := Tiling.(Theta^-1).Write;

# Set/relation computations
In := Coalesce.(TiledRead - (Prev.TiledWrite)); Out := Coalesce.TiledWrite;
Load := In - ((TiledPrev.In) + (TiledPrev.Out));
Store := Out - (TiledNext.Out);
print coalesce (Load % Params); print coalesce (Store % Params);
```

Figure 2.6 – Script `iscc` for the Jacobi1D example.

To prove that we can use $\prec_{\vec{s}}$ (in Equations (2.3) and (2.4)) instead of $\sqsubset_{\vec{s}}$ (in Equations (2.1) and (2.2)), we define the concept of *pointwise functions*. This is a bit more than what we need for the proofs, but this concept makes easier to understand the underlying problems, related to the equality (or not) of some unions of images of sets, which will be even more subtle when dealing with approximations.

2.3.2 Pointwise Functions

If $\mathcal{A}$ is a set, $\mathcal{P}(\mathcal{A})$ denotes the set of subsets of $\mathcal{A}$ (sometimes also written $2^{\mathcal{A}}$). Hereafter, the function F is typically a function such as Out, which maps a tile, i.e., a subset of the tile strip ($\mathcal{A}$), to a subset of all data elements ($\mathcal{B}$).

Definition 3 (Pointwise function)**.** *Let $\mathcal{A}$ and $\mathcal{B}$ be two sets, and $\mathcal{C} \subseteq \mathcal{P}(\mathcal{A})$ be a collection of subsets of $\mathcal{A}$. The function $F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$ is **pointwise** iff there exists a function $f : \mathcal{A} \to \mathcal{P}(\mathcal{B})$ such that $\forall X \in \mathcal{C}$, $F(X) = \bigcup_{x \in X} f(x)$.*

In other words, a function F is pointwise if the image of any set where F is defined (not necessarily all sets) can be summarized by the contributions (through f) of the points it contains. In our case, $\mathcal{A}$ is the set of iterations in the tile strip to be analyzed and $\mathcal{C}$ is the set of all tiles (aligned or unaligned) intersected with $\mathcal{A}$.

If all written values are live-out, then $\text{Out}(\vec{I}) = \text{Write}(\vec{I})$, the values written in $\vec{I}$. Otherwise, this set should be intersected with Liveout, the set of all elements live-out of the tile strip. The function Write is, by definition, pointwise, because it is the union, for all points $\vec{i}$ in $\vec{I}$, of the set of values write($\vec{i}$) written at iteration $\vec{i}$. Also, even if the function $\vec{I} \mapsto \text{In}(\vec{I})$ may not be pointwise, any element read but not written in $\vec{I}$ is live-in for $\vec{I}$, thus $(\text{In} \cup \text{Write})(\vec{I}) = (\text{Read} \cup \text{Write})(\vec{I})$, which is pointwise, by introducing read($\vec{i}$) the set of points read at iteration $\vec{i}$. We get:

$$\begin{aligned}
\text{Load}(\vec{I}) &= \text{In}(\vec{I}) \setminus (\text{In} \cup \text{Write})(\vec{I}' \sqsubset_{\vec{s}} \vec{I}) = \text{In}(\vec{I}) \setminus \bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} \bigcup_{\vec{i} \in \vec{I}'} (\text{read} \cup \text{write})(\vec{i}) \\
&= \text{In}(\vec{I}) \setminus \bigcup_{\vec{I}' \prec_{\vec{s}} \vec{I}} \bigcup_{\vec{i} \in \vec{I}'} (\text{read} \cup \text{write})(\vec{i}) = \text{In}(\vec{I}) \setminus (\text{In} \cup \text{Write})(\vec{I}' \prec_{\vec{s}} \vec{I})
\end{aligned}$$

This is because $\cup_{\vec{I}' \prec_{\vec{s}} \vec{I}} \vec{I}' = \cup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} \vec{I}'$. Indeed, since all tiles aligned with $\vec{I}$ form a partition of $\mathcal{A}$, the points covered by the two unions are the same: these are all the points executed before any point in $\vec{I}$. The same is true for Store($\vec{I}$), which is equal to Liveout$\cap$(Write($\vec{I}$)$\setminus$ Write($\vec{I}' \sqsupset_{\vec{s}} \vec{I}$)), or equivalently equal to Liveout $\cap$ (Write($\vec{I}$) $\setminus$ Write($\vec{I}' \succ_{\vec{s}} \vec{I}$)). This concludes the proof in the exact case.

In summary, because tiles represent points exactly and because the "happens-before" relation (the fact that a point, resp. a tile, happens, during tiled execution, before another point, resp. tile) can be represented by a piece-wise affine relation, it is possible to perform a parametric analysis of inter-tile data reuse.

The equality of the unions of the images for $\vec{I}' \sqsubset_{\vec{s}} \vec{I}$ and for $\vec{I}' \prec_{\vec{s}} \vec{I}$ is actually a general property, and even a characterization, of pointwise functions. Indeed, as Theorem 2

hereafter shows, pointwise functions are exactly those that induce the desired "stability" property on union of sets, i.e., if two unions of sets cover the same points, then the union of their contributions through F are the same. This a more general property than *distributive functions* (for $\cup$), those for which $F(A \cup B) = F(A) \cup F(B)$ because, in our case, $F(A \cup B)$ may not be defined.

Before proving this theorem, we prove a third equivalent characterization, which explicitly builds a function f for a pointwise function F. If F and G are from $\mathcal{C}$ to $\mathcal{P}(\mathcal{B})$, we write $F \subseteq G$ if $\forall X \in \mathcal{C}$, $F(X) \subseteq G(X)$. Theorem 1 below also identifies the "largest" pointwise under-approximation of F.

Theorem 1. *For $F : \mathcal{C} \subseteq \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{B})$, let $F_\circ$ be the pointwise function defined from $f_\circ(x) = \bigcap_{Y \in \mathcal{C},\, x \in Y} F(Y)$. Then $F_\circ$ is the largest pointwise under-approximation of F, i.e., $F_\circ \subseteq F$ and, if F' is pointwise, $F' \subseteq F \Rightarrow F' \subseteq F_\circ$. In particular, F is pointwise if and only if $F = F_\circ$.*

Proof. Let $X \in \mathcal{C}$ and $y \in F_\circ(X) = \cup_{x \in X} f_\circ(x)$: $\exists x_y \in X$ such that $y \in f_\circ(x_y)$. With $Y = X$ in the definition of $f_\circ$, we get $f_\circ(x_y) \subseteq F(X)$, thus $y \in F(X)$, and $F_\circ \subseteq F$. If F' is pointwise and $F' \subseteq F$, then $f'(x) \in F'(Y) \subseteq F(Y)$ for all $Y \in \mathcal{C}$ such that $x \in Y$. Thus $f'(x) \subseteq f_\circ(x)$ by definition of $f_\circ$. Finally, if the function F is pointwise, $F \subseteq F_\circ$, thus $F = F_\circ$ since $F_\circ \subseteq F$. Conversely, if $F = F_\circ$, F is pointwise with $f_\circ$. □

We can now use this characterization to prove the following theorem:

Theorem 2. *$F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$ is pointwise if and only if $\forall \mathcal{C}' \subseteq \mathcal{C}$, $\forall \mathcal{C}'' \subseteq \mathcal{C}$, $\bigcup_{X \in \mathcal{C}'} X = \bigcup_{X \in \mathcal{C}''} X \Rightarrow \bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}''} F(X)$.*

Proof. Let $A = \bigcup_{X \in \mathcal{C}'} X$ and $B = \bigcup_{X \in \mathcal{C}''} X$. If the function F is pointwise, $\bigcup_{X \in \mathcal{C}'} F(X) = \bigcup_{X \in \mathcal{C}'} \bigcup_{x \in X} f(x) = \bigcup_{x \in A} f(x)$, and the same for B. Thus, if $A = B$, the two unions are also equal.

Now suppose that F is not pointwise. Theorem 1 shows that there exist $X \in \mathcal{C}$ and $y \in F(X) \setminus F_\circ(X)$, where $F_\circ(X) = \bigcup_{x \in X} \bigcap_{Y \in \mathcal{C}, x \in Y} F(Y)$, i.e., $\forall x \in X$, $\exists Y_x \in \mathcal{C}$ such that $x \in Y_x$ and $y \notin F(Y_x)$. By construction, $X \subseteq \bigcup_{x \in X} Y_x$ thus $\bigcup_{x \in X} Y_x = X \cup (\bigcup_{x \in X} Y_x)$. But $y \notin \bigcup_{x \in X} F(Y_x)$ while $y \in F(X)$ thus $y \in F(X) \cup (\bigcup_{x \in X} F(Y_x))$, contradiction. □

Note that the previous property on unions is equivalent to $\forall X \in \mathcal{C}$, $\forall \mathcal{C}' \subseteq \mathcal{C}$, $X \subseteq \bigcup_{X' \in \mathcal{C}'} X' \Rightarrow F(X) \subseteq \bigcup_{X' \in \mathcal{C}'} F(X')$, i.e., if a set is covered by a union of sets, then its image is contained in the union of the images of these sets.

To get the intuition for these different concepts, it is simpler to consider objects more general than rectangular tiles. Let $\mathcal{C}$ be the set of all possible "double squares" (in 2D)

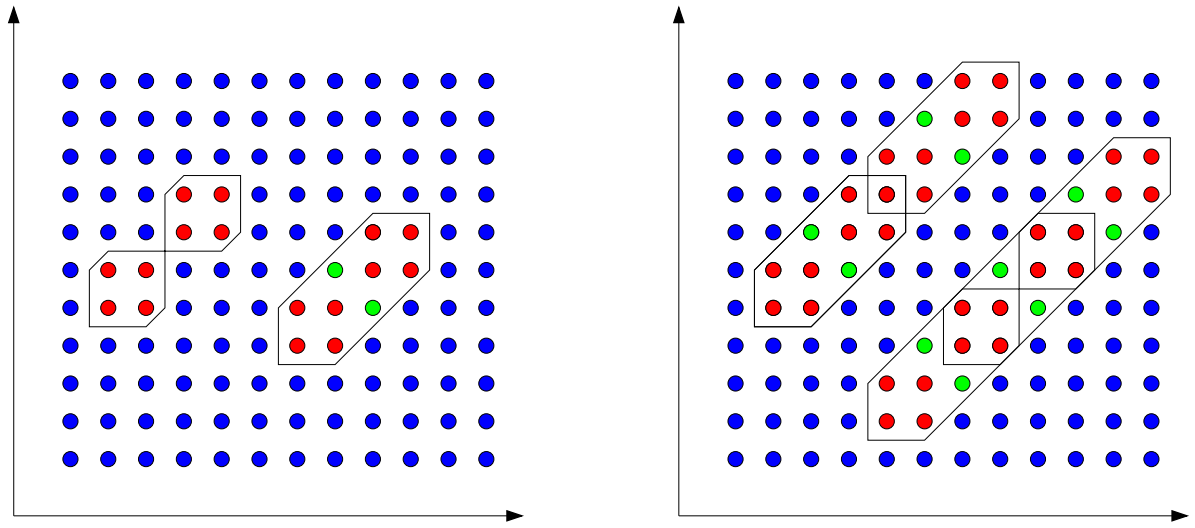


Figure 2.7 – "Double squares" (red), F (image of red & green), non pointwise situations.

defined as two diagonally-neighboring squares as depicted on the left of Figure 2.7 (red points in two boxes). Suppose each point $\vec{i}$ has an image $f(\vec{i})$. If $F(\vec{I})$ is defined for a "double-square" $\vec{I}$ as the union of all $f(\vec{i})$ for $\vec{i} \in \vec{I}$, it is pointwise by definition. Now, suppose $F(\vec{I})$ is defined as the union of all $f(\vec{i})$ for $\vec{i}$ in the convex hull of $\vec{I}$ (red + green points). The first situation on the right of Figure 2.7 shows that each point $\vec{i}$ is included in two "double-squares" whose images by F have only $f(\vec{i})$ in common. Thus F_0 is not equal to F (the image of green points are missing) unless f has some additional property and, according to Theorem 1, F is not pointwise. The second situation on the right of Figure 2.7 shows that a "double-square" is fully contained in two "double-squares", but the image of its green points (if f is injective) is not covered by the image of these two "double-squares" so, according to Theorem 2, F is not pointwise.

2.3.3 The Case of Approximations

We will use the previous properties of pointwise functions for approximations. There are at least four reasons why approximations of the various sets In, Out, Load, and Store may be used in an automatic code analyzer and optimizer.

- The execution of S at iteration $\vec{i}$ is not guaranteed, for example when it depends on a non-analyzable (e.g., data-dependent) `if` condition.
- The access functions are not fully analyzable (e.g., indirect accesses).
- The In/Out sets are approximated on purpose (e.g., they are restricted to polyhedra or hyper-rectangles) due to the algorithms used for analysis.

- The Load/Store sets are approximated to make them simpler, or to get transfer sets of some special form (e.g., vector/array communications).

In the first two cases, the approximation is pointwise, so the Read/Write functions remain pointwise. In the last two cases, it is more likely that $\mathrm{In} \cup \mathrm{Out}$ is not pointwise anymore. We first recall and extend the principles stated in [4] for approximations, assuming that the sets $\overline{\mathrm{In}}$, $\underline{\mathrm{Out}}$, and $\overline{\mathrm{Out}}$ are given such that $\mathrm{In}(\vec{I}) \subseteq \overline{\mathrm{In}}(\vec{I})$ and $\underline{\mathrm{Out}}(\vec{I}) \subseteq \mathrm{Out}(\vec{I}) \subseteq \overline{\mathrm{Out}}(\vec{I})$. Here, the under-approximations (that could benefit from [24, 72]) are not used for correctness, only for accuracy.

Non-Parametric Case.

The first step is to define the Store sets, as exactly as possible from the $\overline{\mathrm{Out}}$ sets, i.e., the sets of data possibly written:

$$\mathrm{Store}(\vec{I}) \;=\; \mathrm{Liveout} \cap (\overline{\mathrm{Out}}(\vec{I}) \setminus \overline{\mathrm{Out}}(\vec{I'} \sqsupset_{\vec{s}} \vec{I})) \tag{2.5}$$

Then, any over-approximation $\overline{\mathrm{Store}}(\vec{I})$ of $\mathrm{Store}(\vec{I})$ can be used. Equation (2.5) means that a possibly-defined element is always stored to remote memory, in case it is indeed written at runtime. But what if this is not the case? We add it to the set of input elements so that its initial value is stored back instead of garbage:

$$\overline{\mathrm{In}}'(\vec{I}) \;=\; \overline{\mathrm{In}}(\vec{I}) \cup (\overline{\mathrm{Store}}(\vec{I}) \setminus \underline{\mathrm{Out}}(\vec{I})) \tag{2.6}$$

Following [4, Thm. 3], loads are defined, as exactly as possible, from the sets $\underline{\mathrm{Out}}$, $\overline{\mathrm{Out}}$, and $\overline{\mathrm{In}}'$ (i.e., after $\overline{\mathrm{Store}}$ is defined). They are valid if for any tile $\vec{I}$:

$$\mathrm{Load}(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}) \text{ contains } \overline{\mathrm{Ra}}(\vec{I}) = \overline{\mathrm{In}}'(\vec{I}) \setminus \underline{\mathrm{Out}}(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) \tag{2.7}$$

$$\mathrm{Load}(\vec{I}) \cap \overline{\mathrm{Out}}(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) = \emptyset \tag{2.8}$$

Equation (2.7) means that all data possibly defined outside of the tile strip – the remote accesses $\overline{\mathrm{Ra}}(\vec{I})$ – have to be loaded before $\vec{I}$. Equation (2.8) means that data possibly defined earlier in the tile strip should not be loaded, as this could overwrite some valid data. Equation (2.9) below gives a non-recursive definition of $\mathrm{Load}(\vec{I})$, simpler (and more usable) than the formula of [4, Thm. 6] (although it is equivalent):

$$\mathrm{Load}(\vec{I}) \;=\; \overline{\mathrm{Ra}}_{\vec{I}} \cap ((\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I}) \setminus (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I'} \sqsubset_{\vec{s}} \vec{I})) \tag{2.9}$$

where $\overline{\mathrm{Ra}}_{\vec{I}}$ denotes all remote accesses for the tile strip w.r.t. $\vec{I}$, i.e., the union of all $\overline{\mathrm{Ra}}(\vec{I'})$, as defined in Equation (2.7), for all $\vec{I'}$ that belong to the same tiling as $\vec{I}$. The

mechanism of Equation (2.9) is actually simple: unlike for the exact case, a remote access live-in for $\vec{I}$ (i.e., in $\overline{\mathrm{In}}'(\vec{I})$) cannot be loaded just before $\vec{I}$ if it *may* be written earlier (i.e., in $\overline{\mathrm{Out}}(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$). Otherwise, the load will erase the right value if, at runtime, it was indeed written earlier. Instead, the trick is to load the element before the first tile $\vec{I'}$ that may write it. This way, either the value is defined locally and the read in $\vec{I}$ gets this value, or it is not defined and the read gets the original value. Theorem 3 below states more formally the correctness and exactness of Equation (2.9). Then, any over-approximation $\overline{\mathrm{Load}}(\vec{I})$ of this "exact" $\mathrm{Load}(\vec{I})$ can be used (even if it may induce some useless loads) as long as it still satisfies $\overline{\mathrm{Load}}(\vec{I}) \cap \overline{\mathrm{Out}}(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) = \emptyset$, as required by Equation (2.8).

To make notations simpler, we write ΔF the function defined from F by $\Delta F(\vec{I}) = F(\vec{I}) \setminus F(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$. Then, with $F = \overline{\mathrm{In}}' \cup \overline{\mathrm{Out}}$, we get $\mathrm{Load}(\vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap \Delta F(\vec{J})$ for all $\vec{J}$ aligned with $\vec{I}$. Also, by induction, for all $\vec{I}$, $\Delta F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}) = F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I})$ (but the first one is a disjoint union) and, similarly, $\Delta F(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) = F(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$. This implies the recursive relation $\Delta F(\vec{I}) = F(\vec{I}) \setminus \Delta F(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$. Also, $\Delta F(\vec{I}) = F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}) \setminus F(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$.

Theorem 3. *Equation (2.9) defines valid loads, which are "exact" w.r.t. the $\overline{In}'$, $\underline{Out}$, and $\overline{Out}$ sets (no useless or redundant loads) and performed as late as possible.*

Proof. We first prove that the loads are valid. First, Equation (2.8) is satisfied since $\overline{\mathrm{Out}}(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$ is subtracted in Equation (2.9). As mentioned above, by defining $F = \overline{\mathrm{In}}' \cup \overline{\mathrm{Out}}$, we get $\mathrm{Load}(\vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap \Delta F(\vec{J})$ for all $\vec{J}$ aligned with $\vec{I}$, and consequently $\mathrm{Load}(\vec{J'} \sqsubseteq_{\vec{s}} \vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap \Delta F(\vec{J'} \sqsubseteq_{\vec{s}} \vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap F(\vec{J'} \sqsubseteq_{\vec{s}} \vec{J})$. As $\overline{\mathrm{Ra}}(\vec{J}) \subseteq \overline{\mathrm{Ra}}_{\vec{I}}$ and $\overline{\mathrm{Ra}}(\vec{J}) \subseteq \overline{\mathrm{In}}'(\vec{J}) \subseteq F(\vec{J})$, then $\overline{\mathrm{Ra}}(\vec{J}) \subseteq \overline{\mathrm{Ra}}_{\vec{I}} \cap F(\vec{J'} \sqsubseteq_{\vec{s}} \vec{J})$, thus Equation (2.7) is satisfied too. Note that the intersection with $\overline{\mathrm{Ra}}_{\vec{I}}$ in $\mathrm{Load}(\vec{I})$ is not needed for correctness but it makes sure there are no useless loads. Also, $\mathrm{Load}(\vec{J}) = \overline{\mathrm{Ra}}_{\vec{I}} \cap (F(\vec{J}) \setminus \Delta F(\vec{J'} \sqsubset_{\vec{s}} \vec{J})) = (\overline{\mathrm{Ra}}_{\vec{I}} \cap F(\vec{J})) \setminus \mathrm{Load}(\vec{J'} \sqsubset_{\vec{s}} \vec{J})$, thus there are no redundant loads. Finally, if $y \in \mathrm{Load}(\vec{J})$, either $y \in \overline{\mathrm{In}}'(\vec{J})$ and y must be loaded before $\vec{J}$ as it may be read in $\vec{J}$, or $y \in \overline{\mathrm{Out}}(\vec{J})$ and it cannot be loaded later or it will overwrite the value possibly written in $\vec{J}$. Loads are thus done as late as possible. □

Parametric Case.

Our goal is now to reformulate Equations (2.5) and (2.9) so that the Store and Load sets can be computed with the tile sizes $\vec{s}$ as parameter. Can we just replace the order $\sqsubseteq_{\vec{s}}$ by $\preceq_{\vec{s}}$ as in the exact case (Section 2.3.1)? No. Doing so may, in general, be incorrect, resulting in missing loads or stores for $\vec{I}$, if subtracting the contribution of unaligned tiles (i.e., those that will not be executed) remove additional elements. This is where pointwise functions come, again, into play.

The easy case is when approximations are at the level of iterations, i.e., the accesses of each iteration $\vec{i}$ are approximated with $\underline{\text{write}}(\vec{i}) \subseteq \text{write}(\vec{i}) \subseteq \overline{\text{write}}(\vec{i})$ and $\text{read}(\vec{i}) \subseteq \overline{\text{read}}(\vec{i})$, resulting in pointwise functions $\underline{\text{Write}}$, $\overline{\text{Write}}$, and $\overline{\text{Read}}$. If the sets $\overline{\text{Out}}$, $\overline{\text{In}}$, then Store are derived from $\overline{\text{Write}}$ and $\overline{\text{Read}}$ with no further approximation, then, as for the exact case, $\overline{\text{Out}}$ and $\overline{\text{In}}' \cup \overline{\text{Out}}$ are pointwise too. Thus, a Store($\vec{I}$) can be computed with Equation (2.5), in a parametric way, with $\succ_{\vec{s}}$ instead of $\sqsupset_{\vec{s}}$. The same is true for the central part of Load($\vec{I}$) in Equation (2.9) with $\prec_{\vec{s}}$ instead of $\sqsubset_{\vec{s}}$. It remains to compute $\overline{\text{Ra}}_{\vec{I}}$ from $\overline{\text{Ra}}(\vec{I}) = \overline{\text{In}}'(\vec{I}) \setminus \underline{\text{Out}}(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$. As the tiles in $\mathcal{L}$ cover the whole iteration space, $\overline{\text{Ra}}_{\vec{I}}$ is the set of all data that are maybe read (or written for stores) and possibly not written before, i.e., live-in for the tile strip, for the schedule induced by the tiling aligned with $\vec{I}$. But if the mapping θ used for tiling was considered legal with the same pointwise approximation of reads and writes, then any shifted tiling (with standard validity conditions) preserves anti, flow, and output dependences, thus $\overline{\text{Ra}}_{\vec{I}}$ does not depend on $\vec{I}$. It is even equal to the live-in data for the tile strip when considering the original order of the code and, thus, can be computed, independently on $\vec{s}$.

The previous approach can be used when Load/Store sets are computed "exactly" but from a pointwise approximation of accesses. We now consider the case where, in addition to this pointwise approximation, even the sets $\overline{\text{Out}}$, $\overline{\text{In}}$, $\overline{\text{Store}}$, and $\overline{\text{Load}}$ can be over-approximated further, for whatever reason. For example, $\overline{\text{Store}}(\vec{I})$ can contain data that are not even in $\overline{\text{Out}}$ or $\overline{\text{In}}$, and thus not remote in the strict sense. However, transfers still need to be correct. We first consider how to handle $\overline{\text{Out}}$ in Equation (2.5) and $\overline{\text{In}}' \cup \overline{\text{Out}}$ in Equation (2.9), which, *a priori*, have no reason to be pointwise. We deal with the computation of $\overline{\text{Ra}}_{\vec{I}}$ later.

We first mention an interesting intermediate situation that works with no further difficulties, even if the approximations are not pointwise. If a pointwise function F is over-approximated through its domain (the iterations) instead of its range (the data), i.e., $\overline{F}(\vec{I}) = F(\overline{\vec{I}})$ with $\vec{I} \subseteq \overline{\vec{I}}$, then it may be the case that, when computing the unions (either with $\sqsubset_{\vec{s}}$ or $\prec_{\vec{s}}$), no new iterations are added with the approximated domains. This is what happens with the approximated "double-squares" of Figure 2.7, typical from parallel tiles. Then $\overline{F}(\vec{I}' \sqsubset_{\vec{s}} \vec{I})$ equals:

$$\bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} \bigcup_{\vec{i} \in \overline{\vec{I}'}} f(\vec{i}) = \bigcup_{\vec{I}' \sqsubset_{\vec{s}} \vec{I}} \bigcup_{\vec{i} \in \vec{I}'} f(\vec{i}) = \bigcup_{\vec{I}' \prec_{\vec{s}} \vec{I}} \bigcup_{\vec{i} \in \vec{I}'} f(\vec{i}) = \bigcup_{\vec{I}' \prec_{\vec{s}} \vec{I}} \bigcup_{\vec{i} \in \overline{\vec{I}'}} f(\vec{i}) = \overline{F}(\vec{I}' \prec_{\vec{s}} \vec{I})$$

In this case, even without pointwise functions, parametric approximations can be designed, with a careful analysis of the "shape" (the sets $\overline{\vec{I}}$) of approximations. But, this situation does not cover the case where approximations are made in the range of F and

cannot be converted into approximations in the domain of F, as it is the case for pointwise functions. We now address this general case.

The key point for approximation is that loading earlier and storing later always keeps correctness. As noticed earlier, $\text{Load}(\vec{I})$ has the form $\overline{\text{Ra}}_{\vec{I}} \cap \Delta F(\vec{I})$ with $\Delta F(\vec{I}) = F(\vec{I}) \setminus F(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$, thus $\Delta F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}) = F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I})$. If we define F° pointwise such that $F \subseteq F^\circ$, then $\Delta F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}) \subseteq \Delta F^\circ(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I})$, i.e., possibly more data are loaded (but no load is delayed), thus the validity condition of Equation (2.7) is satisfied with $\overline{\text{Ra}}_{\vec{I}} \cap \Delta F^\circ$. The same is true for $\text{Store}(\vec{I})$ with $\sqsupseteq_{\vec{s}}$: possibly more data are stored but no store is advanced. Finally, Equation (2.8) is satisfied too as $\overline{\text{Out}}(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) \subseteq F(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) \subseteq F^\circ(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$, which is subtracted in ΔF°. Thus, such an over-approximation mechanism (making F bigger) is always valid.

Theorem 4 below shows how to build such a function F° with the additional property that loads in ΔF that correspond to "pointwise loads" are still loaded for the same tile with ΔF°, i.e., not earlier (thus with no lifetime increase). Indeed, the goal is to try to avoid the naive solution where all data are loaded (resp. stored) before (resp. after) the whole computation of the tile strip.

Theorem 4. *Let $\mathcal{C}$ be the set of all tiles of size $\vec{s}$ and $F : \mathcal{C} \to \mathcal{P}(\mathcal{B})$. Define F° by $F^\circ(\vec{I}) = \cup_{\vec{J},\, \vec{I} \in \vec{J}} F(\vec{J})$, where $\vec{I} \in \vec{J}$ means that $\vec{I}$ is in the tile with origin $\vec{J}$. Then $F \subseteq F^\circ$ and F° is pointwise. Moreover, if y is such that $\forall \vec{I},\ y \in F(\vec{I}) \Rightarrow y \in F_\circ(\vec{I})$ ($F_\circ$ is defined in Theorem 1), then $\forall \vec{I},\ y \in \Delta F^\circ(\vec{I}) \Rightarrow y \in \Delta F(\vec{I})$, i.e., over-approximating F by F° does not load "pointwise" elements earlier.*

Proof. Depending of the context, we use $\vec{I}$ to represent a point in $\mathbb{Z}^n$ but also the tile with origin $\vec{I}$. Of course $F \subseteq F^\circ$ since $\vec{I} \in \vec{I}$. Now, let $f^\circ : \mathbb{Z}^n \to \mathcal{P}(\mathcal{B})$ defined with $f^\circ(\vec{J}) = F(\vec{J} - \vec{s} + \vec{1})$: $\vec{J}$ is the opposite corner in the tile whose origin is $\vec{J} - \vec{s} + \vec{1}$. Then, $\forall \vec{I} \in \mathbb{Z}^n$, $\cup_{\vec{J} \in \vec{I}} f^\circ(\vec{J}) = \cup_{\vec{J} \in \vec{I}} F(\vec{J} - \vec{s} + \vec{1})$. But $\vec{J} \in \vec{I}$ iff $\vec{I} \in \vec{J'} = \vec{J} - \vec{s} + \vec{1}$. Thus, the previous union is equal to $\cup_{\vec{J'}, \vec{I} \in \vec{J'}} F(\vec{J'}) = F^\circ(I)$, i.e., F° is pointwise.

Now, suppose that for all $\vec{I}$, $y \in F(\vec{I}) \Rightarrow y \in F_\circ(\vec{I})$. If $y \in F^\circ(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I})$, which is equal to $\cup_{\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}} \cup_{\vec{J},\, \vec{I'} \in \vec{J}} F(J)$, then $y \in F(\vec{J})$ for some $\vec{J}$ and $\vec{I'}$ such that $\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}$, $\vec{I'} \in \vec{J}$. Thus $y \in F_\circ(\vec{J})$ and $y \in f_\circ(x)$ for some $x \in \vec{J}$ because $F_\circ$ is pointwise. Since $F_\circ \subseteq F$ and since the union of tiles $\cup_{\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}} \cup_{\vec{J},\, \vec{I'} \in \vec{J}} \vec{J}$ spans the same set of points as the union of tiles $\cup_{\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}} \vec{I'}$, this shows $y \in F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I})$. Remember that for any function G, $\Delta G(\vec{I}) = G(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}) \setminus G(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$. Thus if $y \in \Delta F^\circ(\vec{I})$, $y \in F^\circ(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I}) \setminus F^\circ(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$, which implies $y \in F(\vec{I'} \sqsubseteq_{\vec{s}} \vec{I})$ (as we just showed) and $y \notin F(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$ (because $F \subseteq F^\circ$). Thus $y \in \Delta F(\vec{I})$. □

The same technique can be used for the set $\mathrm{Store}(\vec{I})$ but with an expression such as $F^{\circ}(\vec{I}) = \cup_{\vec{J},\vec{J}\in\vec{I}} F(\vec{J})$. It remains to see what to do with the set $\overline{\mathrm{Ra}}_{\vec{I}}$. We can compute, with $\vec{s}$ as parameter, $\overline{\mathrm{Ra}}(\vec{I}) = \overline{\mathrm{In}}(\vec{I}) \setminus \underline{\mathrm{Out}}(\vec{I'} \prec_{\vec{s}} \vec{I})$, thus replacing $\sqsubset_{\vec{s}}$ by $\prec_{\vec{s}}$. We get *a priori* a smaller set, which could be problematic because of the intersection in Equation (2.9). However, it is still correct and, actually, even more precise. Indeed, as Out is exact, we have $\overline{\mathrm{In}}'(\vec{I}) \setminus \mathrm{Out}(\vec{I'} \sqsubset_{\vec{s}} \vec{I}) = \overline{\mathrm{In}}'(\vec{I}) \setminus \mathrm{Out}(\vec{I'} \prec_{\vec{s}} \vec{I})$ and what is actually important in Equation (2.7) is that this set is indeed loaded. Thus, considering $\overline{\mathrm{Ra}}(\vec{I}) = \overline{\mathrm{In}}(\vec{I}) \setminus \underline{\mathrm{Out}}(\vec{I'} \prec_{\vec{s}} \vec{I})$ in Equation (2.7) is fine as it is a superset. Finally, to compute $\overline{\mathrm{Ra}}_{\vec{I}} = \bigcup_{\vec{J},\vec{J}-\vec{I}\in\mathcal{L}} \overline{\mathrm{Ra}}(\vec{J})$, we drop the lattice constraint. If $\overline{\mathrm{Ra}}$ is not pointwise, we get a possibly larger set: this is suboptimal, but correct.

This completes the theory for parametric tiling with inter-tile reuse and approximations. In practice, it needs to be adapted to each approximation scheme but it still provides some general mathematical means to reason on the correctness of approximations for parametric tiling. A possible approximation (to reduce complexity) consists in removing, in all intermediate computations such as Out, Store, In′, all existential variables (projection) and to manipulate only integer points in polyhedra. Another possibility is to rely on array region analysis techniques [24]. This is left for future work. We point out however that generalizing such a parametric inter-tile reuse to more general tilings, where tiles (rectangular or not) are not executed following the axes that define them, will be more difficult if the iteration space covered by tiles that "happen before" a given tile cannot be defined by a piece-wise affine relation. One can still define approximations, even not necessarily pointwise, as long as $(\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I'} \prec_{\vec{s}} \vec{I}) = (\overline{\mathrm{In}}' \cup \overline{\mathrm{Out}})(\vec{I'} \sqsubset_{\vec{s}} \vec{I})$ (and similar equalities), as illustrated with the "double-squares" of Figure 2.7. However such approximations are more difficult to define systematically and may require unacceptable (i.e., too rough) additional over-approximations.

2.4 Conclusion

This work, first published at the IMPACT'14 workshop [26], then as an improved version at the CC'15 conference [27], provided the first parametric solution for generating memory transfers with data reuse when a kernel is offloaded to a distant accelerator, tile by tile after loop tiling, and when all intermediate results are stored locally on the accelerator. In this case, when a value has been loaded or defined in a previous tile, it is read from the local memory and not loaded from the remote memory, which is not yet up-to-date. Our solution is parametric in the sense that we can derive the copy-in/copy-out sets for each tile, exploiting both intra- and inter-tile data reuse, with tile sizes as parameters. Such a

result is quite surprising as parametric tiling is often considered as necessarily involving quadratic constraints, i.e., not analyzable within the polyhedral model. We solve it in an affine way with a different reasoning that considers, in the analysis, all (unaligned) possible tiles obtained by translation and not just the tiles of a given tiling. A similar technique can be used to parameterize the computations of local memory sizes, thanks to parametric lifetime analysis (to be described in Chapter 3) and array contraction with parametric modulos (to be described in Chapter 4) or simply bounded boxes, even for pipeline schedules similar to double buffering. The corresponding computations will be detailed in Chapter 5.

This reasoning can also be extended in the case of approximations, which are needed when dealing with kernels that are not fully affine, or because approximations of communications are desired for code simplicity, complexity issues, or architectural constraints (e.g., vector communication). The main difficulty with approximation is that, when some data can be both read and written, loading blindly from remote memory, in an over-approximate way, is not safe as it may not be up-to-date. We address the problem thanks to the introduction of the concept of pointwise functions, well suited to deal with unaligned tiles. This concept may be useful for other applications linked to extensions of the polyhedral model as it turns out to be fairly powerful. For the moment, our study provides the mathematical foundations to discuss the correctness of approximation techniques that still need to be designed, even if some simple schemes are already possible. The full implementation, from the analysis down to code generation, is still a development challenge. Full experiments will be needed to validate the approach and help designing cost models for tile size selection. Nevertheless, the different performance studies with inter-tile data reuse for GPUs [44, 45, 75] or FPGAs [5, 62], for non-parametric tile sizes, already demonstrate its interest.

"Guessing" the right size of the tiles can be laborious, especially when dealing with multi-level tiling and multi-level caches. The search space can become so wide that even iterative compilation might not be sufficient. As said, our parametric technique provides a direct expression of the copy-in/copy-out sets for each tile, and can then be used for performing array contraction on the accelerator still in a parametric fashion. It is only with such a parametric description that we can hope to design cost models for compile-time tile size selection in the context of tiling with inter-tile data reuse. Such static compilation techniques could then be integrated on top of intermediate languages such as OpenACC or OpenCL, or directly generate lower-level code, providing an automatic way to derive blocking algorithms for accelerators. Other applications are certainly possible, as soon as data reuse among tiles or pages has to be analyzed.

Chapter 3

Liveness Analysis over Parallel Specifications

Summary

In this chapter, we revisit scalar and array element-wise liveness analysis for programs with parallel specifications. In earlier work on memory allocation/contraction (register allocation or intra- and inter-array reuse in the polyhedral model), a notion of "time" or a total order among the iteration points was used to compute the liveness of values. In general, the execution of parallel programs is not a total order, and hence the notion of time is not applicable.

We first revise how conflicts are computed by using ideas from liveness analysis for register allocation, studying the structure of the corresponding conflict/interference graphs. Instead of considering the conflict between two live ranges, we only consider the conflict between a live range and a write. This simplifies the formulation from having four instances involved in the test down to three, and also improves the precision of the analysis in the general case.

Then we extend the liveness analysis to work with partial orders so that it can be applied to many different parallel languages/specifications with different forms of parallelism. An important result is that the complement of the conflict graph with partial orders is directly connected to memory reuse, even in presence of races. However, programs with conditionals do not always define a partial order, and our next step will be to handle such cases with more accuracy.

3.1 Motivation

As recalled in Section 1.1, modern processors are equipped with several levels of memory hierarchy to keep the data as close as possible to the processing units. Because the

locality of reference has significant impact on performance and energy consumption, efficiently utilizing storages at various levels—registers, caches, memories, and so on—has been a topic of many research.

One important analysis, common to many optimizations around storage, is liveness analysis. Live-ranges are used to determine if two values can share a same register and/or a memory location. It is also used to compute live-in/live-out sets, as well as to estimate memory footprint for predicting cache behaviors. Existing techniques [32, 54, 64, 78] mostly assume sequential execution, or only simple forms of parallelism, when computing live-ranges. In this chapter, we revisit liveness analysis for parallel programs, with the ambition to have a common framework suitable for all parallel specifications.

Our contribution is twofold: by analyzing and exploiting the structure of interferences (conflicts between live-ranges), we provide a more efficient analysis for the sequential case, which can be extended to handle some structured forms of parallel specifications (such as nesting of parallel and sequential loops), namely series-parallel graphs. We then provide a generic approach to handle parallel specifications, in particular those based on an happens-before partial order.

We first motivate our work by illustrating the difficulties with liveness analysis in Section 3.1.1 and recall in Section 3.1.2 the notion of *conflict* that we use in formulating the liveness. We then present simplifications to the computation of liveness inspired by register allocation methods in Section 3.2 and extend these algorithms to general parallel specifications in Section 3.3. Finally, we discuss some links to other storage mapping techniques in Section 3.4 and conclude in Section 3.5.

3.1.1 Liveness, Conflicts, and Reuse

We first introduce the readers to liveness analysis and memory reuse, and the difficulties that arise when we add complications such as parallelism. Register allocation and array contraction through intra-array reuse are two similar forms of memory reuse, the latter being a symbolic version of the first. Liveness analysis is here to make sure resource sharing does not change the semantics of the code. The simplest example of register allocation is the following:

```
x = ...;
y = x + ...;
... = y;
```

where the scalar `y` can reuse the memory element allocated to `x`, assuming `x` is never ever used later. The last condition is important, as it enforces that the lifetime of `x` ends right

before the creation of `y`, thus allowing its reuse.

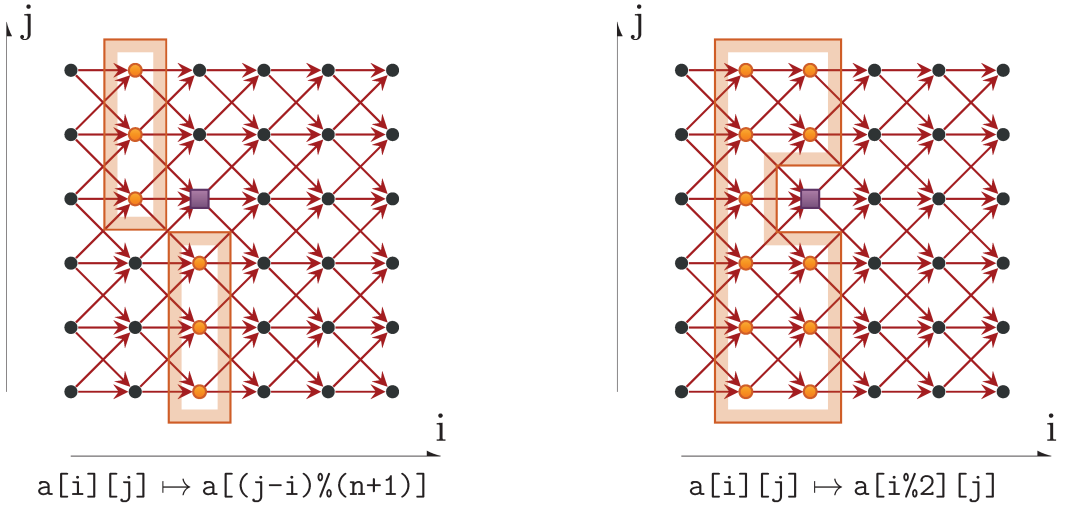
With loops and arrays, memory reuse becomes less straightforward. The same strategy applied on the following code

```
c[0] = 0;
for(i=0; i<n; ++i)
    c[i+1] = c[i] + ...;
```

can easily fold the `c[]` array into a single scalar `c` (thus simply removing the subscript). Again, this assumes that for all `i` (except potentially the last one) `c[i]` is never ever used later. Now, let us first consider nested loops and multi-dimensional arrays, then parallelism. For example, the following code requires a more precise analysis but with the same idea:

```
for(i=0; i<n; ++i)
    for(j=0; j<n; ++j)
        A[i][j] = A[i-1][j-1] + A[i-1,j] + A[i-1,j+1];
```

We cannot store a value (purple square in Figure 3.1a) in the same location as values that are still live (surrounded in orange). Those are said to be conflicting array elements. However, if we ignore live-in/live-out values, we can reuse any other ones. By reusing along the diagonal (see the mapping `a[i][j]` $\mapsto$ `a[(j-i)%(n+1)]` below the figure), a minimal allocation requiring only $n + 1$ cells instead of n^2 is obtained. This is the idea used in standard array contraction techniques [31, 33, 54, 64].



(a) Sequential.

(b) Parallel on `j`.

Figure 3.1 – Conflicts on jacobi-1d.

When we try to handle parallelism, we have to consider all potential conflicts that may happen in one of all the possible parallel executions. On the previous example, if the `j` loop is parallelized (see Figure 3.1b), we end up with additional conflicts due to parallel iterations being potentially past or future. The new mapping `a[i][j]` $\mapsto$ `a[i%2][j]` requires $2n$ cells, which is more than previously; as expected, increased parallelism comes at the cost of additional memory space.

All these examples can be handled by standard techniques, if the conflict analysis is computed with care. However, depending on the way the analysis is done (in the standard way, it implies 6 dimensions: 2 memory locations and 4 references, a write and a read accesses for each), it can be rather costly: we give in Section 3.2 two variants involving fewer dimensions. Moreover, there is a multitude of forms of parallelism—from software pipelining (see Figure 3.2a and Figure 3.2b, which will be detailed later on) to X10-like parallelism [66]—that are too complex to be modeled by such nested loops programs. They call for a more general framework based on a more general "happens-before" relation. Section 3.3 will extend this analysis to such a model. This will give us some new insights to re-interpret, with this new view, some previous works on memory reuse and possibly extend them (see Section 3.4).

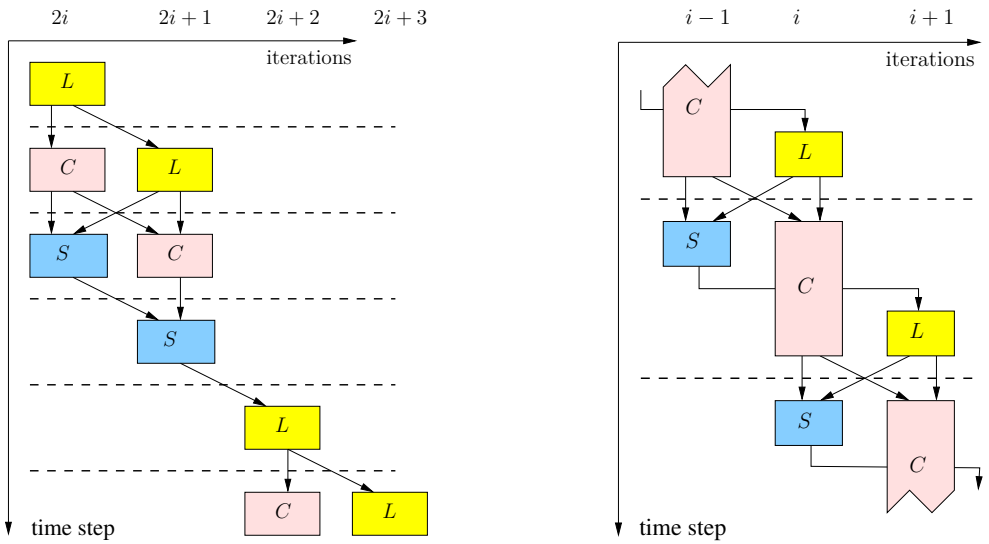


Figure 3.2 – Two software pipelines for kernel offloading, the first one borrowed from previous work on (non-parametric) inter-tile data reuse [5], the second one that we designed to exhibit better regularity and overlapping properties (see details in Chapter 5).

3.1.2 Simultaneously Live Indices

Lattice-based memory allocation [31, 32], as well as all prior work on intra-array reuse [33, 64, 54, 31], is based on the concept of "conflicting" (array) elements. The set of pairs of elements that should not be mapped to the same location is expressed as a binary relation denoted as $\bowtie$. It corresponds to the well-known interference graph in register allocation. It can also be used in other contexts such as for bank allocation or parallel accesses to memory [21, 32], or more generally whenever renewable resources need to be shared. In register allocation, vertices correspond to the scalar variables of the program, edges denote the fact that two variables should not be mapped to the same register, so that graph coloring can be used to derive a valid register assignment. For intra-array reuse, the variables are the array elements (but expressed in a symbolic way, not in an extensive way) and the edges are the pairs defined by the $\bowtie$ relation.

Actually, this view of register allocation is a bit simplistic and limited. In register allocation, instead of considering that a vertex corresponds to a variable name, variables can be renamed during their lifetime (what is called live-range splitting) and each live-range can be assigned a different register. The same is true when a variable is spilled (i.e., moved with a store operation from a register to memory) because it can come back from memory (with a load operation) in a different register. The situation is similar, although different, for $\bowtie$ and array elements. To reduce the complexity of the analysis and of the code rewriting necessary to express the allocation, we consider that an array element is live from its very first access until its very last access[1] and that it is mapped, in this time period, to the same memory location. But we could also cut its live-range into pieces, for example (but not only) distinguishing each live-range starting at a given write and ending at its last corresponding read (as done in exact data-flow dependence analysis as opposed to memory-based dependence analysis), and then map an array element to different memory locations, depending on the program control point. However, this makes the analysis much more complicated and, unlike register allocation where the number of registers is more limited, it is maybe not worth it for allocation in memory. Also, we will consider that there is a single level of memory, i.e., no value is ever spilled during its whole lifetime (this could be useful however, in particular when offloading data to a distant platform, but here we assume that such data movements are explicit in the code, i.e., the spilling has already been taken care of).

[1] In a correct code, the first access is a write. Otherwise, the value is live-in from the region being analyzed, so there is an implicit earlier write to bring it to its memory location. Similarly, the last access is a read otherwise it generates dead code or the value is actually live-out, which means there is an implicit read afterwards, to save it somewhere else.

According to Definition 1 by Darte et al. [32], two array elements identified by the vectors $\vec{m}_1$ and $\vec{m}_2$ conflict (denoted $\vec{m}_1 \bowtie \vec{m}_2$) if they are *simultaneously live* under a schedule θ. In their work, a schedule is a function (which can express parallelism) that assigns to each operation u a "virtual" execution time as an element of a totally ordered set $(\mathcal{T}, \preceq)$ [32]. This definition is kept quite general, as an input to the allocation problem: what an "operation" is, what "simultaneously live" means, and how the values of θ are interpreted is not precisely defined. These notions depend on the context of use and are mostly illustrated for the particular case of affine multi-dimensional schedules, as defined by Feautrier [38], i.e., functions from operations $u = (S, \vec{i})$ (pair statement, iteration) into $\mathbb{Z}^d$ (for some positive integer d) associated with the lexicographic order $\preceq$, that are affine with respect to $\vec{i}$. This defines an execution with inner parallelism in the following sense: if $\theta(S, \vec{i}) \prec \theta(T, \vec{j})$ then $(S, \vec{i})$ is executed strictly before $(T, \vec{j})$, while if $\theta(S, \vec{i}) = \theta(T, \vec{j})$ then both operations are done "in parallel". Again, what "in parallel" means depends on the implementation. In particular, one may need to define precisely how different accesses within a given operation are scheduled, e.g., reads and writes, as indicated by Darte et al. [31] (Footnote 2, Page 3). We come back to this situation later.

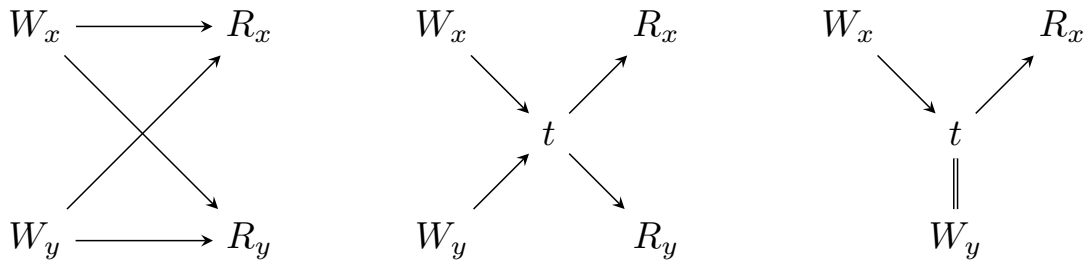
3.2 Special Case Extensions

We now describe how to define the relation $\bowtie$ in more general situations than multi-dimensional affine schedules. It was previously illustrated for quadratic schedules with two instructions [32], but more situations are of interest today. We first describe the "simpler" cases with sequential schedule and loop parallelism at any level (not just inner parallelism) that can still be handled as natural and incremental extensions of the classical analysis using live-ranges.

In Section 3.3, we further extend to other forms of parallelism such as software pipelining and parallel specifications with partial orders (happens-before relations) where such natural extensions are not directly applicable. Although the resulting method is more general, it may nevertheless be less efficient or expressive for handling special cases. For instance, it is not clear how to compute the minimal size of an allocation, or a lower bound of it, using clique computations when there is no notion of global time. Thus, it is still interesting to explore the limits of classical approaches as we do here.

3.2.1 Fully Sequential Schedules

For a fully sequential schedule, affine or not, all operations are done in some particular order, with no parallelism. Bee [2] uses this property to consider that $x \bowtie y$ iff the first write of x (respectively y) is before the last read of y (respectively x), thus creating a "butterfly" diagram shown in Figure 3.3a. Computing this $\bowtie$ relation was deemed rather costly as it required them to perform a crossproduct of QUASTs [35].



(a) Butterfly. (b) Crossproduct. (c) Triangle.

Figure 3.3 – Sequential strategies.

We can, instead, rely on the notion of (sequential) time step. This consideration gives a specific way of computing conflicting elements, similar to the liveness used for register allocation, using live sets. For each time step $t \in \mathcal{T}$, we can first identify the set $\text{Live}(t)$ of all values considered live at t, i.e., to be stored in memory "during" t. Then, all values live at t are conflicting with each other as shown in Figure 3.3b (they form a clique using graph terminology), i.e., the set of conflicting pairs is $\bigcup_{t\in\mathcal{T}} (\text{Live}(t) \times \text{Live}(t))$. As for register allocation, one needs to carefully define the liveness[2] with respect to θ. A memory location $\vec{m}$ is live at t if there are two operations[3], a write $(S, \vec{i})$ and a read $(T, \vec{j})$ of $\vec{m}$ with $\theta(S, \vec{i}) \preceq t \preceq \theta(T, \vec{j})$. The equality is necessary if one considers that a variable spanning a single time step requires storage. This depends on the granularity of the "operation". If one restricts the analysis to variables whose live-range spans at least two time steps, then one can define liveness with one strict inequality, e.g., with $\theta(S, \vec{i}) \prec t \preceq \theta(T, \vec{j})$, or consider the "program points" between two time steps. In back-end optimization, to avoid confusions, one distinguishes live-in and live-out variables for each program instruction.

[2]In back-end code optimizations, liveness is usually defined on control-flow graphs, with the notion of "program point", and live-in and live-out variables at these points. This should be kept in mind when defining liveness with "time steps". Whatever the formalization, what is important is to be able to specify when memory locations are booked or released.

[3]Again, following previous remarks/assumptions, one can just check the cases where $(S, \vec{i})$ is a write and $(T, \vec{j})$ a read.

Let us illustrate these different points with the example corresponding to Figure 3.1a. The code has a single statement. The sequential order defines a 2-dimensional schedule with $\theta(S, \vec{i}) = \vec{i}$ and the lexicographic order. However, this schedule does not specify the order of accesses inside an operation. If all reads are performed before the write, then one can add a dimension to the schedule, distinguishing reads and writes, with $\theta'(S, \vec{i}, R) = (\vec{i}, 0)$ for a read and $\theta'(S, \vec{i}, W) = (\vec{i}, 1)$ for a write, and then consider all 3-dimensional time steps. One can also proceed in an ad-hoc fashion, without this additional dimension, as follows. In general, to be always correct, one should consider all program points ("events") between any two accesses. This includes the program points between the reads and the writes of a given operation, in addition to those between operations. However, since reads precede writes in a given operation, it is sufficient to consider only the liveness at program points between operations (all conflicts are seen there). So, let us consider each time step $\vec{t}$—here in 2D, a vector $\vec{t} = (t_1, t_2)$—and let us interpret it as the program point just before the reads and writes scheduled at time step $\vec{t}$. Then:

$$\begin{aligned}\text{Live}(\vec{t}) = \quad & \{\vec{m} \mid \vec{i} \in \mathcal{D}_S, \vec{i} \prec \vec{t}, \vec{i} = \vec{m}\} \\ \cap \quad & \{\vec{m} \mid \vec{j} \in \mathcal{D}_S, \vec{t} \preceq \vec{j}, (\vec{j} - (1,1) = \vec{m} \\ & \text{or } \vec{j} - (1,0) = \vec{m} \text{ or } \vec{j} - (1,-1) = \vec{m})\}\end{aligned}$$

where $\mathcal{D}_S$ is the set of valid iterations for statement S, giving:

$$\begin{aligned}\text{Live}(\vec{t}) = \text{Live}((t_1, t_2)) = \\ \{(t_1, m_2) \mid 0 \le t_1 \le n-2,\ 0 \le m_2 \le n-1, m_2 \le t_2 - 1\} \cup \\ \{(t_1 - 1, m_2) \mid 1 \le t_1 \le n-1,\ 0 \le m_2 \le n-1, t_2 - 1 \le m_2\}\end{aligned}$$

as depicted in Figure 3.1a. Such computations can be done with the iscc calculator [74], provided with the barvinok library [76], with the following script:

```
# Inputs
Domain := [n] -> { S[i,j] : 0 <= i, j < n };
Read := [n] -> { S[i,j] -> A[i-1,j-1];  S[i,j] -> A[i-1,j];
                 S[i,j] -> A[i-1,j+1] } * Domain;
Write := [n] -> { S[i,j] -> A[i,j] } * Domain;
Sched := [n] -> { S[i,j] -> [i,j] };

# Operators
Prev := { [i,j]->[k,l]: i<k or (i=k and j<l) };
Preveq := { [i,j]->[k,l]: i<k or (i=k and j<=l) };
```

```
WriteBeforeT := (Prev^-1).(Sched^-1).Write;
ReadAfterT := Preveq.(Sched^-1).Read;

# Liveness and conflicts
Live := WriteBeforeT * ReadAfterT;
Conflict := (Live^-1).Live;
Delta := deltas Conflict;
```

In this script, the set `Live`—a map from time indices $\vec{t}$ to array elements $A[\vec{m}]$—is built as previously described. The set `Conflict` is then defined as `Live^-1.Live`, which directly builds the union, for all time steps $\vec{t}$, of the pairs of array elements live at the same time step. It corresponds to the composition (join) of the map $A[\vec{m}'] \to \vec{t}'$ with the map $\vec{t} \to A[\vec{m}]$, i.e., with $\vec{t}' = \vec{t}$. Note that, with this construction, an array element conflicts with itself, thus $\vec{0}$ is a conflicting difference. Then `Delta` gives the set of conflicting differences, which can be used for memory mapping:

$$\begin{aligned}\text{Delta}(n) = \; & \{(1, i_1) \mid i_1 \leq 0,\, n \geq 3,\, i_1 \geq 1 - n\} \cup \\ & \{(0, i_1) \mid i_1 \geq 1 - n,\, n \geq 2,\, i_1 \leq -1 + n\} \cup \\ & \{(-1, i_1) \mid i_1 \geq 0,\, n \geq 3,\, i_1 \leq -1 + n\}\end{aligned}$$

From this set, one can infer, using modulo allocation techniques, that the mapping $A[i, j] \mapsto A'[j - i \bmod (n + 1)]$ of size $n + 1$ is correct. Computing the cardinal of the `Live` set at any time step (with the iscc operation `card`) gives a maximum size of $n + 1$ for $n \geq 3$, which proves the optimality of this mapping, as claimed in Section 3.1.1.

If one wants to keep at all time the information on the time step $\vec{t}$, an alternative method can be used as follows:

```
# Other solution, with liveness for each time step
CLive := Live cross Live;
EqualMap := domain_map identity domain Live;
DeltaMap := deltas_map ((range Read)->(range Read));
TConflict := (EqualMap^-1).CLive;
TDelta := TConflict.DeltaMap;
```

The set `CLive` has type $[\vec{t} \to \vec{t}'] \to [A[\vec{m}] \to A[\vec{m}']]$. Then, the set `TConflict` has type $\vec{t} \to [A[\vec{m}] \to A[\vec{m}']]$ and gives, for a given time step $\vec{t}$, the set of pairs of array elements $A[\vec{m}]$ and $A[\vec{m}']$ live at $\vec{t}$. Finally, the set `TDelta` gives the set of conflicting differences for a given time step $\vec{t}$. Its range should give the same conflicting differences as the set `Delta` computed before.

Finally, to be complete, one should also consider live-in and live-out array elements. Unless they are stored in a different array, live-out array elements must be specified by the context and added to the previous analysis as reads after any time step. They can be computed as we now explain.

```
ReadAfterT := Preveq.(Sched^-1).Read + ((range Sched) -> LiveOut);
```

Similarly, unless all values defined by the kernel are stored in a fresh temporary array, live-in array elements must be computed and integrated in the set of values written before any time step. This can be done as follows:

```
SchedPrev := Sched.(Prev^-1).(Sched^-1);
LiveIn := range(Read - SchedPrev.Write));
WriteBeforeT := (Prev^-1).(Sched^-1).Write + ((range Sched) -> LiveIn);
```

The cross-product of `Live` is the most expensive operation. In particular, if `Live` is composed of a union of polyhedra, the result will have many polyhedra due to the disjunctive expansion. To mitigate the problem, simplifying the expression using heuristics (provided by `isl` [73] in our case) is particularly efficient but is, in itself, expensive too. A slightly different approach is to apply the same strategy than used for register allocation itself: two memory elements conflict if and only if one is live at the definition (write) of the other. This strategy, depicted in Figure 3.3c, can be implemented in the following way:

```
WriteBeforeT := (Preveq^-1).(Sched^-1).Write;
ReadAfterT := Prev.(Sched^-1).Read;
WriteAtT := (Sched^-1).Write;
Live := WriteBeforeT * ReadAfterT;
Conflict := (Live^-1).WriteAtT;
AsymDelta := deltas Conflict;
Sym := { A[i,j] -> A[i,j]; A[i,j] -> A[-i,-j] };
Delta2 := Sym(AsymDelta);
```

Notice that time step consideration changed. We are interested in conflicts produced by a `Write`, i.e., conflicts that exist at the time step after the `Write`. Thus, compared to the previous script, we compute conflicts one step earlier by shifting the `Live` range into the future. This consists in swapping `Prev` and `Preveq` in the equations, with no computational overhead. However, the computed `Conflict` is now potentially asymmetric. We make it symmetric in the end to be consistent with the previous method. While this operation is purely syntactic, thus not costly by itself, it may lead the `isl` library

to compute expensive disjoint unions, which it prefers. However, this did not have any significant impact on tested examples. Also, it is not required if the following uses of the analysis do not require symmetry.

Furthermore, the switch to this method comes with an additional benefit in the case where we accept undetermined control flow. Indeed, as for register allocation, we might eliminate conflicts that were inconsistent. A standard example is what we call the "double diamond" case:

```
if(...) then x = ...; else y = ...;
if(...) then ... = x; else ... = y;
```

Here, live-ranges of x and y technically interfere right in between the `if`s. However, there is no valid execution where both are live at the same time (unless the program is incorrect on purpose), so they can share the same register. In fact, the only executions that make sense are the ones where the same branch is taken in both `if`s, otherwise the variables are used without being defined. Our write-based analysis will consider that there is no conflict as, indeed, none of them is written while the other is live. Only one of them can ever be written (writes are in separate branch of the same `if`).

Undetermined control flow introduces many more problems [22] and is not the focus of this chapter. However, the framework that we develop in Section 3.3 must consider that this situation may exist. This will allow us to handle the case of unaligned tiles (following the terminology of Chapter 2), which cannot be both executed in a given execution trace, as is the case of the two branches of a given `if`.

3.2.2 Affine Schedules and Parallel Loops

Now, consider the same example but with the innermost loop marked as parallel (see Figure 3.1b). This corresponds to the schedule $(i, j) \mapsto i$. In this code, all iterations of the j loop can run in parallel, however several semantics are possible. If the parallel loop is a Fortran-like FORALL loop, all reads of the j loop occur before any write. In this case, there is still a notion of "time step", actually, each iteration of the i loop corresponds to two time steps: a step with all reads for the different values of j and a step with all writes for the different values of j. The liveness can then be computed with the same principle as for a sequential code, with either the Live $\times$ Live approach or the Live $\times$ Write approach, exposed in Section 3.2.1. The only difference is the definition of the `Prev` and `Preveq` relations that depend on the schedule dimension:

```
Prev := { [i,j] -> [k,l]: i < k}; Preveq := { [i,j] -> [k,l]: i <= k};
```

With these definitions, we get that the set of live values at time step (k, l) is the full column $A[k-1, *]$ for $0 < k < n$, and only one column of A is needed if array contraction is performed. However, with a more general parallel loop semantics, and without any information on the order of parallel accesses, reads and writes of different iterations of the j loop should be considered as possibly running concurrently. In other words, a safe definition of liveness is with `Preveq` instead of `Prev` in the definition of `Live`:

```
WriteBeforeT := (Preveq^-1).(Sched^-1).Write;
ReadAfterT := Preveq.(Sched^-1).Read;
```

With this modification, we find that the live values are two successive columns of A, which is the expected set described in Figure 3.1b (and also the expected size of the contracted array). Indeed, when a value is written, all values of the preceding column may still need to be read. It is interesting to notice here the difference in memory size with the sequential execution. If the i and j loops are run sequentially, we saw that the array can be contracted into an array of size $n+1$ with the mapping $A[i, j] \mapsto A'[j - i \bmod (n+1)]$, which is nothing but the mapping $A[i, j] \mapsto A'[ni + j \bmod (n + 1)]$, a mapping that the methods of De Greef, Catthoor, De Man [33] and of Quilleré-Rajopadhye [64] would find.

Of course, the previous computation is based on an over-approximation of the standard semantics. It ignores the fact that, in a parallel loop, there is still some sequentiality, for each iteration, inside the body of the loop. As mentioned by Lefebvre and Feautrier [54] (end of Page 656), taking this additional order into account may be needed to avoid considering that a read occurring before a write within a given statement instance induces a conflict. Note however that, in the previous example, such accuracy is not needed because a conflict still occurs due to other reads and writes on the same array element: each value defined in a parallel front is read in several iterations of the next parallel front. But such cases could arise (see the example later). To handle them in an exact manner, we could compute conflicts as before, with a Live $\times$ Live strategy, and then remove the pairs corresponding to live-ranges ending and starting at the same iteration. But set differences are likely to be more expensive, and also, the removal of conflicts needs to be done with care because the live-ranges can still be conflicting due to other accesses. Another possibility is to build directly the right conflicts, without computing set differences, as follows:

```
# Operators
PrevEqDiff := { [i,j] -> [k,l]: i < k;
                [i,j] -> [i,l]: not (l = j) };
WriteBeforeT := (PrevEqDiff^-1).(Sched^-1).Write;
```

```
ReadAfterT := PrevEqDiff.(Sched^-1).Read;
WriteAtT := (Sched^-1).Write;
ReadAtT := (Sched^-1).Read;

# Liveness and conflicts
LiveCross := WriteBeforeT * ReadAfterT;
LiveEnd := WriteBeforeT * ReadAtT;
LiveStart := WriteAtT * ReadAfterT;
Conflict := (LiveCross^-1).(LiveEnd + LiveStart);
Delta := deltas (Conflict);
```

The computation is done for each particular iteration (i, j), including the parallel counter j. The inequality $l \neq j$ in the definition of `PrevEqDiff` is used to identify all live-ranges that fully "cross" this iteration, or that start or end at a parallel (but different) iteration. These live-ranges conflict with any live-range with a read or a write at iteration (i, j). This is not a Live×Live strategy (which would rather be hierarchical), but it is symmetric, because each pair of conflicting live-ranges is computed for one endpoint of each live-range. A Live×Write strategy would rather define the conflicts by:

```
Conflict := (LiveCross^-1).(LiveStart);
Delta := Sym(deltas (Conflict));
```

or even

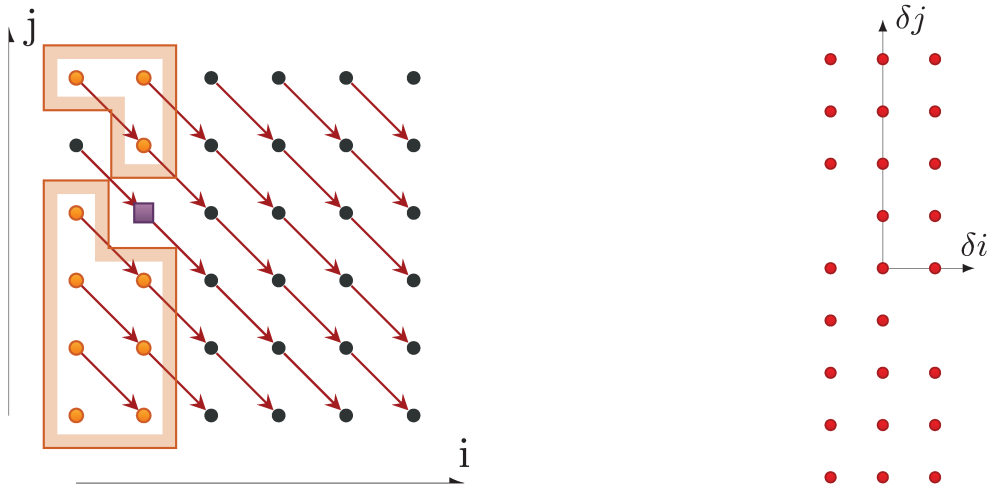
```
Conflict := (LiveCross^-1).(WriteAtT);
```

The following code (whose iteration domain is depicted in Figure 3.4a) is an example where paying attention to the sequentiality in the loop body pays off.

```
for(i=0; i<n; ++i)
    for parallel(j=0; j<n; ++j)
        A[i][j] = A[i-1][j-1] + 1
```

Here, if care is not taken, two successive columns of the array seem to conflict. But with the previous exact method, the set of conflicting differences is as depicted in Figure 3.4b. Then, the mapping $A[i, j] \mapsto A'[i + j \bmod (n + 1)]$, which is not so easy to find automatically, is a suitable array contraction.

The nesting of sequential and parallel loops can be handled in the very same way. For example, with a schedule (i, j, k, l) where j and l are parallel, we just need to define the relation `PrevEqDiff` as follows:



(a) Liveness set. (b) Conflicting differences.

Figure 3.4 – Example with non-chordal interferences: the array elements $A[i,j]$, $A[i,j+1]$, $A[i-1,j+1]$, and $A[i-1,j+2]$ form a cycle of length 4 with no chord.

```
PrevEqDiff := {
    [i,j,k,l] -> [i',j',k',l']: i < i';
    [i,j,k,l] -> [i,j',k',l']: not (j = j');
    [i,j,k,l] -> [i,j,k',l']: k < k';
    [i,j,k,l] -> [i,j,k,l']: not (l = l')
}
```

Then, again, one can compute the live-ranges that overlap with the time step (i,j,k,l) and make them conflict with the live-ranges with reads or writes at time (i,j,k,l). However, note that the corresponding "interference" graph (i.e., the conflicting pairs) may not be an interval graph anymore, because the underlying task graph is not a linear sequence of operations (here it is a series-parallel graph). It is not chordal either (unlike for SSA [18]), i.e., a chordless cycle of length 4 is possible. This arises in this last example where $A[i,j]$, $A[i,j+1]$, $A[i-1,j+1]$, $A[i-1,j+2]$ form a conflicting cycle, but $A[i,j]$ and $A[i-1,j+1]$ do not conflict, and neither $A[i,j+1]$ and $A[i-1,j+2]$. Nevertheless, the conflict graph has still some structure that will be explored later on.

3.3 Partial Orders and Beyond

The particular case of sequentiality in parallel loops, as exposed in Section 3.2.2, already shows the limit of reasoning with a concept of time step, where all variables "live" at this step are considered to be in conflict. Either such a notion of time step is

difficult to extract from the description of the "schedule" or it simply does not exist and the conflict relation is not transitive. In this section, we extend the analysis to work with "happens-before" relations, and to partial orders, to handle parallel specifications.

3.3.1 The Need for Generalizations

Recall the two different software pipelines in Figure 3.2a and Figure 3.2b introduced in Section 3.1. The fact that these software pipelines were defined, in the context of kernel offloading with inter-tile data reuse (see Chapter 2), to organize a double-buffering execution of *tiles* (aggregation of loop iterations within boxes) and not simple *iterations* is not important. They can be summarized as partial orders specifying the execution of three types of statements: loads (L, i), computations (C, i), and stores (S, i), indexed by a single loop iterator i. Both define a "schedule" expressing some restricted form of parallelism for the dependence task graph of Figure 2.4: computation tasks are organized as a sequence of tasks, communication tasks are also organized as a sequence of tasks, but with possibly some overlap between the two sequences (at "distance" at most 2).

The two different (periodic) schedules are implemented with synchronization mechanisms (arrows in the figures), imposing some precedence order. There is no explicit time step but, in these two particular cases, one can identify layers of parallel computations, fully sequentialized by a complete precedence graph between two successive layers (dotted horizontal lines in the figures). They can be used to define semantically-equivalent (in terms of liveness) schedules. For example, the software pipeline of Figure 3.2a behaves as the following schedule: $\theta(L, 2i) = (i, 0)$, $\theta(C, 2i) = \theta(L, 2i+1) = (i, 1)$, $\theta(S, 2i) = \theta(C, 2i + 1) = (i, 2)$, $\theta(S, 2i + 1) = (i, 3)$. This representation assumes that statements scheduled at the same time step—such as $(C, 2i)$ and $(L, 2i + 1)$—behave as parallel statements (or statements in two different parallel iterations). Hence, it is equivalent to the following pseudo-code and can be analyzed as discussed in Section 3.2.2:

```
for(i=...; i<...; ++i) {
  (L, 2i);
  do in parallel { (C, 2i) || (L, 2i+1) };
  do in parallel { (S, 2i) || (C, 2i+1) };
  (S, 2i+1);
}
```

The second software pipeline is a bit more tricky. It behaves as a schedule with a sequence of two parallel blocks, one performing (C, i), the other performing **in sequence**

$(S, i-1)$ then $(L, i+1)$. This is why a live-range ending in $(S, i-1)$ and a live-range starting in $(L, i+1)$ can both overlap with any live-range live in (C, i), but do not overlap with each other. This time, the software pipeline behaves as the following code (excluding epilogue and prologue):

```
for(i=...; i<..., ++i) {
  do in parallel {
    (C, i) || { (S, i-1);
                (L, i+1) }
  };
}
```

To summarize, both software pipelines can be described and analyzed as explained in Section 3.2.2. However, finding the right "layers" from the specification based on precedences among tasks is not obvious, and also it is not always possible. We now show how we can analyze the liveness directly from the description of a partial order among tasks: this is more general, easier when the notion of time step is not explicit, although the complexity may be higher as it depends on the number of statements more than on the number of time steps. It may also compute conflicting pairs in a redundant way (this is why the approaches of Section 3.2 may still be useful, for the cases where they can be applied, even if this should be supported by experimental evidence). The mechanism presented hereafter resembles the technique developed by Cohen and Lefebvre [22, 23], but for slightly different purposes (partial memory expansion given a parallel specification).

3.3.2 Traces and Conflicts

We now seek a method that, given a specification that may correspond to several executions, indicates that two memory locations conflict if there is an execution where they conflict.

An execution can be represented by a *trace* t, i.e., a sequence (a total order) of the operations executed. For each execution, we assume a canonical embedding (injective map) of its operations into a set of generic operations $\mathcal{O}$. This embedding usually follows the syntax of the language and the structure of the AST. For example, in the parametric code `for(k=n; k<2n; k++) S;` the i-th operation $a_{i,n}$ for a given value of n, which corresponds to the execution of S for iteration $k = n+i$ (when $i < n$), is in general abstracted by its code S and its "position vector" k. See also how operations are encoded for X10 analysis [82].

A given trace t may contain only a subset of these generic operations: we write $a \in t$ if a is executed in t and $a <_t b$ if $a \in t$, $b \in t$ and a is executed before b in t. By definition, $<_t$ is a total order for the operations executed in t. We define the relation $\mathcal{S}_\exists$ on $\mathcal{O} \times \mathcal{O}$ by:

$$\mathcal{S}_\exists(a, b) \text{ iff there is a trace } t \text{ such that } a <_t b. \tag{3.1}$$

Then, two memory locations x and y conflict if their live-ranges (intervals) overlap for some trace. There are multiple ways of expressing this overlap. One can use a "butterfly" set of constraints [2, 32], involving 4 operations: W_x (write of x), R_x (read of x), W_y (write of y), and R_y (read of y), with the following order (see dotted arrows in Figure 3.5a):

$$W_x <_t R_x,\ W_x <_t R_y,\ W_y <_t R_y,\ W_y <_t R_x.$$

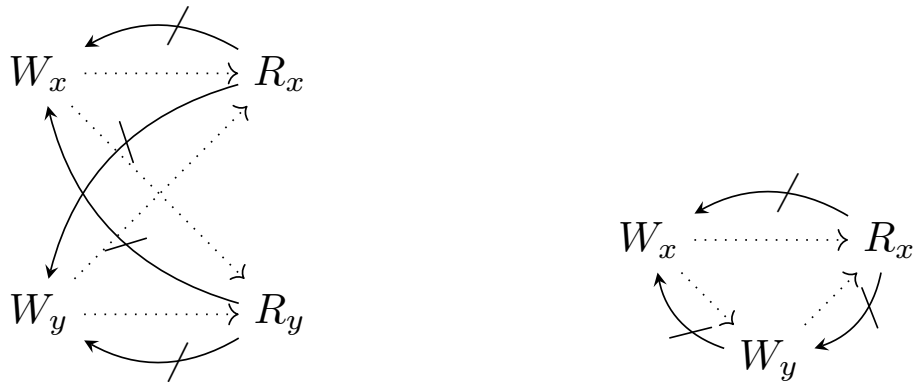
This symmetric method corresponds to the Live × Live approach of Section 3.2.1, given in Figure 3.3a. One can also reason, in an asymmetric manner, "at the time where y is defined" (as in the Live × Write approach of Figure 3.3c), with a set of constraints involving 3 operations, a write and a read of x (W_x and R_x), and a write W_y for y:

$$W_x <_t R_x,\ W_x <_t W_y,\ W_y <_t R_x. \tag{3.2}$$

Although equivalent for a given trace, the "triangle" approach (Figure 3.5b) is, for the same reason as for register allocation, more accurate when generalized to a case where not all operations are executed. We thus now focus on this one (Equation (3.2)).

We first replace the relation $<_t$ by the relation $\mathcal{S}_\exists$:

$$\mathcal{S}_\exists(W_x, R_x),\ \mathcal{S}_\exists(W_x, W_y),\ \mathcal{S}_\exists(W_y, R_x). \tag{3.3}$$



(a) Butterfly. (b) Triangle.

Figure 3.5 – Parallel strategies.

Note that this formalization is, in the worse case, an over-approximation. Indeed, it may be the case that there is no trace t where the 3 operations occur in this order while there are different traces where each two-by-two order is possible. However, in our context, this is most of the time equivalent, e.g., if all operations occur in all traces and if their scheduling freedom does not depend on their execution.[4]

Hereafter, we write $\neg\mathcal{R}$ the complement of a relation $\mathcal{R}$. We can now define the relation $\mathcal{R}_\forall$, generalization of the "happens-before" relations that we mentioned before, with $\mathcal{R}_\forall(a, a)$ for all $a \in \mathcal{O}$, and, for $a \neq b$, with:

$$\mathcal{R}_\forall(a, b) \text{ iff, for all traces } t,\ a, b \in t \text{ implies } a <_t b. \tag{3.4}$$

Since $\neg\mathcal{R}_\forall(a, b)$ iff $\mathcal{S}_\exists(b, a)$, Equation (3.3) becomes (see Figure 3.5b):

$$\neg\mathcal{R}_\forall(R_x, W_x),\ \neg\mathcal{R}_\forall(W_y, W_x),\ \neg\mathcal{R}_\forall(R_x, W_y). \tag{3.5}$$

In general, the relation $\mathcal{R}_\forall$ may be neither anti-symmetric, nor transitive, although Equation (3.5) can still be used to compute conflicts. Consider all possible situations:

- If $\mathcal{R}_\forall(a, b)$ and $\neg\mathcal{R}_\forall(b, a)$, there is a trace t with $a <_t b$ and the same for all traces that contain a and b.
- If $\neg\mathcal{R}_\forall(a, b)$ and $\mathcal{R}_\forall(b, a)$, this is the converse situation.
- If $\neg\mathcal{R}_\forall(a, b)$ and $\neg\mathcal{R}_\forall(b, a)$, a and b are parallel, i.e., there is a trace with a before b, and the converse.
- If $\mathcal{R}_\forall(a, b)$ and $\mathcal{R}_\forall(b, a)$ then a and b are never executed in the same trace (two branches of an `if` for example).

The first two cases induce local asymmetry (but not necessarily transitivity) as in order relations. The third one corresponds to parallelism, as non-comparable operations in partial orders. In the last case, the relation is not asymmetric, and a and b will never contribute to conflicts because Equation (3.5) cannot be true (they are never executed together). Note however that $\mathcal{R}_\forall$ defines a partial order if all operations are executed in any trace, e.g., for codes with no `if` conditions.

If $\underline{\mathcal{R}}_\forall$ is an **under-approximation** of $\mathcal{R}_\forall$ (i.e., $\underline{\mathcal{R}}_\forall \subseteq \mathcal{R}_\forall$), the relation $\underline{\mathcal{R}}_\forall$ (and the corresponding $\overline{\mathcal{S}}_\exists$) exhibits more traces, Equations (3.3) and (3.5) are more likely to be satisfied, and the resulting conflicts are thus conservative. One way to get a **partial**

[4]However, for critical sections (or the `atomic` construct), the analysis is more accurate with Equation (3.2) than with Equation (3.3).

order (if needed) as an under-approximation of $\mathcal{R}_\forall$ is to make a consistent asymmetric choice between $\mathcal{R}_\forall(a,b)$ and $\mathcal{R}_\forall(b,a)$, for example by defining, for $a \neq b$:

$$\underline{\mathcal{R}}_\forall(a,b) \text{ iff, for all traces } t,\ a <_t b \text{ or } b \notin t \tag{3.6}$$

(or the symmetric version with $a \notin t$). In this case, $\underline{\mathcal{R}}_\forall(a,b)$ implies that if b is executed, then a is always executed too, and interpreted as "the execution of a is always visible to b". Assuming that all operations execute at least once, one can easily prove that $\underline{\mathcal{R}}_\forall$ is anti-symmetric and transitive, thus a (strict) partial order. Now let us focus on partial orders, a particular case of special interest.

When $\underline{\mathcal{R}}_\forall$ is a partial order $\preceq$, Equation (3.5) becomes:

$$R_x \not\prec W_x,\ W_y \not\prec W_x,\ R_x \not\prec W_y. \tag{3.7}$$

This situation, where the freedom of parallelism (the set of all possible executions, and even more) is described through some representation (scheduling function or language constructs) expressing a partial order $\preceq$, is very common. This is the case in all previous examples (sequential code, nested loop parallelism as in OpenMP, software pipelining). The X10 "happens-before" relation (at least in the setting of affine control loops with `async/finish` keywords [82]) is also a partial order. The relation $\prec_{\vec{s}}$ we used in Chapter 2 for unaligned tiles is an under-approximation of $\mathcal{R}_\forall$ to get a partial order on all tiles and make liveness analysis for parametric tiling feasible, in a more accurate way than with Equation (3.6) (see also the discussion in Section 3.4).

Note that $R_x \not\prec W_y$ means that either $W_y \prec R_x$ or W_y and R_x are not comparable, i.e., can be executed in parallel ($W_y \,||\, R_x$). When $\preceq$ can be expressed in a (piece-wise) affine way, the conflicts can be computed similarly. Equation (3.7) has some similarity with the Live $\times$ Write method (Section 3.2.1). There is no absolute time, but we can reason relative to the "time" when W_y is being computed to see if W_x may happen before W_y and, similarly, if R_x may happen after W_y.

3.3.3 Partial Orders and Structure of Conflicts

We have shown how to compute conflicts given an extended concept of happens-before relations and partial orders. In this section, we prove the following important structure theorem on conflicts for partial orders:

Theorem 5. *For a partial order $\preceq$, with no dead code, no undefined read, but possibly data races, the complement of the conflict graph is a comparability graph (i.e., defines a strict partial order $\lhd$), from which one can define an optimal polynomially-computable static reuse of memory locations.*

Intuitively, this is because if x and y do not conflict, only two cases arise. Either all reads and writes of x occur before (following $\preceq$) any write of y (we write $x \lhd y$), or the converse (we write $y \lhd x$). This defines a strict partial order, which is an orientation of the complement of the conflict graph. Furthermore, when $x \lhd y$, then y can be mapped safely at the same location as x, in a form of memory reuse. The full proof is as follows.

Proof. Assume that, for any memory location x and any read R_x of x, there is no execution (according to $\preceq$) where x is not defined (but races are possible), i.e., there is a write W_x of x such that $W_x \prec R_x$. Assume also that for any write W_x of x, there is a read R_x of x such that $W_x \prec R_x$.[5] Now, consider two memory locations x and y that do not conflict, according to Equation (3.7), and two writes W_x and W_y of x and y respectively.

First, W_x and W_y are always comparable for $\prec$. Indeed, if $W_x \not\prec W_y$ and $W_y \not\prec W_x$, then with R_x such that $W_x \prec R_x$, we get $R_x \not\prec W_x$ (as $\prec$ is asymmetric) and $R_x \not\prec W_y$ (otherwise $W_x \prec W_y$ by transitivity of $\prec$), and thus x and y would conflict. Now, if W_x, W'_x, and W_y are such that $W_x \prec W_y \prec W'_x$, then since we consider that a value is live from its very first write to its very last read (without considering lifetime "holes"), then with R'_x such that $W'_x \prec R'_x$, we get $W_x \prec W_y \prec R'_x$, which implies $R'_x \not\prec W_x$, $W_y \not\prec W_x$, and $R'_x \not\prec W_y$, thus x and y conflict.

We just proved that all writes of x are before all writes of y (or the converse). Assume the first ($W_x \prec W_y$ for any writes of x and y), then for any read R_x of x, $R_x \prec W_y$. Indeed, if $R_x \not\prec W_y$, then $R_x \not\prec W_x$ (otherwise $R_x \prec W_y$ by transitivity). And since $W_y \not\prec W_x$, the memory locations x and y would conflict. □

This result has a lot of similarities with the work of Berson et al. [12] and Touati [71]. The difference is that, instead of looking for the minimal number of memory locations sufficient for any schedule, which is shown to be NP-complete, we look for an allocation with minimal number of memory locations valid for any schedule (a possibly slightly larger number). Since the *reuse graph* (the complement of the conflict graph) is a comparability graph, the conflict graph is also a perfect graph, its chromatic number can be computed in polynomial time, and an allocation of same size, based on static reuse, can be defined by a maximal number of independent chains in the reuse graph. When these graphs are not given by extension but through conflicting relations, it is not clear how this can be exploited to find better memory allocations. However, the formulation of the reuse graph gives some conceptual insight on previous work based on memory reuse and occupancy vectors, as we now discuss.

[5] This "read" can be artificially added, just to code the fact that even if W_x may be useless for a given execution, unless we do dead code elimination, it stores some value and can destroy a live value. It thus counts for conflicts.

3.4 Links with Previous Work

The procedures described in Sections 3.2 and 3.3 are generalizations of previous approaches to compute conflicts between memory locations (registers and array elements), a necessary step to enable memory reuse. The case of sequential codes [2], of parallelization through multi-dimensional affine scheduling [38] resulting in inner parallel loops, were well-known. The fact that the sequentiality within a statement of such inner parallel loops needs to be taken into account as a particular case was a folk theorem. We do not recall such previous work here. All other situations we covered, either to derive special techniques (Section 3.2), or to handle more general parallelism description (Section 3.3), were not proved correct or even handled before. We now discuss some other related work to which our study brings some new insight.

The first and (unexpectedly) closest work is the study of Cohen et al. [22, 23], not in the context of memory contraction (reuse) but in the context of memory expansion: find the minimal expansion needed to correct a parallelization based on flow dependences only (thus ignoring anti-dependences). It is comforting to see that, when x and y are different, the conditions for minimal memory expansion given by Equation (5.21) in Cohen's Ph.D. thesis [22] (Page 193) are the same, but with different arguments and setting. The additional complication in their work comes from the will to avoid expansion by exploiting some knowledge, from the sequential execution, on conditionals [22]. This is not our case: we start directly from a given parallel specification, but revisiting their work may give good insight to represent conditionals and deal with them in a more accurate way than with the under-approximation of Equation (3.6).

Even if it was not stated in these terms, the work described in Chapter 2 on parametric tiling also uses a special partial order to under-approximate $\mathcal{R}_\forall$. To make the problem piece-wise affine, "unaligned" tiles are introduced (tiles in shifted tilings), which are, by definition, never executed with the tile corresponding to W_y in Equation (3.5). A partial order among all tiles is then defined ($T \prec T'$ if every point in T is executed before any point in T') to define the conflicts. This method is much more accurate for liveness analysis than defining, as suggested in Equation (3.6), $\underline{\mathcal{R}_\forall}(a, b)$ iff, for all traces t, $a <_t b$ or $b \notin t$. The latter assumes that tiles in different tilings (i.e., unaligned) can execute in parallel, making all array elements conflict with each other, which is of course not satisfactory because way too conservative.

Finally, Theorem 5 gives new insight on the concept of *occupancy vectors*. An occupancy vector $\vec{o}$ for an array A is such that $A[\vec{i} + \vec{o}]$ can reuse the memory location of $A[\vec{i}]$ for all $\vec{i}$. Lattice-based memory allocation [32] is based on the set DS of conflicting dif-

ferences, computed from the conflict graph ($\vec{d} \in DS$ if $\vec{d} = \vec{i} - \vec{j}$ such that $A[\vec{i}]$ and $A[\vec{j}]$ conflicts). Occupancy vectors (or reuse vectors) give the dual view, in the complement (the reuse graph): $\vec{o}$ is such that it is never a conflicting difference, i.e., it is in the complement of DS. This duality was partly exploited in lattice-based memory allocation [32] for the design of heuristics. We will exploit it further in Chapter 4 to extend this technique to select more suitable directions of array reuse.

It also gives new insight for the concept of *universal* occupancy vectors (UOV) [68], an occupancy vector valid for all possible schedules, constrained by memory dependences only. The theory developed here could be used to address this problem: what we need is the relation $\preceq$ defined by $a \preceq b$ if there is a dependence path from a to b, i.e., the transitive closure of dependences. The problem is that, if an over-approximation can be built in the context of Presburger arithmetic, here we need an under-approximation. In the work of UOVs [68], the problem can be solved because it is restricted to uniform dependences and assuming large-enough iteration domains. So, transitivity of dependences is obtained by addition of dependence vectors.

Similarly, QUOV (quasi UOV) [83], designed to handle occupancy vectors valid for all possible tilings of a code, makes an assumption on the dependence cone that enables to capture the transitive closure. Finally, Thies et al. [70] proposed a method to build occupancy vectors valid for all possible one-dimensional affine schedules (AUOV). The set of all such schedules θ can be expressed with Farkas lemma. Then, imposing $\theta(b) < \theta(a)$ when b depends on a through a direct dependence captures the transitivity of $\prec$ through the transitivity of $<$. However, building a true UOV (i.e., for all schedules), even for affine dependences, remains open, unless the transitive closure of dependences can be expressed.

3.5 Conclusion

In this chapter, we presented our results, published at the IMPACT'16 workshop [29], on how to extend liveness analysis, and more precisely interferences/conflicts between scalar or array elements, to parallel specifications. The most generic "happens-before" relation we considered ($\mathcal{R}_\forall$ - Equation (3.4)) is not even a partial order but we may still compute conflicts. We also focused on cases when the happens-before relation is a partial order (or can be approximated as one), which arises in many parallel programming models such as OpenMP, X10, and so on.

In extending the liveness analysis, we have described several ways to compute the conflicting relation, depending on the situation. They differ in their computational complexity (e.g., number of dimensions, or of unions involved), what they can express, and if

they can be used for intermediate simplifications (coalescing of unions, approximations). It is not clear yet which solution will be, in practice, the most efficient one for real programs, either programs with complex accesses or large programs involving many different accesses. Also, even if Theorem 5 states that the minimal size of an allocation can be computed when the conflict graph is described in extension, it is not clear how it can be done in a symbolic (polyhedral) way. Lower bounds can be derived by clique computations, and it is more likely that exploiting the structure of the conflict graphs for special cases, as done in Section 3.2, will lead to more accurate lower bounds.

Finally, we hope that the analysis presented in this chapter to serve as a stepping stone to the analysis of data reuse on any parallel language. As for Theorem 5, which explicits the link between memory conflicts (or interferences) and memory reuse, it already gave us some inspiration in order to improve element-wise array memory allocations (see our technique described in Chapter 4), by expliciting the connection between the interferences set (constraints to be enforced) and its complement (where valid reuse is explicit). We will also illustrate in Chapter 5 how the "triangle" equation, i.e., for a partial order Equation (3.7), can be used to build the conflict sets that occur when offloading the tiles defined in Chapter 2 following the software pipeline given in Figure 3.2b. This step is necessary to be able to implement the local storage with sliding windows, when bounding boxes are not enough due to pipelining and inter-tile data reuse.

Chapter 4

Memory Allocation

Summary

The results presented in this chapter extend lattice-based memory allocation [32], an earlier work on memory reuse through array contraction. Such an optimization is used for optimizing high-level programming languages where storage mapping may be abstracted away from programmers and to complement code transformations that introduce intermediate buffers, such as those presented in Chapter 2. The main motivation for this extension is to improve the handling of more general forms of specifications we see today, e.g., with loop tiling, pipelining, and other forms of parallelism available in explicitly-parallel languages. Specifically, we handle the case when conflicting constraints (those that describe the array indices that cannot share the same location, as studied in Chapter 3) are specified as a (non-convex) union of polyhedra. The choice of directions (or basis) of array reuse becomes important when dealing with non-convex specifications.

We extend the two dual approaches proposed in the original work on lattice-based memory allocation [31] to handle unions of polyhedra, and to select a suitable basis. Our final approach relies on a combination of the two, also revealing their links with, on one hand, the construction of multi-dimensional schedules for parallelism and tiling (but with a fundamental difference that we identify) and, on the other hand, the construction of universal reuse vectors (UOV), which was only used so far in a specific context, for schedule-independent mapping.

4.1 Motivation

As the gap between memory performance and compute power keeps increasing, the importance of efficient memory usage is also increasing. As recalled in Section 1.1, this is even more emphasized when exploiting hierarchical memories and/or when accelerators such as GPUs or FPGAs are used as they are often limited by the on-chip memory

capacity and/or the bandwidth between the host and the accelerator. The problem of efficient memory allocation is further complicated by the trade-off between parallelism and memory usage.

Memory reuse is a standard technique for allocating scalar variables to registers. Memory reuse for arrays, in particular intra-array reuse, is used to reduce statically the memory footprint of data-intensive applications, after analyzing the liveness of the different elements of an array. The need for such array contraction is of course important for high-level program specifications (for example array languages) where the programmer expresses its applications in an abstract view of the storage locations, possibly even using arrays in single assignment, thus without paying too much attention to memory usage. But such a memory allocation technique is also required within compilers themselves as a complementary step to many code transformations, for the design of intermediate buffers introduced by the compiler and the management of local memories, as is required for the kernel offloading approach introduced in Chapter 2, and to reduce the effect of some previous array expansion phases.

In this chapter, we extend a technique called lattice-based memory allocation [31, 32] that was originally proposed as a generalization of different strategies based on affine mappings with foldings by modulo operations (called *modular mappings*), formalized with integer lattices. The original work was aimed at handling regular kernels executed by sequential and/or limited forms of parallel schedules where simple optimization strategies appeared to be sufficient. We extend this framework on two main aspects:

- *conflict set* (a relation to express array elements that may not be mapped to the same memory location) as a union of polyhedra, where the initial work is limited to a convex polyhedron;
- *optimized basis* heuristics to choose the direction of the modular mapping allocation, or the basis of the corresponding lattice.

The key insight is that optimizing the dimensions of a multi-dimensional modular mapping successively, greedily minimizing the resulting array size for each dimension, may lead to worse overall allocations, a phenomenon that is exacerbated when the conflict set is not convex. We start with an example in Section 4.2 to illustrate this situation, and to give the key intuition behind our approach to address this issue. Section 4.3 introduces the necessary background, defining more precisely the notions of conflict sets and modular mappings. Section 4.4 presents the main theory and algorithms extending two previously proposed approaches [31] to non-convex conflict sets and with optimized basis. It also shows how these two heuristics can be combined by first building short reuse

vectors (indicating array elements mapped to the same location), reminiscent of UOV (universal occupancy vector) construction [68, 70, 83], which then constrain the way the full mapping is built, with a technique similar to multi-dimensional scheduling [38, 15]. Section 4.5 illustrates our technique on several examples, showing how a tool such as the Integer Set Library (`isl`) [73] can be used to implement it. Finally, we discuss links with UOV, scheduling, and earlier work on intra-array reuse in Section 4.6, and we conclude in Section 4.7.

4.2 Intuition of the Approach

In this section, we illustrate the key intuition behind our work using an example with a simplified view of the problem. There are many existing array mapping techniques that work well when the conflict set is described with a single polyhedron [13, 32, 54, 64]. We are interested in more complex cases that involve unions of polyhedra.

Finding a storage for live-out values of tiles after loop tiling [81] is a common situation that gives rise to non-convex unions. As an example, suppose we seek an optimized allocation for a live-out set (that can be computed as exposed in Chapter 3) corresponding to a reverse-L shaped region depicted in Figure 4.1. The number of live-out values is $4N - 4$ if N is the length of the square edge (24 in the figure, with $N = 7$). One "good" way to allocate all these values to different locations is to map array elements along $(1, 1)$ modulo 2 (as depicted in Figure 4.1a). This corresponds to the mapping $(x, y) \mapsto (x - y, y) \bmod (2N - 1, 2)$, with an array of size $2(2N - 1)$, thus only 2 elements more than the optimal.

Existing techniques struggle to find this allocation for different reasons. One of the earliest, yet powerful, method for memory allocation by Lefebvre and Feautrier [54] consists in choosing some successive modulo folding for each dimension, restricting to mappings along the canonical axes. The first modulo should be larger than the maximal distance between two points (here $N - 1$ along the x axis), then the second modulo larger than the maximal distance between two points with the same value of x, which is also $N - 1$. This results here in an allocation of size N^2, with the corresponding mapping $(x, y) \mapsto (x, y) \bmod (N, N)$. First computing the convex hull of the reverse-L shaped region will not help either because leading to a too coarse over-approximation.

Several other techniques have been presented that explore allocations using non-canonical projections or mappings [13, 32, 64]. We leave the detailed discussions of these work to Section 4.6 and only give here a high-level description of the most recent work by Bhaskaracharya, Bondhugula, and Cohen [13], which has the same objectives

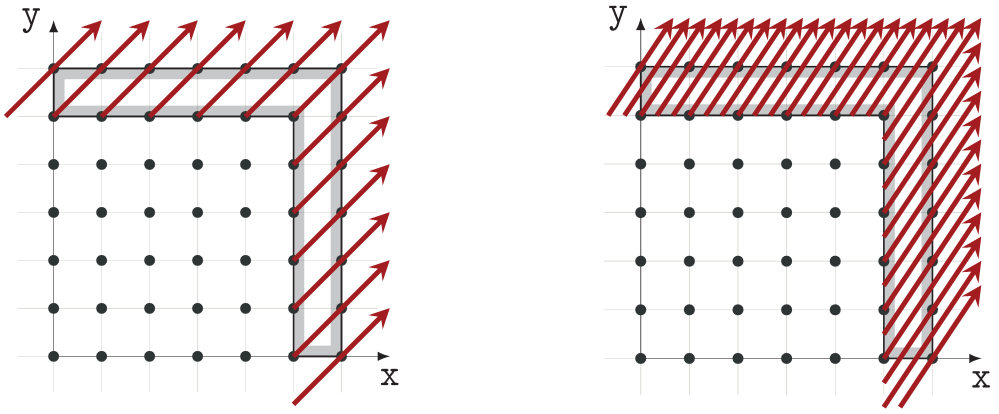


Figure 4.1 – An example to illustrate the key intuition. The reverse-L shape of width 2 may be viewed as the live-out values of a square tile of size N, when the data dependences (not depicted here) across tiles have a uniform length of 2 in each dimension. The memory allocation characterized by the reuse vectors in Figure 4.1a leads to the optimal affine mapping (projection plus modulo 2) that we seek. Figure 4.1b illustrates another valid affine mapping (projection) that may be found by other techniques, with N extra storage.

as ours. They proposed a method that combines ideas from multi-dimensional affine scheduling with memory allocation. The key idea is to interpret each dimension of a multi-dimensional affine mapping as a family of parallel hyperplanes that separate points from each other. If each parallel hyperplane contains at most one point of the domain to be stored, the corresponding affine function represents a legal memory allocation mapped to a one-dimensional array where the "latency" [37] (width or maximal distance), plus 1, is the number of elements in the array. Using this formulation, their approach first tries to find such a family of hyperplanes that contain at most one point each. If such hyperplanes cannot be found, their technique finds a second family of hyperplanes focusing on points that lie in the same hyperplane of the first family, and so on. The number of dimensions of the final array is the number of linearly-independent hyperplanes generated.

One possible allocation that can be found by their technique is illustrated in Figure 4.1b. Since the mapping they compute also depends on how the domain is decomposed into a union of polyhedra, their technique may find other (worse) allocations, as explained in Section 4.6.2. The key reason of this inefficiency lies in the objective functions used. Their primary objective tries to minimize the number of hyperplane families, i.e., the number of dimensions of the mapped array, by maximizing at each stage the number of polyhedra whose points are all separated by hyperplanes. Their secondary objective is to minimize the number of elements in the corresponding dimension, i.e., to find hyperplanes yielding minimal width.

The primary objective can be satisfied by using the hyperplanes shown in Figure 4.1b, parallel to the vector $(2, 3)$ (or similarly parallel to $(3, 2)$), thus orthogonal to $(3, -2)$, with width $5(N-1)$. The corresponding mapping is $(x, y) \mapsto (3x - 2y) \bmod (5N - 4)$. For $N = 7$, this gives, as depicted, 31 different memory locations. However, if a one-dimensional allocation cannot be found, the greedy heuristic will minimize the number of elements in the current dimension, which, as explained in Section 4.6.2, may favor in this example hyperplanes parallel to the canonical axes (i.e., the shortest one-dimensional schedules), leading to the same mapping as Lefebvre and Feautrier, with size N^2. The optimal solution is not obvious in this iterative formulation, since the "good" hyperplanes parallel to $(1, 1)$ (i.e., orthogonal to $(1, -1)$) do not satisfy the primary objective. Indeed, some points are still mapped to a common location, and it is not optimal for this dimension because its corresponding width is $2(N-1)$ while $N-1$ is achievable.

In our work, we use a different formulation to overcome inefficiency in these cases. For our example, it can be intuitively explained as finding the shortest vector that points to somewhere outside of the domain of interest, which is the vector $(2, 2)$. In general, we need multiple linearly-independent vectors, captured through lattices.

4.3 Background

We now recall the two key concepts used in this chapter, *conflict set*, a concept already used in Chapter 3 and which gives the constraints for valid mappings, and *modular mappings*, a particular form of functions used for intra-array reuse.

4.3.1 Conflict Set

Lattice-based memory allocation [31, 32], as well as all prior work on intra-array reuse [33, 64, 54], is based on the concept of **conflicting (array) elements**, i.e., the set of pairs of elements that should not be mapped to the same location. This set is the counterpart, for array elements, of the well-known interference graph defined for register allocation. In register allocation, vertices of this graph correspond to scalar variables and edges indicate that two variables should not be mapped to the same register. Graph coloring can then be used to derive a valid register assignment. For intra-array reuse, the set of conflicting indices for a given array A is not expressed in extension, but in a symbolic way, e.g., with polyhedra specifying a symmetric relation $\bowtie$: $\vec{i} \bowtie \vec{j}$ if $A(\vec{i})$ and $A(\vec{j})$ should not be mapped to the same location. This relation is then used to derive a **valid mapping**, i.e., a function σ such that $\sigma(\vec{i}) \neq \sigma(\vec{j})$ if $\vec{i} \bowtie \vec{j}$ and $\vec{i} \neq \vec{j}$.

Such array mappings can be defined in a post-scheduling phase, i.e., for a particular execution or schedule (schedule-dependent mappings [33, 54, 64]), or before the final schedule is defined (schedule-independent mappings [68, 70, 83]), so that the mapping is valid for any further valid code transformation (or a subclass, such as loop tiling). The latter situation arises also when compiling programs, expressed in a parallel language such as OpenMP or X10, on top of a runtime system, in which case the exact schedule is not statically known. The mapping is then defined for the set of all possible schedules induced by the parallel language constructs.

For intra-array reuse, the construction of the conflicting indices requires some symbolic liveness analysis. Actually, as we explained in Chapter 3, all the situations previously mentioned are particular instances of the more general problem of defining liveness analysis and "simultaneously live" array elements for explicitly-parallel specifications, in particular for specifications defining a partial order on operations. How to build such a relation $\bowtie$ is not our concern here: we assume it has been computed, possibly over-approximated, possibly using the different methods developed in Chapter 3, and that it expresses, for each array to be contracted, the set of pairs of conflicting indices (or their differences) in a compact symbolic form. For optimization purposes, we focus on the case where the conflicting differences are the integer points in a union of polyhedra $\mathcal{K} = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_r$. But the theory can possibly be used in more general situations, e.g., with Presburger arithmetic formulas as available in the `isl` library [73], or even polynomial expressions [39], as long as the corresponding optimizations can be carried out.

4.3.2 Modular Mappings

In lattice-based memory allocation, the mappings are restricted, both for optimization and code generation purposes, to **modular mappings**. A modular mapping $(M, \vec{b})$, defined by a $p \times n$ integral matrix M and a positive integral vector $\vec{b}$ of dimension p, maps the index $\vec{i}$ of a n-dimensional array to $\sigma(\vec{i}) = M\vec{i} \bmod \vec{b}$ (the modulo is applied component-wise) in a p-dimensional array of shape $\vec{b}$. The **size** of the mapping is the size of the resulting array, i.e., the product of the elements of $\vec{b}$. Its **dimension** is the number of dimensions of the resulting array, i.e., p. As modular mappings are affine, it is enough to work with the set of **conflicting differences** $\mathcal{K} = \{(\vec{i} - \vec{j}) \mid \vec{i} \bowtie \vec{j}\}$. Indeed, $(M, \vec{b})$ is valid iff $M(\vec{i} - \vec{j}) \neq \vec{0} \bmod \vec{b}$ if $\vec{i} \bowtie \vec{j}$ and $\vec{i} \neq \vec{j}$, i.e., iff $\vec{x} \in \mathcal{K}$ and $M\vec{x} = \vec{0} \bmod \vec{b}$ imply $\vec{x} = \vec{0}$. The initial theory of lattice-based memory allocation [31, 32] focuses on the case where $\mathcal{K}$ is the set of integer points in a 0-symmetric (symmetric with respect to $\vec{0}$) polyhedron. Then, lower and upper bounds on memory size can be given and more properties proved.

Although here we focus on intra-array reuse, it is worth pointing out that modular mappings can be used in other contexts, e.g., for bank allocation, to allow parallel accesses to memory [21, 32], or more generally whenever renewable resources need to be shared.

4.4 Greedy Mapping and Lattice Constructions

In this section, we present our main contribution that extends lattice-based memory allocation. Our work is based on two dual approaches proposed by Darte et al. [31] that are equivalent when the set of conflicting differences $\mathcal{K}$ is a polyhedron. We show how we can extend them to handle the case where $\mathcal{K}$ is a union of polyhedra, and then to further optimize the selection of the basis (i.e., the matrix M in the modular mapping) to reduce the memory size. The resulting optimizations lead to two complementary greedy approaches, in which the rows of the matrix M are optimized in the opposite order. We then show in Section 4.4.3 how to combine them to try to get the best of both worlds, the first one being well suited to handle parameters while the second one is often more suitable to detect directions with constant (i.e., non-parametric) reuse.

Both approaches define mappings given a basis of $\mathbb{Z}^n$, i.e., how to choose the modulo vector $\vec{b}$. The first approach directly works with the matrix M of the mapping while the second one works with its kernel, i.e., the set of vectors $\vec{i}$ such that $\sigma(\vec{i}) = 0$. Such a set is a *lattice* of $\mathbb{Z}^n$, thus the name of the technique. Section 4.4.1 extends the first over the mapping space, Section 4.4.2 the second over the lattice space. We combine the two in Section 4.4.3. Unimodular matrices (invertible in the integers) play an important role in this construction, both for the optimizations in the two approaches, and for their combination. The first approach builds some rows of M, the second some columns of M^{-1}.

4.4.1 Basis Selection in Mapping Space

The following mechanism [31] is the "successive modulo" principle [54], generalized to any set of n linearly independent integral vectors. Following the previously given notations, $\mathcal{K}$ is the 0-symmetric set of conflicting differences.

Heuristic 1.

- *Choose n linearly independent integral vectors $(\vec{c}_1, \ldots, \vec{c}_n)$.*
- *Compute $F_i^*(\vec{c}_i) = \sup\{\vec{c}_i.\vec{z} \mid \vec{z} \in \mathcal{K}, \forall j < i, \vec{c}_j.\vec{z} = 0\}$, successively for all $1 \leq i \leq n$.*
- *Define M the matrix with row vectors $(\vec{c}_i)_{1 \leq i \leq n}$ and $\vec{b}$ an integer vector such that $b_i > F_i^*(\vec{c}_i)$ for all $1 \leq i \leq n$.*

For each i, $F_i^*(\vec{c}_i)$ is the **width** along $\vec{c}_i$ of the intersection of $\mathcal{K}$ and the orthogonal of the vector space defined by $(\vec{c}_1, \ldots, \vec{c}_{i-1})$. Our goal is to adapt this heuristic when $\mathcal{K}$ is described by a union of polyhedra and to design a method to choose a suitable basis.

Theorem 6. *The modular mapping built by Heuristic 1 is a valid mapping for $\mathcal{K}$, assuming that $\mathcal{K}$ is bounded and 0-symmetric.*

Proof. Let $\vec{x} \in \mathcal{K}$ with $M\vec{x} \bmod \vec{b} = 0$. Since $b_1 > F_1^*(\vec{c}_1) = \sup\{\vec{c}_1.\vec{x} \mid \vec{x} \in \mathcal{K}\} = \sup\{|\vec{c}_1.\vec{x}| \mid \vec{x} \in \mathcal{K}\}$ (as $\mathcal{K}$ is 0-symmetric), $\vec{c}_1.\vec{x} = 0 \bmod b_1$ implies $\vec{c}_1.\vec{x} = 0$. Then, $\vec{c}_2.\vec{x} = 0 \bmod b_2$, but $b_2 > F_2^*(\vec{c}_2) = \sup\{|\vec{c}_2.\vec{x}| \mid \vec{x} \in \mathcal{K}, \vec{c}_1.\vec{x} = 0\}$, thus $\vec{c}_2.\vec{x} = 0$. Continuing this process, we get $\vec{c}_i.\vec{x} = 0$ for all i. Since the $\vec{c}_i$ are n linearly independent vectors, this implies $\vec{x} = 0$. This shows that the modular mapping $\sigma = (M, \vec{b})$ is valid. Indeed, if $\vec{i} \bowtie \vec{j}$, then $\vec{i} - \vec{j} \in \mathcal{K}$ and thus $\sigma(\vec{i}) \neq \sigma(\vec{j})$, unless $\vec{i} = \vec{j}$. □

Theorem 6 shows that, although it was initially designed assuming that $\mathcal{K}$ is a polyhedron [31], Heuristic 1 is actually valid with weaker hypotheses. Also, the following important properties remain true:

- The different values F_i^* depend on the order in which the vectors $(\vec{c}_i)_{1 \leq i \leq n}$ are considered. Considering all $n!$ orders can help reducing the size of the mapping. (In practice, n is small.)
- Increasing the values of $\vec{b}$ keeps the validity of the mapping. This can be used for example to restrict the values of $\vec{b}$ to power of 2. Also, if $\mathcal{K}$ is a union of different 0-symmetric pieces, one can compute a vector $\vec{b}_i$ for each piece, each obtained with a possibly different order of the basis vectors, and then take the maximum, component-wise, of the $\vec{b}_i$ to get a valid $\vec{b}$ for the whole $\mathcal{K}$.
- The same proof as for Theorem 6 shows that the modular mapping $\vec{c}_1.\vec{x} + b_1(\vec{c}_2.\vec{x} + b_2(\cdots + b_{n-1}\vec{c}_n.\vec{x}))$ mod $\prod_i b_i$ is valid, of dimension 1, and with same size. In other words, the dimension of a mapping is not related to its size, and some 1D mappings can even be found with a multi-dimensional approach.

Optimizing the Basis

We now show how the previous heuristic can be extended to define an optimized version, where the basis $(\vec{c}_i)_{1 \leq i \leq n}$ is built in a greedy fashion, so that $F_i^*(\vec{c}_i)$ is minimized at each step:

$$\vec{c}_i = \text{argmin}\{F_i^*(\vec{c}) \mid \vec{c} \in \mathbb{Z}^n \text{ linearly independent with } \vec{c}_j, j < i\}$$

This is a min-max problem, which can be solved, when $\mathcal{K}$ is a union of polyhedra, in the same way schedules with minimal latency (which corresponds to the width) are built [38, 30], i.e., either with the duality theorem of linear programming or with the affine form of Farkas' lemma [67] (hereafter, Farkas lemma for short). Let us detail the technique with Farkas lemma, which we recall here.

Lemma (Farkas, affine form)**.** *Let P be a non-empty polyhedron defined by affine inequalities $P = \{\vec{x} \mid Q\vec{x} \leq \vec{e}\}$. Then $\vec{c}.\vec{x} \leq \delta$ for all $\vec{x} \in P$ iff there exists $\vec{y} \geq \vec{0}$ such that $\vec{c} = \vec{y}.Q$ and $\vec{y}.\vec{e} \leq \delta$.*

Now, the width of P along a vector $\vec{c}$ can be computed as follows:

$$\begin{aligned}\max\{\vec{c}.\vec{x} \mid \vec{x} \in P\} &= \min\{\delta \mid \vec{c}.\vec{x} \leq \delta \text{ for all } \vec{x} \text{ s.t. } Q\vec{x} \leq \vec{e}\} \\ &= \min\{\delta \mid \vec{y} \geq \vec{0},\ \vec{c} = \vec{y}.Q,\ \vec{y}.\vec{e} \leq \delta\}\end{aligned}$$

Similarly, the quantity $F_i^*(\vec{c})$ is equal to:

$$F_i^*(\vec{c}) = \min\{\delta \mid \vec{y} \geq \vec{0},\ \vec{c} = \vec{y}.Q + \vec{z}.C_{i-1},\ \vec{y}.\vec{e} \leq \delta\} \tag{4.1}$$

where C_{i-1} is the matrix whose rows are $\vec{c}_1, \ldots, \vec{c}_{i-1}$, and the vector $\vec{z}$ does not need to be nonnegative. Finally, for a union of polyhedra $\mathcal{K} = \cup_{1 \leq j \leq r} \mathcal{P}_j$ where $\mathcal{P}_j = \{\vec{x} \mid Q_j\vec{x} \leq \vec{e}_j\}$, one just needs to collect the different constraints expressed in Equation 4.1, with a $\vec{y}_j$ and a $\vec{z}_j$ for each P_j, plus the additional constraints $\vec{y}_j.\vec{e}_j \leq \delta$ for all $1 \leq j \leq r$. It remains to find $\vec{c}$ such that $F_i^*(\vec{c})$ is minimized, by solving a linear program, with objective function δ, where all these different variables, $\vec{c}$, δ, $\vec{y}_j$ and $\vec{z}_j$, are unknowns.

Linear Independence and Unimodularity

With no additional constraint, the optimal solution is of course $\vec{c} = \vec{0}$. But $\vec{c}$ should be restricted so that it is linearly independent with all $\vec{c}_j$ with $j < i$, and nonzero if $i = 1$. One way to do this is to complete (e.g., by computing the Hermite normal form [57]), at each step, the vectors $(\vec{c}_j)_{j<i}$, into a n-dimensional basis, with additional vectors $(\vec{d}_j)_{j \geq i}$, and to write $\vec{c} = \sum_{j<i} \lambda_j \vec{c}_j + \sum_{j \geq i} \lambda_j \vec{d}_j$. Then, it is sufficient to solve one linear program for each $j \geq i$ with the additional constraint $\lambda_j \geq 1$, and to define $\vec{c}_i$ as the best of these $(n - i + 1)$ solutions. There is no need to check for $\lambda_j \leq -1$ since $\vec{c}$ and $-\vec{c}$ lead to similar mappings. Another trick is to select a random integer vector $\vec{r}$ and to add a single constraint $\vec{c}.\vec{r} \geq 1$ (or a similar reasoning with the λ_j). Except by bad luck, this would be enough to not miss the optimal with a single linear program. Alternate solutions are possible, e.g., by constructing a basis of the orthogonal of the vector space defined by the

vectors $(\vec{c}_j)_{j<i}$ or a pseudo-inverse (see also the discussion in Section 4.6.1 on the design choices of the Pluto scheduler [15, 1]).

Since $F_i^*(\vec{c}) = F_i^*(\vec{c} - \sum_{j<i} \lambda_j \vec{c}_j)$, one can restrict the search to vectors $\vec{c} = \sum_{j \geq i} \lambda_j \vec{d}_j$, i.e., with $\lambda_j = 0$ for $j < i$. The resulting basis $(\vec{c}_i)_{1 \leq i \leq n}$ (i.e., the matrix M) will then be automatically unimodular (i.e., with determinant ± 1). This property is not formally needed to define a mapping but it makes the construction easier. In particular, completing at each step the vectors into a basis is just one iteration of the Hermite normal form computation. Also, imposing $\vec{c}$ to have integer components is then equivalent to looking for integer values for the λ_j. To see this, suppose that, before Step i of the heuristic, we have built a unimodular matrix U whose first $i-1$ rows (matrix C_{i-1}) are the vectors $(\vec{c}_j)_{j<i}$ built so far, and the remaining rows (matrix D_i) are the vectors $(\vec{d}_j)_{j \geq i}$. At Step i, the chosen solution $\vec{c}_i$ is such that $\lambda_j = 0$ for $j < i$ and the common divisor d of the λ_j, for $j \geq i$, is 1 (otherwise a better solution can be defined by dividing by d). Thus, there is a unimodular matrix V_i of size $(n - i + 1)$ such that $(\lambda_i, \ldots, \lambda_n)$ is the first row of V_i. Then:

$$\begin{pmatrix} I_{i-1} & 0 \\ 0 & V_i \end{pmatrix} \begin{pmatrix} C_{i-1} \\ D_i \end{pmatrix} = \begin{pmatrix} C_{i-1} \\ V_i D_i \end{pmatrix}$$

is a unimodular matrix whose first i rows are the $(\vec{c}_j)_{j \leq i}$. This technique can be used to enforce a final unimodular matrix but also to complete the basis at each step, on the fly.

Handling Parameters

In practice, the set of conflicting pairs and the set of conflicting differences are parameterized by structure parameters from the program, such as loop or array bounds. Heuristic 1 has the nice property that it can easily handle parameters as long as they constrain $\mathcal{K}$ in an affine way. If $P_j = \{\vec{x} \mid Q_j \vec{x} \leq E_j \vec{n} + \vec{e}_j\}$, the standard technique [38] is to bound the width as an affine function of the parameters $\Delta \vec{n} + \delta$ and to apply Farkas lemma considering that $\vec{n}$ is a variable too, possibly taking into account additional affine constraints on the parameters. We then get a set of constraints as before, this time with the variables δ and Δ, with the objective of minimizing $\Delta \vec{n} + \delta$. Classically, parameters are then ordered with some priority so as to optimize the width in a lexicographic manner with respect to this order. The width is then parametric (thus the corresponding modulo), but the vector $\vec{c}$ built at each step has constant components. Thus, all mechanisms presented before for basis completion, linear independence, and unimodularity remain true.

Wrap Up

The optimized version of Heuristic 1 presented here has some similarities with multi-dimensional scheduling and tiling, as explained in Section 4.6.1. It can help finding better mappings than without basis optimization. For example, it finds the right mappings of constant size for the two examples that were designed to show the limitations of reasoning with a fixed basis [32]. However, unlike multi-dimensional scheduling for maximal parallelism detection for which a greedy approach is asymptotically optimal [25], selecting the smallest width at each step can be sub-optimal, even in order of magnitude, because it may be better to first select a direction with larger width if the next width is then smaller in the orthogonal.

This situation arises for the example of Section 4.2: the heuristic simply selects the canonical basis with both widths equal to N, i.e., with no array reuse. See more detailed explanations in Figure 4.2, which depicts the set $\mathcal{K}$ of differences of the live-out points in Figure 4.1. The more complex optimized variant of Heuristic 1 proposed by Bhaskaracharya et al. [13] tries to avoid this caveat, but as explained in Section 4.6.2, it can also fail to find directions of reuse with a small (constant) modulo. The lattice-based heuristic proposed in the next section is aimed to avoid this pitfall.

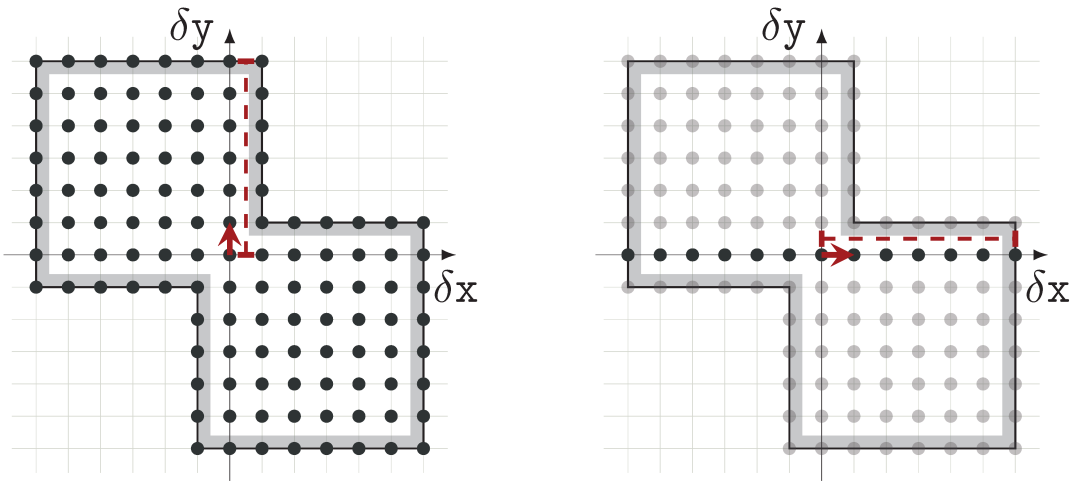


Figure 4.2 – Optimized version of Heuristic 1 for the set $\mathcal{K}$ of differences of the live-out points in Figure 4.1. The selected basis is $\vec{c}_1 = (0, 1)$, then $\vec{c}_2 = (1, 0)$ (or the converse). The left figure shows the width $F_1^*(\vec{c}_1) = N - 1$. Then $\mathcal{K}$ is restricted to the orthogonal space of $\vec{c}_1$, and $F_2^*(\vec{c}_2) = N - 1$ is computed as illustrated on the right. The resulting mapping is $(x, y) \mapsto (y, x) \bmod (N, N)$, which is not a good allocation for this example. This is because the basis vectors are greedily selected to minimize the width at each dimension, missing the concavity along $(1, 1)$.

4.4.2 Basis Selection in Lattice Space

A dual approach to derive a modular mapping $(M, \vec{b})$ is to build an **integer strictly admissible lattice** for $\mathcal{K}$. Given n linearly independent vectors $(\vec{a}_i)_{1\leq i\leq n}$, the lattice generated by $(\vec{a}_i)_{1\leq i\leq n}$ is the set $\Lambda = \{\vec{x} \mid \vec{x} = A\vec{u},\, \vec{u} \in \mathbb{Z}^n\}$ where A is the matrix with column vectors $(\vec{a}_i)_{1\leq i\leq n}$. It is strictly admissible for $\mathcal{K}$ if $\lambda \cap \mathcal{K} = \{\vec{0}\}$. Following Section 4.3.2, a modular mapping is thus valid if and only if its kernel is an integer strictly admissible lattice for $\mathcal{K}$, the set of conflicting differences. Also, from such a lattice Λ, one can build a valid modular mapping whose kernel is Λ and whose size is equal to the determinant of Λ, i.e., $\det(A)$. The smaller is $\det(A)$, the more compact is the allocation. This is the underlying idea of lattice-based memory allocation [31, 32].

The scaling mechanism [31] used in the following heuristic (Heuristic 2) gives a way to build a strictly admissible lattice from a set of n linearly independent integer vectors $(\vec{a}_i)_{1\leq i\leq n}$. Again, as for Heuristic 1, we first want to check that it is still valid for any $\mathcal{K}$, not just for a convex polyhedron $\mathcal{K}$ as initially formulated. Then we want to derive mechanisms to optimize the basis it works with. The connections between the two heuristics, which were proved [31] to lead to the same mapping for a fixed basis and a convex polyhedron $\mathcal{K}$, will be explored later.

Heuristic 2.

- *Choose n linearly independent integral vectors $(\vec{a}_1, \ldots, \vec{a}_n)$.*
- *Compute $F_i(\vec{a}_i) = \inf\{\lambda \geq 0 \mid \vec{a}_i \in \lambda\mathcal{K}_i\}$, successively for all $1 \leq i \leq n$, where $\mathcal{K}_i = \mathcal{K} + Vect(\vec{a}_1, \ldots, \vec{a}_{i-1})$.*
- *Define the lattice Λ generated by the vectors $(\rho_i\vec{a}_i)_{1\leq i\leq n}$ where the ρ_i are integers such that $\rho_i > 1/F_i(\vec{a}_i)$.*

For each i, $\mu\vec{a}_i$ with $\mu = 1/F_i(\vec{a}_i)$ is the largest "multiple" of $\vec{a}_i$ that belongs (if $\mathcal{K}$ is closed) to the **extrusion** $\mathcal{K}_i$ of $\mathcal{K}$ along the vectors $(\vec{a}_j)_{j<i}$, i.e., $\mathcal{K}_i = \mathcal{K} + \mathrm{Vect}(\vec{a}_1, \ldots, \vec{a}_{i-1}) = \{\vec{x} \mid \vec{x} = \vec{y} + \sum_{j<i} \alpha_j\vec{a}_j,\, \vec{y} \in \mathcal{K}\}$. In other words, $\mu\vec{a}_i - \sum_{j<i} \alpha_j\vec{a}_j \in \mathcal{K}$ for some real numbers $(\alpha_j)_{j<i}$ and, for any $\rho > \mu$, $\rho\vec{a}_i \notin \mathcal{K}_i$.

Theorem 7. *The lattice Λ built by Heuristic 2 is a strictly admissible integral lattice for $\mathcal{K}$, if $\mathcal{K}$ is bounded and 0-symmetric.*

Proof. Let $\vec{x} \in \Lambda$. If $\vec{x} \neq \vec{0}$, one can write $\vec{x} = \sum_{j=1}^{i} u_j\rho_j\vec{a}_j$ for some integers $(u_j)_{j\leq i}$ with $u_i \neq 0$. Suppose $\vec{x} \in \mathcal{K}$. If $\mathcal{K}$ is 0-symmetric, one can assume $u_i \geq 1$ without loss of generality. Then $\vec{a}_i = (\vec{x} - \sum_{j<i} \vec{a}_j)/(\rho_i u_i)$, thus $F_i(\vec{a}_i) \leq 1/(\rho_i u_i)$ by definition of $F_i(\vec{a}_i)$,

then $F_i(\vec{a}_i) \leq 1/\rho_i$, which is impossible by definition of ρ_i. Thus, Λ is a strictly admissible integral lattice for $\mathcal{K}$. □

Optimizing the Basis

We now assume that $\mathcal{K}$ is **star-shaped**, i.e., if $\vec{x} \in \mathcal{K}$, then $\lambda\vec{x} \in \mathcal{K}$ for all $0 \leq \lambda \leq 1$. In this case, ρ_i is the smallest integer such that $\rho_i\vec{a}_i \notin \mathcal{K}_i$. Theorem 7 shows that Heuristic 2, like Heuristic 1, is still valid even if $\mathcal{K}$ is not a polyhedron. But how can we optimize the basis $(\vec{a}_i)_{1\leq i\leq n}$? One could think that minimizing $\lceil 1/F_i(\vec{a}_i)\rceil$ is the right thing to do, i.e., that it is sufficient to pick any vector outside the ith extrusion $\mathcal{K}_i$ of $\mathcal{K}$ along the previously-built vectors (in which case $\rho_i = 1$ can be chosen). This is of course not true: unlike for Heuristic 1, the size of the resulting mapping cannot be read directly from the ρ_i, it depends also on the determinant of the $\vec{a}_i$. It seems thus more profitable to directly optimize the basis vectors $\vec{l}_i = \rho_i\vec{a}_i$ of Λ, e.g., by minimizing their norm, for some adequate norm to be defined. This is because the determinant, in absolute value, is bounded by the product of the Euclidian norms (and all norms have the same order of magnitude). Also, if the direction $\vec{a}_i$ is chosen, the best solution is to choose $\vec{l}_i$ to be the smallest integral vector out of the extrusion, in the direction of $\vec{a}_i$.

Another indication of why looking for a small $\vec{l}_i$ is more likely to be good is that each $\vec{l}_i$ is a **reuse vector** (or occupancy vector in the UOV terminology [68]), i.e., $A(\vec{x})$, $A(\vec{x}+\vec{l}_i)$, …, $A(\vec{x}+k\vec{l}_i)$ will reuse the same memory location. The mapping will exploit more reuse if a larger number of copies of $\vec{l}_i$ can traverse the original array space thus, intuitively, a smaller vector will lead to more reuse. However, as is the case for the UOV optimization [68], the best direction remains difficult to anticipate: it depends on the extent of the next extrusion $\mathcal{K}_{i+1}$, i.e., of the other $\vec{a}_j$ not yet defined. Nevertheless, despite this inaccuracy, minimizing the norm is interesting because it will select, in priority, non-parametric reuse vectors, thus favor the reduction of the mapping size in order of magnitude (if there is one large parameter N and p constant reuse vectors are found, the mapping size will be of order N^{n-p}). For all these reasons, at each step of the heuristic, we will look for an integral vector with minimal norm that is out of the extrusion.

For computation reasons, we will choose a norm that can be minimized with linear programming, for example $||.||_\infty$ (max of the absolute value of components) or $||.||_1$ (sum of the absolute value of components, i.e., Manhattan distance). To break ties, we can also use the norm proposed for the computation of AUOV [70], which is a two-dimensional lexicographic optimization, first the Manhattan distance, then the sum of the absolute value of all differences of two components, which tends to lead to more "diagonal" vectors.

Linear Independence and Unimodularity

Unlike Heuristic 1 for which we perform an optimization based on an expression of $\mathcal{K}$, here we want to find a short vector **not in** $\mathcal{K}_i$, or at least on the border of $\mathcal{K}_i$. We tried many different optimization schemes, using Farkas lemma or the duality theorem, but we found no better solution than working directly with the complement of $\mathcal{K}_i$, expressed as a union of polyhedra. Then, we just need to minimize the norm in each piece, with integer linear programming, and to pick the best solution $\vec{l_i}$. Since any linear combination of the vectors $(\vec{a}_j)_{j<i}$ belongs to $\mathcal{K}_i$, linear independence is automatically satisfied.

Now, let us see if we can enforce some unimodularity property and how it can be used to simplify the computations. Suppose the vectors $(\vec{a}_j)_{j<i}$ have been computed so that they can be completed into a unimodular matrix, thanks to additional vectors $(\vec{d_j})_{j\geq i}$. We can look for the vector $\vec{l_i}$ expressed in this basis: $\vec{l_i} = \sum_{j<i} \lambda_j \vec{a}_j + \sum_{j\geq i} \lambda_j \vec{d_j}$. Then, we can either minimize the norm in the original basis (minimizing the components of $\vec{l_i}$) or in this new basis (minimizing the λ_j). In both cases, since $\vec{l_i} \notin \mathcal{K}_i$ then, by definition of $\mathcal{K}_i$, the same is true if we subtract from it any linear combination of $(\vec{a}_j)_{j<i}$. This does not change $F_i(\vec{a}_i)$, the subsequent extrusions, and the determinant of the lattice. We can thus restrict to $\lambda_j = 0$ for $j < i$. Then, if ρ_i is the common divisor of the $(\lambda_j)_{j\geq i}$, we can select $\vec{a}_i = (\sum_{j\geq i} \lambda_j \vec{d_j})/\rho_i$ so that $\vec{l_i} = \rho_i \vec{a}_i$. Finally, as we did for enforcing unimodularity in Heuristic 1, we can then complete $(\vec{a}_j)_{j\leq i}$ into a unimodular matrix, with one simple computation of the Hermite normal form. The basis of the final lattice Λ is then given as a unimodular matrix (A) times a diagonal one (the ρ_i).

Things are also simpler when unimodularity is enforced. A valid mapping σ can be obtained with $M = A^{-1}$ and $\vec{b} = \vec{\rho}$. Indeed, $\sigma(\vec{x}) = \vec{0}$ iff $M\vec{x} = \vec{0} \bmod \vec{b}$ iff there exists $\vec{y} \in \mathbb{Z}^n$ such that $A^{-1}\vec{x} = R\vec{y}$ where $R = \mathrm{diag}(\rho_1, \ldots, \rho_n)$, i.e., $\vec{x} = AR\vec{y}$, which means $\vec{x} \in \Lambda$. When A is unimodular and $\mathcal{K}$ is expressed by a formula involving only integer variables and affine inequalities, it is also simpler to build the set of integral vectors in $\overline{\mathcal{K}_i}$. Indeed, in this case, $\mathcal{K}_i$ is the set of integral vectors in $\mathcal{K} + \{\vec{x} \mid \vec{x} = \sum_{j<i} \alpha_j \vec{a}_j,\ \alpha_j \in \mathbb{R}\}$, i.e., $\mathcal{K}_i = \mathcal{K} + \{\vec{x} \mid \vec{x} = \sum_{j<i} \alpha_j \vec{a}_j,\ \alpha_j \in \mathbb{Z}\}$, which is also defined with integer variables and affine inequalities. This allows $\mathcal{K}_i$, as well as the set of integral vectors in its complement, to be computed by tools such as `isl`.

Handling Parameters

As we saw in Section 4.4.1, Heuristic 1 can be extended with no difficulties to handle the case where $\mathcal{K}$ depends affinely on parameters. The final mapping may depend on these parameters, but not the matrix M, only the modulo vector $\vec{b}$. This is more complicated

for Heuristic 2. If all vectors $(\vec{l}_i)_{1\leq i\leq n}$ are constant or equal to $\lambda\vec{a}_i$ where only λ depends on the parameters, the previous construction can be performed and a valid mapping can be computed from the lattice. However, if this is not the case, i.e., if the direction of $\vec{l}_i$ depends on the parameters, then computing $\mathcal{K}_{i+1}$ involves quadratic constraints (α_i is multiplied by a parameter in the definition of $\mathcal{K}_{i+1}$), which is problematic. Similarly, defining $\vec{a}_i$ and completing the vectors $(\vec{a}_j)_{j\leq i}$ into a basis will need knowledge on the arithmetic properties (gcd) of the parametric components of $\vec{l}_i$.

An easy situation is when all vectors $\vec{a}_i$ built during the optimization process are constant, except possibly the last one. Indeed, for $i = n$, there is no $\mathcal{K}_{i+1}$ to build and no basis completion to perform. This is the case for the example of Section 4.2, where we find successively $\vec{l}_1 = \rho_1\vec{a}_1 = (2, 2)$, then $\vec{l}_2 = \rho_2\vec{a}_2 = (2N-1, 0)$, which corresponds to the 2D mapping $(x, y) \mapsto (y, x - y) \bmod (2, 2N - 1)$. See detailed explanations in Figure 4.3. By construction, the 1D mapping $\sigma'(x, y) = x - y + (2N - 1)y \bmod 2(2N - 1)$ is also a valid mapping [31].

Finally, let us point out that as long as, during the process, we seek a constant reuse vector $\vec{l}_i$, then $\vec{l}_i$ belongs to $\overline{\mathcal{K}_i}$ for any value of the parameters, thus to the intersection of all $\overline{\mathcal{K}_i}$ when the parameters vary, i.e., to the complement of the union of all $\mathcal{K}_i$ when the

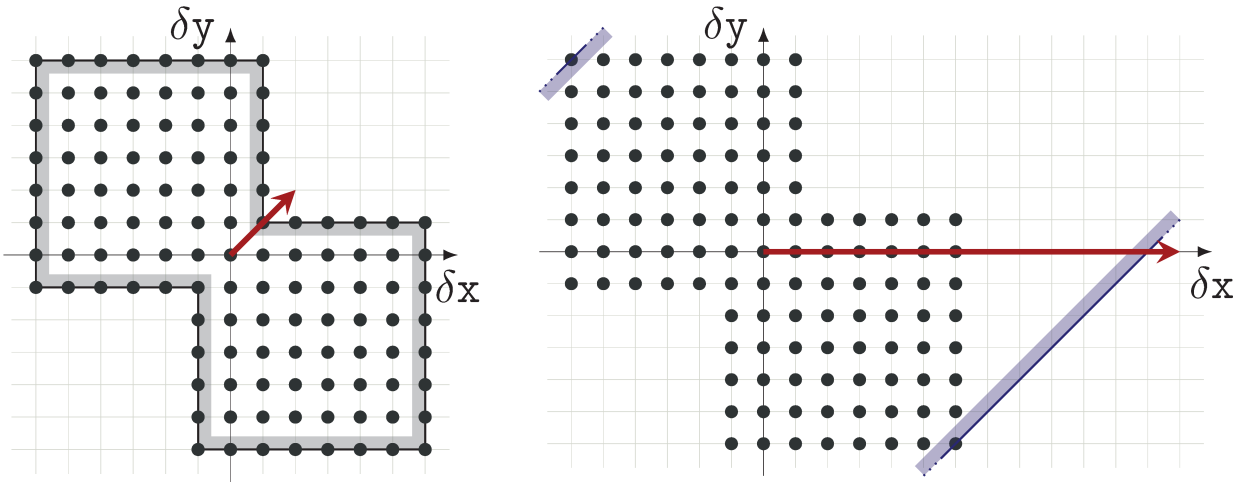


Figure 4.3 – Optimized version of Heuristic 2 for the set $\mathcal{K}$ of differences of the live-out points in Figure 4.1. The selected basis is $\vec{a}_1 = (1, 1)$, then $\vec{a}_2 = (1, 0)$, or equivalently $(0, 1)$. The figure to the left shows the basis vector of the lattice $\vec{l}_1 = \rho_1\vec{a}_1 = (2, 2)$, which is the shortest vector (a multiple of $\vec{a}_1$) that points outside of $\mathcal{K}$. In the next step, the set $\mathcal{K}$ is extruded along $\vec{a}_1$ and is now the infinite diagonal band shown in the right figure, leading to the second basis of the lattice: $\vec{l}_2 = \rho_2\vec{a}_2 = (2N - 1, 0)$. The final mapping is given by the inverse of the matrix $[\vec{a}_1\ \vec{a}_2]^{-1} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & -1 \end{smallmatrix}\right]$ and the modulo factors defined by $\rho_1 = 2$ and $\rho_2 = 2N - 1$, which is the mapping $(x, y) \mapsto (y, x - y) \bmod (2, 2N - 1)$.

parameters vary. This means that we can first project out the parameters in $\mathcal{K}$ (equivalent to the union when the parameters vary), then compute the successive extrusions and complements. In the illustrating example of Section 4.2, projecting out the parameter N in $\mathcal{K}$ leads to the non-parametric complement $\overline{\mathcal{K}}$ of $\mathcal{K}$ defined as a union of two polyhedra $\{(x, y) \mid (x \geq 2,\, y \geq 2)$ or $(x \leq -2,\, y \leq -2)\}$, from which we easily find $\vec{l_i} = (2, 2)$. In general, we can thus first build a non-parametric set $\mathcal{K}$, by projecting out the parameters, and work with it as long as we find a constant reuse vector.

Star Shaping

There remains one potential problem, if $\mathcal{K}$ is not naturally star-shaped. In this case, the set $\mathcal{K}_i$ can have "holes" along a direction and the optimization of the norm in $\overline{\mathcal{K}_i}$ can lead to a small integral reuse vector $\vec{l_i} \in \overline{\mathcal{K}_i}$ such that $\lambda\vec{l_i} \in \mathcal{K}_i$ for some positive integer λ, resulting in an invalid lattice. A possibility is to ignore this problem and to check, a posteriori, that this does not happen. This can be done with integer linear programming if $\vec{l_i}$ is a constant vector.

Another safer strategy is to first modify $\mathcal{K}$ into its star-shaped extension defined as $\mathcal{K}^* = \{\vec{x} \mid \exists\lambda \in [0, 1], \vec{x} = \lambda\vec{y},\, \vec{y} \in \mathcal{K}\}$. This however has to be done with care as an over-approximation of $\mathcal{K}^*$ will exclude valid reuse vectors. Indeed, suppose that $\mathcal{K} = \{(0, 0)\} \cup \{(x, y) \mid 2 \leq |y| \leq 3\}$. The star-shaped extension of $\mathcal{K}$ is the open set $\{(x, y) \mid 0 < |y| \leq 3\} \cup \{(0, 0)\}$, while naively projecting out λ in the first expression of $\mathcal{K}^*$ would add the line $y = 0$. But, as we are interested only in integral vectors, we want to work with $\mathcal{K}^* = \{(0, 0)\} \cup \{(x, y) \mid 1 \leq |y| \leq 3\}$. This can be done as follows. First suppose that $\mathcal{K}$ is the integral points in a polyhedron $\mathcal{P} = \{\vec{x} \mid A\vec{x} \leq \vec{b}\}$. We consider the set of integral points in $\mathcal{K}^* = \{\vec{x} \mid \exists\lambda,\, 0 < \lambda \leq 1,\, A\vec{x} \leq \lambda\vec{b}\}$ to which will be then added the vector $\vec{0}$. We then eliminate λ using the Fourier-Motzkin method, which leads to the following constraints:

$$\mathcal{K}^* = \{\vec{0}\} \cup \left\{ \vec{x} \;\middle|\; \begin{array}{ll} (A\vec{x})_i \leq b_i & \text{if } b_i > 0 \\ (A\vec{x})_i \leq 0 & \text{if } b_i = 0 \\ (A\vec{x})_i < 0 & \text{if } b_i < 0 \\ \dfrac{(A\vec{x})_i}{b_i} \leq \dfrac{(A\vec{x})_j}{b_j} & \text{if } b_j < 0 < b_i \end{array} \right\}$$

The inequality $(A\vec{x})_i < 0$ is then modified in $(A\vec{x})_i \leq -1$ since A and $\vec{x}$ have integral components. As the star-shaped extension of a union of polyhedra is the union of the star-shaped extensions of each polyhedron, the previous technique gives an algorithm to star-shape any union of polyhedra. Note however that, as is the case for the convex

hull, it does not work, in general, for a parametric set as the result is not always affinely parametric. But, as we use this procedure to find constant reuse vectors, we can first project out the parameters from $\mathcal{K}$, then build the star-shaped extension $\mathcal{K}^*$ of this non-parametric $\mathcal{K}$ as we just explained. To illustrate this principle, suppose that $\mathcal{K}$ is defined as $\mathcal{K} = \{(0,0)\} \cup \{(x,y) \in \mathbb{Z}^2 \mid 2 \leq |y| \leq 3, |x| \leq N\}$. Eliminating N produces $\{(0,0)\} \cup \{(x,y) \mid 2 \leq |y| \leq 3\}$, i.e., the set we discussed previously. Finally, with Fourier-Motzkin, we get the star-shaped extension $\{(0,0)\} \cup \{(x,y) \mid 0 < |y| \leq 3\}$, and, restricting to integer points, $\mathcal{K}^* = \{(0,0)\} \cup \{(x,y) \mid 1 \leq y \leq 3\}$ as expected. We can then compute the set of integer points not in $\mathcal{K}^*$ as $\{(x,y) \mid |y| \geq 4\} \cup \{(x,y) \mid |x| \geq 1,\ y = 0\}$.

Wrap Up

The optimized version of Heuristic 2 presented here can be viewed as a generalization of all approaches based on reuse/occupancy vectors: UOV [68], QUOV [83], AUOV [70], and even (pseudo)-projection methods [64]. See more details in Section 4.6.3. It is a strict generalization in the sense that our technique is the first *multi-dimensional* reuse vector technique, thanks to the concept of extrusion. Unlike Heuristic 1, it manipulates $\overline{\mathcal{K}}$, the complement of $\mathcal{K}$, and not $\mathcal{K}$ itself. This duality was also identified in the context of liveness analysis and conflicting pairs, as we showed in Chapter 3. In some specialized contexts (as in all previous work on reuse vectors), the set $\overline{\mathcal{K}}$, or an over-approximation of it, can be built directly, and not as the complement of $\mathcal{K}$.

In Figure 4.4, we describe how the different optimization criteria for the two heuristics favor one set of vectors over the other. It is important to emphasize that the vectors for the two approaches have different meanings. One is the direction of the mapping, and the other is the reuse vector. Providing the equivalent mapping vectors that would be obtained from lattice-based approach, in the reverse order, i.e., $\vec{c}_1 = (1,-1)$ and $\vec{c}_2 = (0,1)$, to the Heuristic 1 yields the same modulos, but computed as widths, thus the same mapping.

Currently, our technique can be fully implemented only when one can guarantee that the reuse vectors produced are constant, i.e., do not depend on parameters. In contrast, Heuristic 1 is naturally adapted to parametric optimization. This motivates the development of a combination of the two heuristics as it will be exposed in Section 4.4.3: we will first look for constant reuse vectors with Heuristic 2, then complete the lattice/mapping by optimizing the dimensions in the opposite order thanks to Heuristic 1, constrained by the reuse vectors already found by Heuristic 2.

Note that when p constant vectors are found, one could re-optimize in this vector space, thanks to a suitable enumeration of lattices using the Hermite normal form [32],

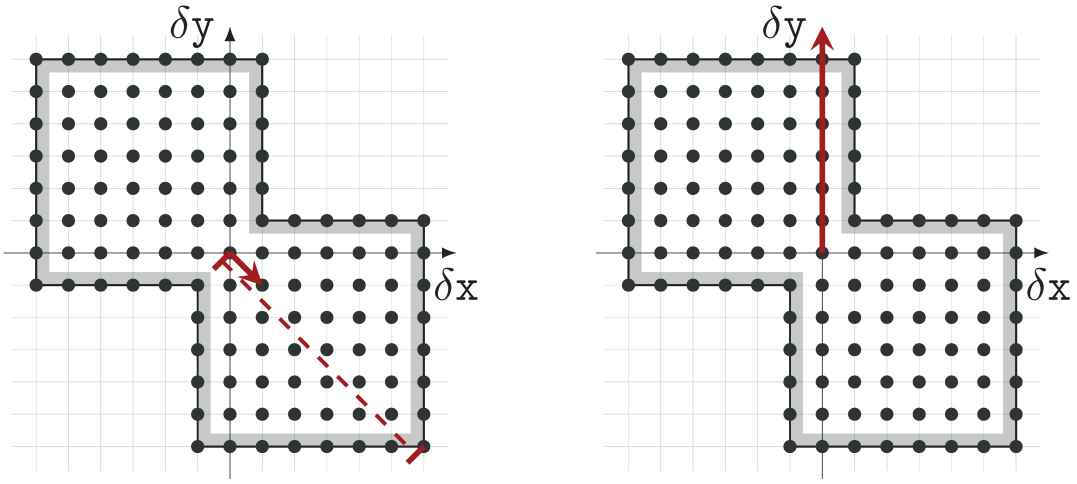


Figure 4.4 – How the basis selections of the two heuristics favor one basis over the other. The basis selection for the mapping space (Heuristic 1) favors, for the first vector, the canonical axis $(1, 0)$ (or equivalently $(0, 1)$) over $(1, -1)$ because the width along $(1, -1)$ is wider. As shown on the left, the width is defined by the vector $(N - 1, 1 - N)$ with a width equal to $2N - 2$, which is larger than $N - 1$ obtained from the canonical basis. In the lattice space, i.e., for Heuristic 2, the canonical basis leads to a much longer reuse vector (it has to be multiplied by $\rho = N$) as depicted on the right. Since a shorter (and constant) reuse vector is preferred in the basis selection, the vector $(2, 2)$ is selected.

to find the best one in this space. However, such a search is potentially expensive if the determinant is large. Also, how to find the best constant strictly admissible sub-lattice in general remains open because we do not know in which subspace the search is to be done.

4.4.3 Combining the Two Approaches

The two previous approaches can be combined by understanding the connection between a valid mapping and an admissible lattice, in particular when the lattice has a "scaled" unimodular basis, i.e., is given by a unimodular basis A multiplied by a diagonal matrix $R = \text{diag}(\rho_1, \ldots, \rho_n)$. If $(\vec{a}_i)_{1 \leq i \leq n}$ are the columns of A, then the $\vec{l}_i = \rho_i \vec{a}_i$ form a basis of reuse vectors for the lattice. If $C = A^{-1}$ is the inverse of A, with row vectors $(\vec{c}_i)_{1 \leq i \leq n}$, then σ defined by $\sigma(\vec{x}) = C\vec{x} \bmod \vec{b}$ (following the notations of Heuristic 1) with $b_i = \rho_i$ is a valid mapping. Furthermore, if $\mathcal{K}$ is the set of integral points in a union of polyhedra, then the b_i obtained with Heuristic 1 for the basis $(\vec{c}_i)_{1 \leq i \leq n}$, from 1 to n, are equal to the ρ_i obtained with Heuristic 2 for the basis $(\vec{a}_i)_{1 \leq i \leq n}$, but considering the vectors in the reverse order, i.e., from n to 1. This can be proved the same way as for a single polyhedron [31], thanks to duality in linear programming. Because of this, we will number the column vectors found by Heuristic 2 in the reverse order, i.e., from $\vec{a}_n$ to $\vec{a}_1$.

Now, suppose that only the last $(n-i)$ column vectors $(\vec{a}_j)_{j>i}$ of A have been computed, for example because we were not able to find an additional constant reuse vector. How can we complete them with i parametric reuse vectors? These $(n-i)$ vectors do not fully constrain the final mapping, they only define a partial mapping with the last $(n-i)$ row vectors $(\vec{c}_j)_{j>i}$ of the inverse of A. However, they fully determine the vector space they generate, as well as its orthogonal, i.e., the vector space where the missing i vectors $(\vec{c}_j)_{j\leq i}$ should lie. The idea is then to use Heuristic 1, constrained to this orthogonal, to optimize them and complete the mapping. This gives rise to the following combined heuristic.

Heuristic 3.

- *Project out the parameters in $\mathcal{K}$ to get a description $\mathcal{K}'$ of the union of constraints valid for all parameters.*
- *Build the star-shaped extension $\mathcal{K}'^*$ of this non-parametric $\mathcal{K}'$, if needed, as explained in Section 4.4.2.*
- *Use the optimized version of Heuristic 2 with $\mathcal{K}'^*$ until no reuse vector is found. The output is a unimodular matrix A with column vectors $(\vec{a}_j)_{1\leq j\leq n}$, such that the $\vec{l}_j = \rho_j \vec{a}_j$, for j from n to $i+1$, are the successively-built reuse vectors.*
- *Use the optimized version of Heuristic 1 with the initial set $\mathcal{K}$, restricting the search to the space orthogonal to all $(\vec{a}_j)_{j>i}$, to get i optimized mapping vectors $(\vec{c}_j)_{j\leq i}$. Let $(w_j)_{j\leq i}$ be their corresponding successive widths.*
- *Define the matrix M with row vectors $(\vec{m}_j)_{1\leq j\leq n}$ such that $\vec{m}_j$ is the j-th row of A^{-1} if $j > i$, and $\vec{m}_j = \vec{c}_j$ if $j \leq i$. Define $\vec{b}$ with $b_j = \rho_j$ if $j > i$ and $b_j = w_j$ if $j \leq i$.*

Theorem 8. *The mapping $\sigma = (M, \vec{b})$ is a valid mapping for $\mathcal{K}$.*

Proof. Let $\vec{x} \in \mathcal{K}$ such that $M\vec{x} = \vec{0} \bmod \vec{b}$. With the same argument used in the proof of Theorem 6, and by definition of $(\vec{m}_j)_{j\leq i}$ in Heuristic 3 and of $(b_j)_{j\leq i}$ in Heuristic 1, we first get $\vec{c}_1.\vec{x} = 0$, then successively $\vec{c}_j.\vec{x} = 0$ for all j from 1 to i. Thus $\vec{x}$ belongs to the orthogonal of $(\vec{c}_j)_{j\leq i}$, i.e., to the vector space spanned by $(\vec{a}_j)_{j>i}$. As A is unimodular, we can write $\vec{x} = \sum_{j>i} \lambda_j \vec{a}_j$ for some integers λ_j. As the last $(n-i)$ rows of M are those of the inverse of A, $\vec{m}_j.\vec{x} = \lambda_j$ for $j > i$. Thus λ_j is a multiple of $b_j = \rho_j$ as defined in Heuristic 2. With the same arguments used in the proof of Theorem 7, we conclude that $\vec{x} = \vec{0}$, which means that the modular mapping defined by M and $\vec{b}$ is valid. □

To force the search in the orthogonal of the $(\vec{a}_j)_{j>i}$, we can look for an integer linear combination of the first i rows of A^{-1} only. Then, using the same principles as for

Heuristic 1, we can impose that the vectors $(\vec{c}_j)_{j\leq i}$ form a unimodular basis of this orthogonal, which, combined with the last $(n-i)$ rows of A^{-1}, will form an $n \times n$ unimodular matrix M. There is thus in this case a complete correspondence between the b_i found through Heuristic 1 for the row vectors of M, starting from the first one, and the ρ_i found through Heuristic 2 for the column vectors of M^{-1}, starting from the last one. By limiting the search of $(\vec{c}_j)_{j\leq i}$ to the orthogonal of the small reuse vectors $(\vec{a}_j)_{j>i}$ found, we may find larger widths than without this constraint. But this is on purpose, to make sure we keep the good directions of constant reuse. This is what happens on the example of Section 4.2 if we apply the combined heuristic. We first find the reuse vector $(2, 2)$, then we limit the search to the orthogonal direction. Here, in 2D, we directly get the mapping vector $(1, -1)$. In general, this principle remains a heuristic as the orthogonal of the reuse vector(s) may not be the right space in terms of width. But it is more likely to lead to the right size in order of magnitude because it guarantees i non-parametric modulo factors. This concludes the formal description of our combined optimization.

4.5 Evaluation

We validated our technique by running the different optimization steps with the `iscc` [74] calculator, which offers, through scripts, some of the functionalities of `isl`. As `iscc` does not provide sufficient genericity, the scripts had to be tailored to each example but a generic implementation could be done using `isl` directly.

4.5.1 Reverse-L Shaped Region and Optimizing Scripts

Our illustrating example can be solved using the following script:

```
Kp := {[N]->[x,y]: N>=2 and
    (-1<=-x,y<N or -1<=x,-y<N)};
K_1 := range Kp;
nK_1 := {[x,y]} - K_1;
Norm := { [x,y]->[m,-x,-y]: m >= x,y,-x,-y };
a_1 := (Norm^-1)(lexmin (Norm(nK_1)));
```

`Kp` represents $\mathcal{K}$, with the exception that its parameters are variables (to circumvent the fact that `iscc` cannot project parameters out). We then build `K_1` by projecting these parameters. A star-shaping step should also take place (but again is not easy to do with `iscc`) but would give the same polyhedron as it is already star-shaped (because $\vec{0}$ is in

each basic set). We then compute `nK_1` as the complement of `K_1`. The `Norm` map provides the $||.||_\infty$ norm, which we want to minimize (the $||.||_1$ norm would give the same result here as the optimal solution is a vertex of `nK_1`). The negation of the x and y coordinates is cosmetic and is so that we select the smallest lexicographically nonnegative vector as a side effect of minimization. We then proceed to find the smallest vector, which, in our example, leads to $\vec{l_1} = (2, 2)$ as expected, providing our first lattice vector $\vec{a}_1 = (1, 1)$ and its associated modulo 2.

We then complete the vector $\vec{a}_1$ into a unimodular basis, for example with the vector $(1, 0)$, and we search for the next vector:

```
Extr_1 := {[x,y]->[x',y']:
    (exists e: x'=x+1*e and y'=y+1*e)};
Unimod_1 := range {[t]->[1*t,0*t]: t>0};
K_2 := coalesce Extr_1(K_1);
nK_2 := {[x,y]}-K_2;
a_2 := (Norm^-1)(lexmin (Norm(nK_2*Unimod_1)));
```

`Extr_1` is an extrusion operator along $\vec{a}_1$ and `Unimod_1` is a constraint enforcing the unimodularity of the future basis. It is obtained by looking for vectors colinear to the vectors (here only $(1, 0)$) that complete $\vec{a}_1$ into a unimodular basis. In this example, `K_2` is now the whole space, due to the parameter projection, so `nK_2` is empty, and we do not find a second constant reuse vector to complete the first. If the parameters were not projected (and if `K_2` was detected as already star-shaped), we would have found $\vec{l_2} = (2N - 1, 0)$, which is a valid parametric reuse vector (see again Figure 4.3).

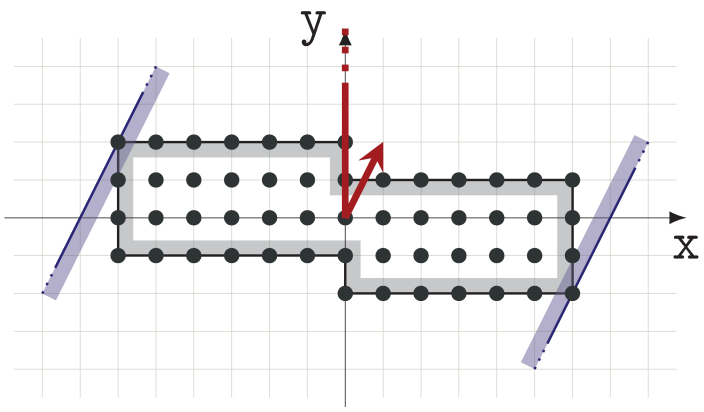
Now, let us assume that we did not find this last vector because we stopped Heuristic 2 as soon as it does not find a constant vector. We then compute the inverse of the unimodular extension of $\vec{a}_1$ (with this vector as last column): $\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix}$. This already gives the last vector $\vec{c}_2 = (0, 1)$ of the final mapping and its associated modulo equal to 2. We then continue with Heuristic 1:

```
K := Kp([N]->{[N]});
fK := (unwrap coefficients K)^-1;
Opt := {[c_cst,c_N]->[c_N,c_cst]};
Ortho_1 := range {[t]->[1*t,-1*t]: t>0};
b := (fK.Opt)*Ortho_1;
b_min := lexmin range b;
c_1 := sample (b^-1)(b_min);
```

The set `K` is the parameterized `Kp` and `fK` is the dual of `K` in the sense of Farkas lemma. `Opt` is used to order the coefficients in the correct order for the minimization: we want a small coefficient for N, then a small constant term. `Ortho_1` constrains the vector to the space orthogonal to $\vec{a}_1$ (or equivalently the space that completes $\vec{c}_2$ into a unimodular matrix), i.e., it is the first row of the inverse we computed. `b` combines these operators and is a map that gives, for a given vector, the width of the symmetric conflict set along this direction. For example, `lexmin (fK.Opt)([0,1])` gives `[1,0]`, which represents $1 \times N + 0$ in the Farkas space. We then compute the smallest width, `b_min`, and proceed to find a vector that attains it, here $\vec{c}_1 = (1, -1)$. The smallest modulo is $2N - 1$, as `b_min` $= [2, -2] \mapsto 2N - 2$ is reached and we need a modulo strictly larger.

4.5.2 Blur Filter

On the blur filter with an interleaved schedule [13], we find the optimal allocation provided that we optimize both for the Maximum distance $||.||_\infty$ and the Manhattan distance $||.||_1$, as discussed in Section 4.4.2. Indeed, as shown in Figure 4.5, $(1, 2)$ is equivalent to $(2, 2)$ for $||.||_\infty$ and to $(0, 3)$ for $||.||_1$. With this optimization, we get the desired allocation of $blurx[(y - 2x) \bmod (2N + 1)]$.

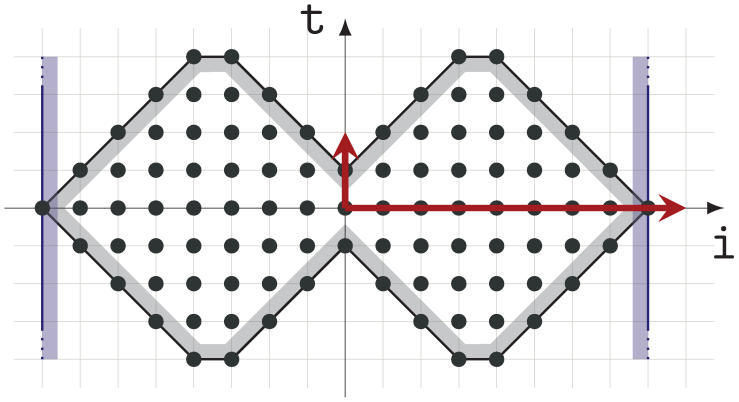


$$
\begin{aligned}
N \to \{\, (x, y) \mid\ & (x \geq 1 \wedge x < N \wedge y \leq 1 \wedge y \geq 0) \\
& \vee (x \geq 0 \wedge x < N \wedge y \leq -1 \wedge y \geq -2 \wedge y > -N) \\
& \vee (x \leq 0 \wedge x > -N \wedge y \geq x \wedge y \leq 2 \wedge y \geq -1 \wedge y < N) \,\}
\end{aligned}
$$

Figure 4.5 – Conflict differences for blur filter (interleaved schedule).

4.5.3 Diamond Tiling

Another interesting example is the diamond tiling example [13] whose conflict set is shown in Figure 4.6. The technique of Bhaskaracharya et al., analyzed in Section 4.6.2, finds an allocation of size $6B-5$ with the mapping allocation $A_B[(t-3i) \bmod (6B-5)]$, where B is the tile size. Our algorithm finds the mapping $A_B[t \bmod 2, i \bmod (2B-1)]$, a mapping aligned with the canonical basis that Lefebvre and Feautrier [54] thus also find (or equivalently Heuristic 1) if i is the first canonical dimension, and t the second. In our case, our combined heuristic does not fall in the trap: the first lattice vector found is the vector $(0,2)$, i.e., $\vec{a}_1 = (0,1)$ and a modulo of 2, then the second vector is a mapping vector $(1,0)$ with a modulo of $2B-1$, so the canonical basis in the right order is found.



$$
\begin{aligned}
B \to \{\, (t,i) \mid\ & (i \le 0 \wedge t \le 1-i \wedge t \ge -1+i \wedge t \le 2B+i \wedge t \ge -2B-i) \\
& \vee (i \ge 0 \wedge t \le 1+i \wedge t \ge -1-i \wedge t \le 2B-i \wedge t \ge -2B+i) \\
& \vee (i \le B \wedge i \ge -B \wedge t \le 1-i \wedge t \ge -1-i \wedge t \le B \wedge t \ge -B) \\
& \vee (i \le B \wedge i \ge -B \wedge t \ge -1+i \wedge t \le 1+i \wedge t \le B \wedge t \ge -B)\,\}
\end{aligned}
$$

Figure 4.6 – Conflict differences for diamond tiling.

4.5.4 Comparison of Our Mappings with Other Work

We compare allocations by our approach with prior work in Table 4.1. The examples are taken from the work by Bhaskaracharya et al. [13], except for the last two examples. The `head-2d-tiled` example is an allocation for a tile of the 2D heat equation, tiled after a skew by $(t,i,j) \mapsto (t, t+i, t+j)$. All the examples were computed by optimizing for the $||.||_\infty$ norm, and then for the $||.||_1$ norm.

Table 4.1 – Comparison of mappings found by the successive modulo technique [54] (LeFe), Bhaskaracharya et al. algorithm [13] (BBC), and our approach with Heuristic 3 (Lattice). The scripts take less than a second for each example with 2GHz CPU and 1GB RAM.

Example	Algorithm	Reuse Vector	Mapping	Reduction*
blur-interleaved	LeFe		$(y,x) \mapsto (y,x) \bmod (3,N)$	1
	BBC		$(y,x) \mapsto (2x-y) \bmod (2N+1)$	3/2
	Lattice	$(1,2)$	$(y,x) \mapsto (y-2x) \bmod (2N+1)$	3/2
blur-tiled	LeFe		$(x,y) \mapsto (x,y) \bmod (B,B)$	1
	BBC		$(x,y) \mapsto (y-2x) \bmod (3B-2)$	$B/3$
	Lattice	$(1,2)$	$(x,y) \mapsto (y-2x) \bmod (3B-2)$	$B/3$
LBM-D2Q9	LeFe		$(t,i,j) \mapsto (t,i,j) \bmod (2,N,N)$	1
	BBC		$(t,i,j) \mapsto (i-2t,j) \bmod (N+2,N)$	2
	Lattice	$(1,1,1)$	$(t,i,j) \mapsto (i-t,j-t) \bmod (N+1,N+1)$	2
LBM-D3Q19	LeFe		$(t,i,j,k) \mapsto (t,i,j,k) \bmod (2,N,N,N)$	1
	BBC		$(t,i,j,k) \mapsto (i-2t,j,k) \bmod (N+2,N,N)$	2
	Lattice	$(1,1,1,0)$	$(t,i,j,k) \mapsto (j,i-t,k-t) \bmod (N,N+1,N+1)$	2
LBM-D3Q27	LeFe		$(t,i,j,k) \mapsto (t,i,j,k) \bmod (2,N,N,N)$	1
	BBC		$(t,i,j,k) \mapsto (i-2t,j,k) \bmod (N+2,N,N)$	2
	Lattice	$(1,1,1,1)$	$(t,i,j,k) \mapsto (k-t,i-t,j-t) \bmod (N,N+1,N+1)$	2
diamond-tile	LeFe		$(t,i) \mapsto (t,i) \bmod (B,2B-1)$	1
	BBC		$(t,i) \mapsto (t-3i) \bmod (6B-5)$	$B/3$
	Lattice	$(2,0)$	$(t,i) \mapsto (i,t) \bmod (2B-1,2)$	$B/2$
Example in Figure 1	LeFe/BBC†		$(x,y) \mapsto (x,y) \bmod (N,N)$	1
	Lattice	$(2,2)$	$(x,y) \mapsto (x-y,y) \bmod (2N-1,2)$	$N/4$
heat-2d-tiled	LeFe/BBC†		$(t,i,j) \mapsto (t,i,j) \bmod (B,B,B)$	1
	Lattice	$(1,1,1)$	$(t,i,j) \mapsto (i-j,j-t) \bmod (2B-1,3B-2)$	$B/6$

* The reduction over the successive modulo technique (LeFe) in order of magnitude with respect to the parameters (the larger, the better).

† BBC finds different allocations depending on the decomposition of the conflict polyhedra. Using a natural decomposition with the lexicographic order, it gives the same mappings as those given by the successive modulo technique for these two examples. With some other decompositions, it may find mappings similar to ours. See Section 4.6.2 for detailed discussion.

4.6 Related Work

There are a number of existing techniques for memory allocation in the polyhedral model, i.e., for programs on which code analysis and optimizations based on manipulations of polyhedra and linear programming techniques can be applied [13, 31, 54, 64].

The techniques we described in Section 4.4 extends the framework of Darte et al. on lattice-based memory allocation [31, 32], both to handle (non-convex) unions of polyhedra and to improve the choice of projections, i.e., mapping functions. In this respect, the work by Lefebvre and Feautrier [54] can be viewed as a special case of lattice-based allocation (more precisely in the form of Heuristic 1, i.e., in the space of mappings), where only a subset of the mapping space is used (and not optimized), i.e., it works for a fixed basis.

The technique proposed by Quilleré and Rajopadhye [64], on the other hand, is more similar to Heuristic 2, in the sense that it explores the lattice space, characterizing legal projective allocations (possibly with modulo reuse), including projections along non-canonical directions. However, their primary objective is in minimizing the dimensionality of the resulting projection, and no algorithm is available to search among legal projections. Also, it can be used only with strong hypotheses, in particular for multi-dimensional schedules. Nevertheless, in this limited context, it does find constant reuse vectors when the conflict set is "flat", i.e., not fully-dimensional. Our optimized technique with successive extrusions generalizes this approach to a broader context.

Our technique has strong links with the search for multi-dimensional affine functions for scheduling, in particular for detecting tiling bands in nested loops (see Section 4.6.1). Our optimized version of Heuristic 1 uses similar linear programming techniques, as does the recent work of Bhaskaracharya et al. for intra-array reuse [13]. This latter work has the same objectives as ours, but it uses a different approach to tweak Heuristic 1 and try to avoid a N^2 mapping in the main example of Section 4.2. We explain in Section 4.6.2 why this approach still has some weaknesses, at least for intra-array reuse. The way we tweak Heuristic 1, with the help of Heuristic 2, reveals new connections with early work on universal occupancy (reuse) vectors as we explain in Section 4.6.3.

4.6.1 Link with Multi-Dimensional Scheduling and Tiling

It is interesting to note the strong link with the algorithm used in the Pluto compiler [15, 1] to build code transformations enabling tiling. In Pluto, operations in nested loops, each captured by a textual statement S and a loop counter (or iteration) vector $\vec{i}$, are reordered by defining affine mappings $\sigma_S(\vec{i})$ such that $\sigma_T(\vec{j}) - \sigma_S(\vec{j}) \geq 0$ whenever there is a dependence from $(S, \vec{i})$ to $(T, \vec{j})$. The objective function is to minimize the

distance between these operations (this tends to improve data reuse), i.e., the maximum of all $\sigma_T(\vec{j}) - \sigma_S(\vec{i})$, as we do for the width computation in Heuristic 1.

However, there are three main differences compared to the scheduling algorithm. The first difference is that we use a single σ, i.e., $\sigma_T = \sigma_S$, so that we can work with the difference $\vec{j} - \vec{i}$. But we could also apply Heuristic 1 to map different arrays in the same memory space, each with a possibly different mappings, as explored in SMO [14]. The second difference is that, at each step, we remove all pairs such that $\sigma(\vec{j}) - \sigma(\vec{i}) = \vec{0}$ (as in the search for maximal parallelism [38]) while, in Pluto, dependences need to be kept for defining other dimensions for tiling. The third difference is that there are no constraints such as $\sigma(\vec{j}) - \sigma(\vec{i}) \geq \vec{0}$ as conflicting differences can have any sign. In other words, the technique is similar but we will get a smaller width at each step (because of fewer constraints).

We can import all tricks used in Pluto [15] and Pluto+ [1], in particular for enforcing linear independence. However, since our situation is simpler, the techniques we use are sufficiently cheap in our case. Nevertheless, this shows a strong link between multi-dimensional scheduling/tiling and array mapping: the former uses dependences, and the latter uses pairs of conflicting elements. The fundamental difference is that the pairs of conflicting differences are not directed and can thus be "satisfied" by a mapping (i.e., $\sigma(\vec{i}) \neq \sigma(\vec{j})$) either as a positive or a negative value.

4.6.2 Link with Bhaskaracharya et al. Intra-Array Reuse

Heuristic 1 is the natural extension of Lefebvre-Feautrier successive modulo technique [54] with a greedy optimization of the basis. As we already mentioned, it has some similarities with multi-dimensional scheduling and tiling.

While optimizing the basis is usually not needed when $\mathcal{K}$ is convex (at least in order of magnitude) [32], this is not the case anymore when $\mathcal{K}$ is a union of polyhedra, and different bases can give different mapping sizes in order of magnitude. In particular, Heuristic 1 may suffer from the fact that, at each step, the optimization is equivalent to considering the convex hull of the intersection of $\mathcal{K}$ with the orthogonal of the previously-built vectors. Also, minimizing the width to get a small modulo may induce a bad choice for the next steps. This is our motivation, shared with Bhaskaracharya et al. [13], to find a mechanism to force Heuristic 1 to *not* choose an hyperplane with smallest width, i.e., smallest modulo. Our result is Heuristic 3, and we now highlight the differences with respect to the work by Bhaskaracharya et al. [13] in the following.

Their approach is formulated with a relation describing pairs of conflicting elements $(\vec{i}, \vec{j})$

instead of conflicting differences $\vec{i} - \vec{j}$ as we do. For intra-array reuse, only one σ is searched, so this is equivalent. We thus rather explain their technique with conflicting differences to simplify the discussions and visualizations.

As we do, they assume that the conflict set $\mathcal{K}$ is given by the integer points in a union of polyhedra $\mathcal{K} = \cup_{1 \leq j \leq r} \mathcal{P}_j$, but $\mathcal{K}$ is only half of the conflict to exclude $\vec{0}$ (as it belongs to any vector hyperplane) and to make it asymmetric[1]. Then, instead of searching for a vector hyperplane with minimal width, they search for one that intersects as few polyhedra $\mathcal{P}_j$ of the union as possible, i.e., they are fully on one side of the vector hyperplane: either $\sigma(\vec{x}) \geq 1$ for all $\vec{x} \in \mathcal{P}_j$ or $\sigma(\vec{x}) \leq -1$ for all $\vec{x} \in \mathcal{P}_j$. Among such solutions, the one with smallest width is chosen. If such a hyperplane is found for all j, it forms a valid one-dimensional allocation. When such a hyperplane does not exist, a second hyperplane is computed, focusing on the intersection of $\mathcal{K}$ with the orthogonal to the first hyperplane (as in Heuristic 1), and so on until all the conflicts are resolved with these separating hyperplanes.

Although their algorithm may work well in many cases, a careful look reveals certain situations where their approach misses good allocations, sometimes even using higher-dimensional array than necessary. The two main problems are that, as is the case for Heuristic 1, minimizing the moduli in such a greedy fashion is not always good, and, more importantly, the resulting allocation is dependent on the way the conflict set is decomposed into a union of polyhedra, due to their primary objective defined at the granularity of the constituting polyhedra. This makes the approach quite unstable.

Figure 4.7 illustrates the problem with their heuristic, and how the decomposition may influence the allocation. In contrast, our approach finds better allocations for these examples by using a different objective function that does not rely on the way the conflict set is represented as a union of polyhedra. The key point is that, despite the link between multi-dimensional scheduling and affine mapping explained in Section 4.6.1, there is one fundamental difference. In scheduling, due to the constraint $\sigma(\vec{i}) - \sigma(\vec{j}) \geq 0$, all dependences are made nonnegative (weakly separating hyperplane) and some are made positive $\sigma(\vec{i}) - \sigma(\vec{j}) \geq 1$ (separating hyperplane). If a first schedule σ_i separates $\mathcal{P}_i$ and a second schedule σ_j separates $\mathcal{P}_j$, then $\sigma_i + \sigma_j$ is a also a schedule that separates both $\mathcal{P}_i$ and $\mathcal{P}_j$. This is the reason why a greedy separating approach [38] is optimal for detecting maximal parallelism. Here, this is not true because separation can be ≥ 1 or ≤ -1, which breaks the analogy.

[1]How this is performed is not detailed. We assume that it is done by taking the intersection with the strict lexicographic order, in some basis.

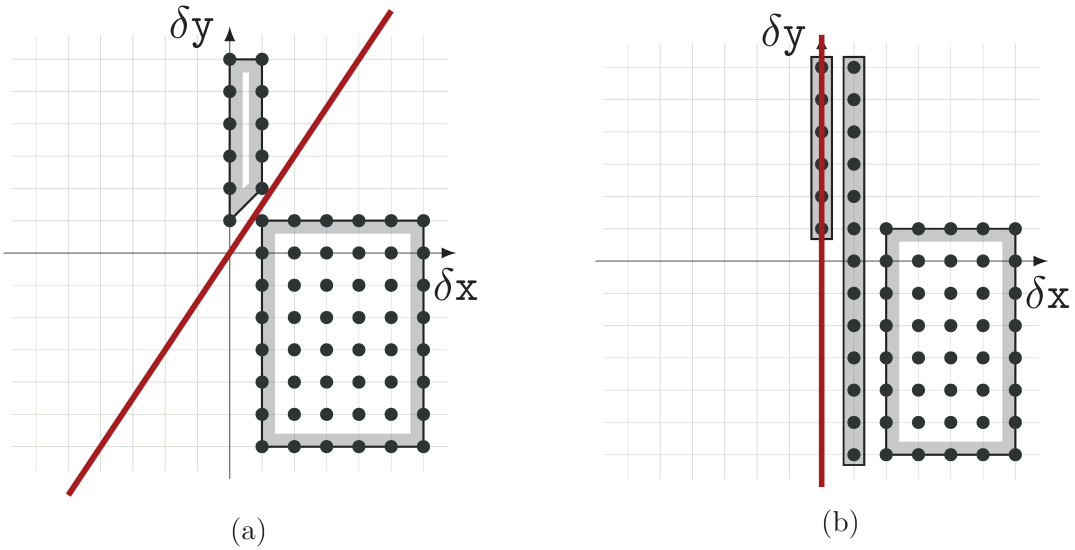


Figure 4.7 – How the technique by Bhaskaracharya et al. [13] encounters difficulties with the example from Figure 4.1. Figures 4.7a and 4.7b are two possible decompositions of the asymmetric version of the conflict set. In Figure 4.7a, a hyperplane exists that does not intersect with the entire conflict polyhedra, satisfying their primary objective function. This leads to N extra storage as we illustrated in Section 4.2. Figure 4.7b is another possible decomposition where no hyperplane can partition the entire conflict set. In this case, their primary objective to maximize the number of polyhedra (in the disjoint union) that are fully on one side of the hyperplane, in combination with the secondary objective to minimize the width of a dimension in the final array, gives the vertical line. This produces N^2 storage in the end, worse by one full dimension than what we achieve.

4.6.3 Link with Universal Occupancy Vectors

We already briefly mentioned in Chapter 3 (Section 3.4) the concept of occupancy vectors. An occupancy vector is a vector that points to another iteration that can safely reuse the memory location. Universal occupancy vectors (UOVs) are those valid for any legal schedule of a program [68]. For UOVs, the formulation is as follows. An iteration point $\vec{v}$ may overwrite a value produced at another iteration $\vec{u}$ if $\vec{v}$ (transitively) depends on all uses of $\vec{u}$. Since all uses of $\vec{u}$ must be executed for $\vec{v}$ to execute without violating dependences, the allocation is valid for all possible schedules. Constraints for expressing UOVs are exactly the complement of constraints for expressing conflicts, in the context of schedule-independent mappings. The theory of UOV is however rather limited, as only codes with uniform dependences and constant UOVs are captured. Otherwise, the set of legal UOVs is hard to define. Thies et al. [70] showed how to handle affine dependences, but they had to restrict to the space of legal affine one-dimensional schedules so that the set of all constant valid occupancy vectors (called AUOV, for affine UOV) can be defined.

Although allocations legal for all possible schedules may seem too conservative, UOVs can still give efficient allocations for many programs. In fact, it is no coincidence that UOV-based allocations (and in particular QUOV-based allocations [83], designed for tiling) find good allocations for live-out values of tiled stencil programs [13]. If we look at an allocation for a tile, the live-out values are those that must be preserved at the end of the execution of a tile. These values should never be overwritten since there are other uses outside of a tile. Thus, even if the tile itself has a specific schedule, live-out variables are captured well with schedule-independent mappings, e.g., with UOV-based allocations. These exploit possible reuse within the tile while keeping the live-out values.

Partially due to the restriction of UOVs that only use one projection (they thus lead to a $(d-1)$-dimensional array for a d-dimensional iteration space) the primary heuristic used in the search for a UOV is the length of the UOV [68]. The gcd of the UOV is directly connected to the memory consumption through modulo factors along the projection. Increases in the UOV length that do not influence the gcd often increase the memory usage by making the projection to be more steeply angled. For example, projecting a $N \times N$ domain along the vector $(1, 0)$ gives a line of length N, while a diagonal projection along $(1, 1)$ gives $2N - 1$. If such a projection is along some boundary of the domain, it may decrease the memory usage, but the length of the UOV is a good approximation otherwise. This shares the same idea with the heuristic used in our approach.

4.7 Conclusion and Future Work

In this chapter, we presented our results, published at the CC'16 conference [28], on how to extend the lattice-based memory allocation in two important directions: unions of polyhedra, and better objective functions to find the basis vectors. The key insight is taken from the two dual approaches in the original lattice-based method, the first working with hyperplanes on one side and the second with reuse vectors on the other side. We have shown that reuse vector based selection of the basis of the mappings finds more compact mappings with different examples.

Several research directions remain to be explored after this work.

- Heuristic 2 may fail finding the maximal number of constant modulo factors and, because of this, Heuristic 3 may be sub-optimal, even in order of magnitude. Can we solve this issue?
- While Heuristic 1 can be generalized to inter-array optimizations with different arrays sharing a common space with different affine mappings, as Bhaskaracharya

et al. did for their intra-array mapping [14], it is unclear how to extend our method based on reuse vectors and conflict differences.

- Although we have focused on compactness of the allocation in this chapter, it is not always the best for performance. All prior work on memory allocation in the polyhedral literature have never taken locality into account. An important direction to explore is how we can explore memory layout transformations for improved performance, e.g., through more cache-friendly layouts, and in particular cases where introducing redundant storage leads to better overall performance.

In this chapter, we have separated the construction of conflict sets as an orthogonal component, since our method works for conflict sets that may come from many different inputs. In addition to the classical conflict sets computed based on multi-dimensional affine schedules, we may take those from other specifications, including explicitly-parallel specifications (such as OpenMP, X10, OpenStream, and so on), or software pipelining used to overlap computation and I/O, as exposed in Chapter 3. Memory reuse analysis is important in many different scenarios, and now we have provided the theory that seamlessly generalizes across different cases. As an illustration, the different steps (definition of the pipeline schedule, computation of the transfer sets, of the conflict sets, and finally of the mapping) of the particular kernel offloading with tiling studied in Chapter 2 are described in more details in Chapter 5.

Chapter 5

Kernel Offloading

Summary

Kernel offloading in its largest acceptation is a code transformation that consists in isolating a computationally-intensive kernel from a larger program in order to offload its computation to a specialized architecture. In terms of analysis, it is the opposite of function inlining, thus it is a form of function outlining. It involves computing live-in and live-out memory accesses (to produce the memory transfers from and to the accelerator), possibly pipelining computation and communication (to hide the latency of these transfers), and optimizing the code for the new architecture (taking into account parallelism and synchronization capabilities), all of that constrained by the memory organization of the accelerator(s) and host.

In this thesis, we studied a particular form of kernel offloading strategy, relying on a form of recursive parametric tiling to decompose the kernel into block of computations, where each level of tile corresponds to a memory layer. Intermediate tiling levels might be inserted in order to reuse local storage between successive tiles (i.e., executed in sequence) at a given memory layer. We developed several concepts to make this possible, how to analyze inter-tile data reuse for parametric tiling in Chapter 2, how to analyze the resulting liveness conflicts between array elements in Chapter 3, a preliminary analysis to be able to map data in local memories thanks to array contraction, a technique that we revisited in Chapter 4. This chapter aims to put all these steps together, not with a fully implemented solution yet, but with a presentation of our design choices and methodology (e.g., parametric tiling, software pipelining), an illustration through small (but already complex) kernels on the type of results we can obtain with the conflict analysis and mapping techniques studied in previous chapters, and a discussion on our vision and attempts on how all this could be implemented to GPUs, as an extension of the PPCG compiler.

5.1 Motivation

One of our main motivations to study parametric tiling was to have a mean to explore a wide range of alternate solutions, through cost models and tile sizes selection, with or without pipelining, with or without reuse, with or without memory constraints, with or without live-range splitting of array elements, etc. Tile size selection is driven by many constraints, one is the memory footprint. When a kernel is limited by data transfers (bandwidth-bound kernels), the tile size impacts the amount of communication (thanks to data reuse) and therefore the performance. Figure 5.1 represents the expected effects of the different analyses and optimizations, and thus the trade-offs we wanted to explore or at least enable to explore.

On the top left is the original code, without tiling, and where memory accesses are always performed through remote communications. It has the minimal memory footprint as the only data that need to be stored locally are those involved in the current statement executed. It has also the worst execution time as, even assuming perfect pipelining, we are limited by the throughput at every level, and data are even stored back to external memory after each update. On the other end of the spectrum is the code where we send the whole data to the accelerator and perform the whole computation locally, before updating back the external memory afterwards. This requires a lot of memory on the accelerator

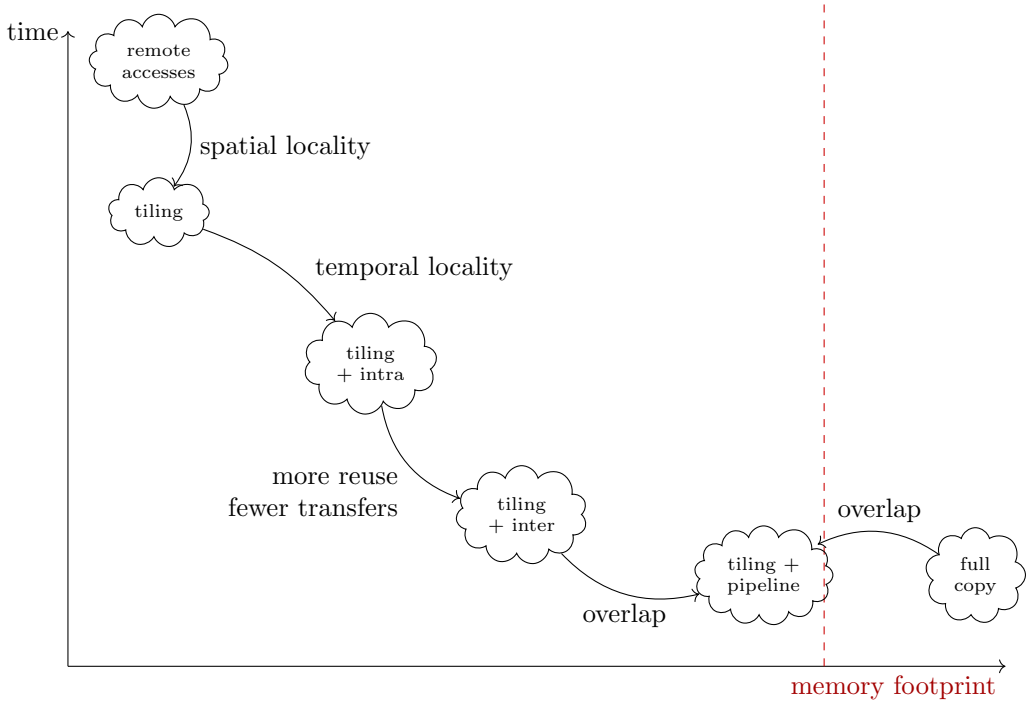


Figure 5.1 – Relative expected performance effect of different optimizations.

(a full copy of the portion of the external memory concerned with the computation, either as input or output data), but it induces the minimal amount of communication. It is usually not possible to pipeline these communications with the computations (unless these computations are cut into blocks, which goes back to tiling) and we might thus pay the full communication latency.

Still on the same figure, on the far left is represented the expected effect of tiling (only as a loop transformation, not with any coalescing of transfers). There, the memory accesses are still done remotely (we do not cache the data), hence the memory footprint stays the same. Even then, this usually improves the performance. Indeed, tiling changes the execution order and usually improves spatial locality, as memory transfers usually send a small region around the requested data in any case (a cache line or memory line), which can diminish the amount of communications for free if these data are indeed useful. This can also enable hardware prefetching or enforce some order of accesses to the external memory and thereby reduce row switching (precharge/activation cycles) as it was experienced for FPGA [60, 11]. One can then use the local memory to temporarily store the working data of the tile. This trades an amount of memory that potentially scales with the tile size, but greatly improves performance. On architectures with caches, this is automatically done.

Our inter-tile reuse strategy usually requires more local memory, as we keep data for future tiles or from past tiles. This again can save communications, but it requires to store a larger amount of data in the local memory. In our scheme, some tiles may keep data that they do not even use but that a previous tile requested and that a later tile will need. The effects of intra-tile reuse (classic in most compilers) and of inter-tile reuse (studied in Chapter 2) are depicted from left to right in Figure 5.1 with more temporal reuse and a larger footprint.

Finally, inter-tile reuse enables the pipelining of tiles even in the general case of flow dependences between successive tiles (see Figure 5.2 for a picture), which allows the execution to overlap communication with computation, thus potentially doubling the performance (if they were balanced). The software pipeline of tiles we consider (see Section 5.2 for more explanations) corresponds to a kind of double-buffering technique, which can thus double the amount local memory needed (but potentially less).

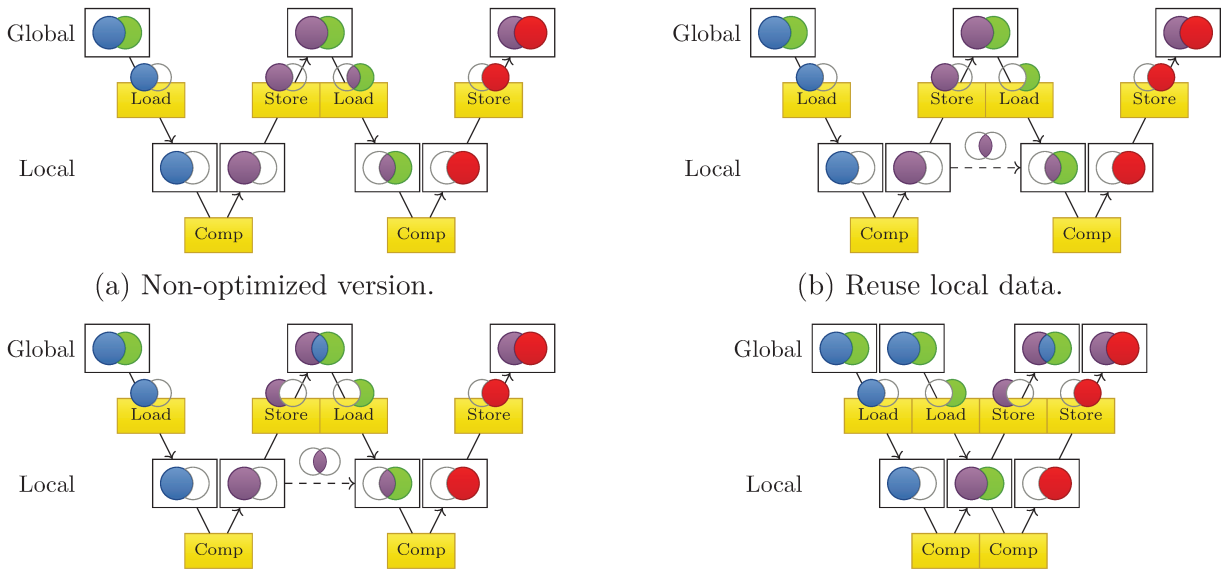
In summary, changing the tile sizes allows the compiler to explore different trade-offs between memory footprint and improved communications (at all levels of the memory hierarchy). Our parametric approaches (for inter-tile reuse, liveness analysis, memory mapping) give a conceptual way to explore the effect of the tile sizes on the communications and therefore to guide the search. This is left for future work however.

5.2 Pipelining and Double Buffering

One of the motivations to perform inter-tile data reuse was also to be able to pipeline communications and computations, and to automate double buffering, an optimization that some designers do by hand, but in general with pointer swapping (and not sliding windows as obtained with modulo allocations, which can handle more general situations).

The reason why inter-tile data reuse enables pipelining is illustrated in Figure 5.2. In the initial non-optimized version, the first set of data (blue disk) is loaded, modified (becomes violet), then stored back to global memory. Then, the second set of data (green disk with a violet slice) is loaded, then modified (becomes red), and stored back. In Figure 5.2b, data is kept in local memory for future reuse, but the global memory is still kept up-to-date. In Figure 5.2c, the part that is going to be modified a second time is not stored back immediately, it is stored only at the end, when it turned red. Thanks to this, pipelining is now possible as depicted in Figure 5.2d. There is also no need to bother with the (possibly unpredictable) time it can take to store back in global memory. There is now a complete decoupling between loads and stores, which leads to the task graph of Figure 2.4. All computations are done with local data, with a global memory that is not fully up-to-date, except at the start and end of the process.

There are now many possibilities to schedule (pipeline) this task graph, using a form of software pipelining, so as to hide communication latency with a double or even a triple buffering technique. Indeed, depending on the hardware synchronization capabilities



(a) Non-optimized version.

(b) Reuse local data.

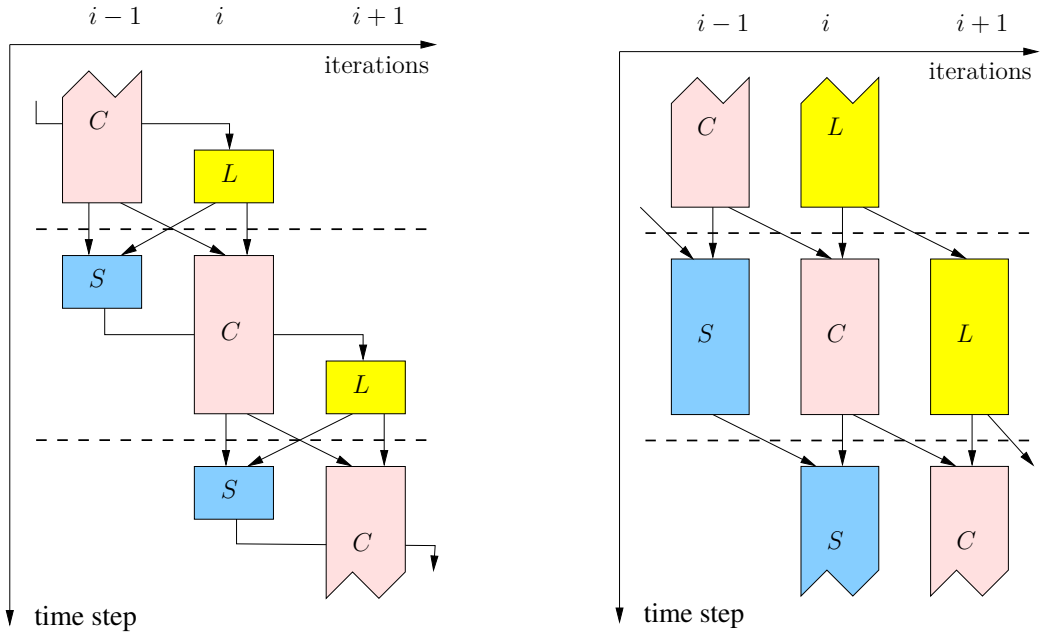
(c) Store back later.

(d) Pipelined version.

Figure 5.2 – An illustration of pipelining enabled by inter-tile reuse.

and the number of communication channels, the actual pipeline may vary. Figure 5.3a depicts a double buffering pipeline with half-duplex communication channel. The load and store communications are sequentialized on purpose so that they do not interfere, usually because they share the same channel and because interleaved accesses on different rows of a DDR SDRAM can increase latency. Figure 5.3b depicts a possible triple buffering pipeline with a full-duplex communication channel. Loads and stores can happen at the same time. Both share the same idea, which is to store the data produced by the previous tile and to load the data needed by the next tile at the same time as computing the current tile. This allows the execution to overlap communication and computation, thus hiding the latency of the communication, given sufficient time is spent computing.

The reason why we designed the pipeline of Figure 5.3a instead of the pipeline of Figure 3.2a (see Chapter 3), which was used for kernel offloading for FPGA in [60, 5], is twofold: it has more potential for overlapping and it is more regular so that the corresponding schedule used to compute liveness and interferences of array elements can be easily expressed. The pipeline of Figure 3.2a, as explained in [5], needed to distinguish odd and even tiles, in a symbolic unrolling by 2, which also induced problems for code generation as loops do not always start at even tiles. Anyway, now with the techniques developed in Chapter 3, there is no need to describe a schedule, conflicts can be computed by expressing the task graph dependences (arrows in Figure 5.3). This will be elaborated



(a) Half duplex: load/store are sequential.

(b) Full duplex: load/store are parallel.

Figure 5.3 – Double and triple buffering pipelines: L are loads, C computes, and S stores.

in Section 5.3. When such a pipeline is used to transfer data between an external memory and the global memory of a GPU, it just needs to be expressed, with events, through the use of the GPU streams. On FPGA, it can be expressed with explicit synchronizations through FIFOs [3, 5]. However, if barriers are the only mean of synchronization, then a lock-step schedule (as in Figures 3.2a and 5.3b) indeed needs to be defined.

Pipelining comes at a cost however: as the name of the technique suggests, double buffering might require twice the amount of local memory, and triple buffering thrice. It is to be noted that for a fixed local memory size, using triple buffering to fully utilize the bi-directional communication might not always give better performance than double buffering as it might require smaller tile sizes and thus increase the amount of communication. Actually, it is possible to get a double-buffering type of execution with full duplex, one just has to make sure that the stores at (tile) iteration T are enforced before the loads at iteration $T + 2$ (in addition to the dependences of the task graph of Figure 2.4), i.e., compared to the pipeline of Figure 5.3a, one just have to remove (i.e., there is no need to enforce) the dependence from loads at iteration T to stores at iteration $T - 1$. It can then be checked that the conflicts described in Section 5.3 do not change despite the increased parallelism. This is because the only possible effect is to store data earlier compared to loads, which tends to reduce live-ranges. Note also that, in many kernels, the arrays that are read and the arrays that are produced are different, in which case to get double buffering, there is no need to enforce sequentiality between stores at iteration T and loads at iteration $T + 2$. It is sufficient that computes (resp. stores) at iteration T are done before loads (resp. computes) at iteration $T + 2$.

5.3 Deriving Memory Conflicts

As already mentioned, an important criteria and even constraint for choosing tile sizes is the local memory usage. On architectures with a hardware cache, it is not needed and usually not possible to manually allocate the local memory or to even know how much is used by an application. Still, if the effective amount of memory needed to execute a tile exceeds the cache size, memory accesses will trigger cache misses and the corresponding memory level may be badly utilized. On architectures without hardware caching, that is when the memory level is a scratchpad memory, the local memory has to be manually allocated and a mapping strategy has to be decided. It this case, it is required, for the local memory, to fit in the corresponding physical memory.

In both cases, either simply to analyze the amount of data movement or to derive an actual mapping of the data to a scratchpad, one first needs to know when data become live

(i.e., are created) and when they become dead (i.e., are no longer used) for any possible execution (this is liveness analysis) and when memory location can be shared/reused safely (this is conflict analysis). These problems were addressed in Chapter 3.

The goal of this section is to illustrate how to apply the techniques presented in Section 3.3, for partial orders, thanks to an `iscc` script, to our kernel offloading with inter-tile data reuse, and to discuss some complexity issues that can arise. We will use again the `jacobi_1d_imper` example of Chapter 2, with reuse over the innermost loop only (i.e., a 1D tile band), assuming the double-buffering pipeline of Figure 5.3a. We will need the expressions of `Load`, `Store`, `Read`, and `Write` from the script of Figure 2.6.

Our objective is to compute the conflicts between all live-ranges, including conflicts within the computations of tiles (`Write` to `Read`), between the loads and the computations (`Load` to `Read`), and between the stores and the computations (`Write` to `Store`). Following the theory developed in Chapter 3 (see in particular Equation (3.7) and Figure 3.5b), we want to find writes that may happen during the live-range of another variable:

$$R_x \not\prec W_x,\ W_y \not\prec W_x,\ R_x \not\prec W_y.$$

To do this computation, we would need to define the happens-before relation $\prec$ for our pipeline schedule (a partial order), and then take its complement ($\not\prec$). Instead, we can directly express the pipeline into the complementary relation. In other words, we describe our pipeline directly through a may-happen-before relation (i.e., $\not\succ$). From the point of view of the local memory, a write occurs either as a load, or as a write.

To compute the conflicts due to a load, we need to characterize the set of live-ranges x that are possibly live at the execution of the load of y. To do so, we describe the elements from the relations `Load` and `Write` that may happen before this specific load ($W_y \not\prec W_x$), and the elements from the sets `Read` and `Store` that may happen after ($R_x \not\prec W_y$). The inequality $R_x \not\prec W_x$ is always satisfied in our case.

These constraints depend on the chosen pipeline. For the pipeline of Figure 5.3a, we get the following relations where the specific load W_y is always on the left hand-side of the relation and Before (resp. After) means may happen before (resp. after):

```
LoadBeforeLoad  := [st,si] -> { [T,I] -> [T,I'] : I' <= I };
WriteBeforeLoad := [st,si] -> { [T,I] -> [T,I',t',i',k'] : I' <= I-si };
ReadAfterLoad   := [st,si] -> { [T,I] -> [T,I',t',i',k'] : I' >= I-si };
StoreAfterLoad  := [st,si] -> { [T,I] -> [T,I'] : I' >= I-si };
```

For `LoadBeforeLoad`, the current load set induces conflicts with itself as all values are transferred before the tile starts to compute and possibly to consume the data. For the

relation `WriteBeforeLoad`, this is as expected as there is a dependence with the compute of the current tile, i.e., $L(I)$ happens before $C(I)$, thus the last compute tile that may happen before is $C(I - s_i)$. `ReadAfterLoad` includes the previous compute as it may be executed after the current load. Finally, `StoreAfterLoad` includes the previous store as the store one step before is the last one scheduled before the current load.

Notice that, due to our parametric tiling, the previous relations consider all the unaligned tiles too (an expression such as `I' <= I-si` captures all tiles, not just tiles aligned with a lattice, which would bring a quadratic constraint back). This is still correct, we might add spurious conflicts if the functions `Load` and `Store` are not point-wise (see Chapter 3 for more details), but this is not a problem in practice (and it is always legal).

We then compute `AliveLoad`, the set of array elements that are live during the loads of a given tile, i.e., the set of array elements that may be both written before (in the form of either a load or a write) and read after (as a read or a store):

```
AliveLoad := ((LoadBeforeLoad.Load) + (WriteBeforeLoad.Write))
           * ((ReadAfterLoad.Read) + (StoreAfterLoad.Store));
AlignCoalesced := { [T,I] -> [[T,I]->[T,I]] };
ConflictLoad := AlignCoalesced.(Load cross AliveLoad);
```

The purpose of `AlignCoalesced` is to equate the left "sides" created by the cross product into a single common load, so as to obtain a relation that associates to a load (the W_y we considered) the set of pairs of array elements that are in conflict (obtained, for each `[T,I]`, as the cross product of its images by `Load` and its images by `AliveLoad`).

We proceed the same way for the conflicts due to a write. The main difference is that writes are scheduled at a precise time inside a tile. This is important for the expression of `WriteBeforeWrite` and `ReadAfterWrite`:

```
LoadBeforeWrite  := [st,si] -> { [T,I,t,i,k] -> [T,I'] : I' <= I+si };
WriteBeforeWrite := PrevOrEq +
    [st,si] -> { [T,I,t,i,k] -> [T,I',t',i',k'] : I' <= I-si }
ReadAfterWrite   := Next +
    [st,si] -> { [T,I,t,i,k] -> [T,I',i',j',k'] : I' >= I+si };
StoreAfterWrite  := [st,si] -> { [T,I,t,i,k] -> [T,I'] : I' >= I-si };
```

The expression of `PrevOrEq` and `Next` represents the execution order inside a tile (which we chose to be the lexicographic order, for sequential execution):

```
PrevOrEq := { [T,I,t,i,k] -> [T,I,t',i',k'] : t',i',k' <<= t,i,k };
Next     := { [T,I,t,i,k] -> [T,I,t',i',k'] : t',i',k' >>  t,i,k };
```

Then, in the same way, we compute `AliveWrite`:

```
AliveWrite := ((LoadBeforeWrite.Load) + (WriteBeforeWrite.Write))
            * ((ReadAfterWrite.Read) + (StoreAfterWrite.Store));
Align := { [T,I,t,i,k] -> [[T,I,t,i,k] -> [T,I,t,i,k]] };
ConflictWrite := Align.(Write cross AliveWrite);
```

where `Align` serves the same purpose as `AlignCoalesced` before.

To find a good lattice-based memory mapping, using the technique of Chapter 4 or a simpler successive-modulo approach, we then need to compute the set of differences between conflicting pairs. This is done using the following `Delta` relation, which associates each pair to the difference, and `Symm` which makes the result symmetric with respect to $\vec{0}$:

```
Delta := { [A[m] -> A[m']] -> A[m'-m] } + { [B[m] -> B[m']] -> B[m'-m] };
Symm := { A[e] -> A[e]; A[e] -> A[-e] } + { B[e] -> B[e]; B[e] -> B[-e] };
```

which we apply finally as follows:

```
DeltaLoad  := range (ConflictLoad.Delta.Symm);
DeltaWrite := range (ConflictWrite.Delta.Symm);
Deltas := DeltaLoad + DeltaWrite;
```

While the final expression for `Deltas` (the conflicting differences) is usually relatively simple, the full expression for the memory conflicts can be quite complex due to the many special cases that frequently arise, mostly for very small (and not necessarily useful in practice) tile sizes. These are cases that can easily be eliminated by constraining tile sizes to be larger than a small constant (3 or 4 works well). The reason is that when tiles are really thin and the schedule is skewed by a coefficient of 2 or more, there are directions in which, when projected, some points are missing. Figure 5.4 shows an example where a skewing with a coefficient of 3, for tiles of "thickness" 2, creates holes in the projection. This means that in this case, the expression of the projection will have integer divisions, leaving the door open to an explosion of special cases on the boundaries.

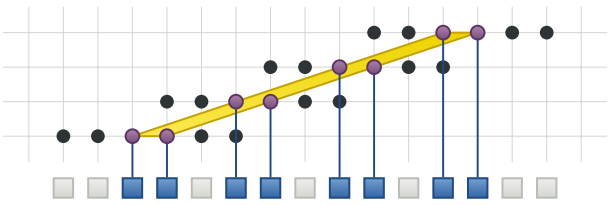


Figure 5.4 – Projecting a skewed thin tile may lead to a complicated formula.

The example of `jacobi_1d_imper` is one of the worst in this regard (`jacobi_2d_imper` does not have the same problem), due to a skew by 2, and leads to a combinatorial explosion of special cases when we try to solve it for all possible tile sizes simultaneously, i.e., in a parametric fashion without constraints on tile sizes. Limiting the tile sizes to be large enough makes the computation practicable, but the final expression still contains hundreds of cases. To avoid such situations, using a simple Fourier-Motzkin elimination from time to time, instead of an exact quantifier elimination over integers dramatically improves the situation. This limits the number of cases that arise when eliminating integer variables (such as for the projection of Figure 5.4), thus improving the efficiency of the analysis at the cost of precision (actually, as we then perform convex optimizations, such as maximization, for computing an allocation, it is not a problem in practice to work with a union of rational polyhedra instead of a union of integer points within polyhedra). In practice, we insert such rational eliminations when costly reductions happen and over-approximation is legal, such as for computing `ConflictLoad`, `ConflictCompute`, `DeltaLoad`, and `DeltaCompute`.

The expression of `Deltas` can then be dramatically improved by coalescing the different sets with the coalescing heuristic of `isl` (a technique to merge two polyhedra when their union can be expressed as a single one). Coalescing the expressions of `In`, `Out`, `Load`, and `Store` did also improve dramatically the time of the computation. It was however a bad idea on most examples to coalesce the conflict relations (`ConflictLoad` and `ConflictCompute`) directly as they are in a high dimensional form (computed from a cross product) and have many disjoint sets that cannot be merged easily. This resulted in a coalescing that was extremely expensive but found very few coalescing opportunities to improve the representation, thus it was inefficient at this stage and we do not recommend to apply it for these expressions.

On most PolyBench [63] examples (except `jacobi_1d_imper`), the analysis behaves without too much hassle. Coalescing in the steps specified previously was always beneficial for both the execution time and the resulting expression. The `jacobi_1d_imper` kernel however is really astonishing. It takes around a second to compute without any optimization and gives a result of 63KB, that once coalesced is reduced to 24KB. Assuming a tile size of at least 2 along the dimension i (innermost dimension) reduces the result further to, still, 5KB. If, instead of this assumption, we do the reduction using Fourier-Motzkin (by changing the polyhedra to rational sets, or by executing `isl_set_remove_divs`, but there is no easy access to such functions inside `iscc`) at the steps previously proposed, we get an output of 2.6KB. Finally, combining all these optimizations, we obtain a result of around 600B, in less than a second, suitable for further analysis and even readable by a

determined human. Surprisingly, the case of tiles of size 1 in the innermost computation dimension can then be analyzed separately and the results combined. The final expression is the following where `st`, `si`, `tsteps`, and `n` are the parameters, i.e., tile size in time dimension, tile size in computation inner dimension, number of iterations for time loop, and number of iterations for the computation loop.

```
[st,si] -> { st >= 1 and si > 1 } * [st, si, tsteps, n] -> {
  A[d] : tsteps > 0 and n >= 3 and d > -n and d > -2si - 2tsteps and
    -2st - 2si < d < n and d < 2si + 2tsteps and d < 2st + 2si;
  B[d] : tsteps > 0 and 3 - n <= d <= -3 + n and ((st >= 2 and tsteps >= 2
    and 3 - si <= d <= -3 + si + 2tsteps and d <= -3 + 2st + si) or
    (st >= 2 and tsteps >= 2 and d >= 3 - si - 2tsteps and
    3 - 2st - si <= d <= -3 + si) or (d >= 3 - 2tsteps and
    3 - 2st <= d <= -3 + 2si + 2tsteps and d <= -3 + 2st + 2si) or
    (d >= 3 - 2si - 2tsteps and 3 - 2st - 2si <= d <= -3 + 2tsteps and
    d <= -3 + 2st)); B[d = 0] : tsteps > 0 and n >= 3 } +
[st,si] -> { st >= 1 and si = 1 } * [st, si, tsteps, n] -> {
  A[d] : tsteps > 0 and n >= 3 and d > -n and d >= -1 - 2tsteps and
    -1 - 2st <= d < n and d <= 1 + 2tsteps and d <= 1 + 2st;
  B[d] : d >= 3 - n and d > -2tsteps and -2st < d <= -3 + n and d < 2tsteps
    and d < 2st and ((d >= 3 - 2tsteps and d >= 3 - 2st) or
    (d <= -3 + 2tsteps and d <= -3 + 2st));
  B[d = 0] : tsteps > 0 and n >= 3 };
```

We will detail the polynomial product kernel in the next section, providing as an illustration the load sets and the final mapping, as it is an example whose parametric solution is easy to interpret. The conflict sets themselves are easy to compute and do not reveal anything worth discussing, this is why we do not report them here.

5.4 Size and Mapping of Local Memory

In the case of an architecture with a hardware caching mechanism, there is no need to compute an actual mapping. However, it is a good indication of the amount of cache required for the execution of a pipelined tile. Indeed, if the analysis of Chapter 3 could, in theory, be used to derive lower bounds on the required memory (by computing cliques in the interference graph), computing a mapping actually provides an upper bound, which in practice can be close to the optimum. In the case of a scratchpad however, the alloca-

tion and memory transfers have to be manually setup, so an actual mapping is *de facto* necessary. In either case, this is where Chapter 4 comes into play. It provides a mapping for each local array based on modular arithmetic, whose memory usage is given by the product of the moduli. If the scratchpad is allocated in pages, it might be necessary to round up the final size to the closest multiple of the page size to get the exact physical size.

The parametric nature of our analysis produces an expression of the total amount of local memory required for the execution of a given tile. This expression should help devise performance models of the tiled code (in the case of a cache) and check the validity of the produced code (in the case of a scratchpad), i.e., make sure the data will indeed fit in local memory.

Although Chapter 4 provides both a good basis (direction along which to fold the array) and its associated moduli, it is possible to refine these moduli with a final recomputation via the standard successive modulo technique. Indeed, all the heuristics proposed in Chapter 4 only produce moduli that are affine in the parameters of the problem (this is due to the use of Farkas lemma). By construction, these moduli are valid for any values of the parameters, but it is always more precise to provide an affine function that depends on the values of these parameters in a piece-wise manner. For example, even in 1D, on the following conflicting differences:

```
[n] -> { [i] : -n < i < n or -2 < i < 2}
```

we would expect a memory of size n, but these heuristics actually produce $n + 2$, as the first one is in fact invalid for $n < 2$. The smallest modulo valid for all $n \geq 0$ and for which the expression is affine in n is indeed $n + 2$. Doing a final successive modulo computation produces piece-wise expressions for the moduli, and is possible as the basis is now fixed. In the previous case, we would get n for $n > 2$ and 2 otherwise, i.e., $\max(n, 2)$ as expected.

We now illustrate the kind of parametric mappings that we can get, for the kernel offloading strategy analyzed in Chapter 2, with a pipelining scheme as recalled in Section 5.2, following the conflict/liveness analysis of Chapter 3, as illustrated in Section 5.3. Again, one of the interests of computing the Load/Store sets in a parametric fashion is that, now, after all these steps, the size of the resulting local memory (e.g., obtained by bounding boxes, successive modulo, or the technique of Chapter 4) can also be computed in a parametric manner. Such a parametric scheme seems almost mandatory in a context such as described in [5, 62], for HLS from C to FPGA. Indeed, as explained in [5], some manual (though systematic) changes must be done to the tiled code so that it is accepted by the HLS tool. Doing these changes for all interesting tile sizes is not reasonable. Also, as explained in [62], identifying the right tile sizes may require executions

of multiple scenarios. Parametric code generation would help speeding up such a design space exploration. With this parametric inter-tile reuse, combined with parametric code generation [65], one should now be able to derive a fully automatic scheme, with parametric tile sizes. This also makes the design and use of analytic cost models possible, in particular to explore hierarchical tiling (whose search space is huge) and to predict its impact on local memory size.

We only report here some examples of the mappings we can obtain, for two schedules, as an illustration. The first schedule performs all computations in sequence (without pipelining): tiles are serialized and each tile performs its loads, then its computations, then its stores before a new tile is computed (in other words, it performs the tasks of the dependence graph of Figure 2.4) by successive columns, i.e., iterations). The second one is with the double-buffering pipeline (in each tile strip) depicted in Figure 5.3a, i.e., a schedule which, in addition to the precedences of Figure 2.4, serializes the transfers as $\text{Load}(\vec{I}_2) \to \text{Store}(\vec{I}_1) \to \text{Load}(\vec{I}_3) \to \text{Store}(\vec{I}_2) \to \ldots$, where $\vec{I}_1$, $\vec{I}_2$, $\vec{I}_3$ are three successive tiles for $\sqsubseteq_{\vec{s}}$. All other overlappings (in particular parallelism between computations and transfers) can arise at runtime, achieving a kind of double-buffering style of execution.

The `jacobi_1d_imper` code of Figure 2.1 has two parameters N and M defining the loop bounds (in the previous section, these bounds were denoted tsteps and n to recall PolyBench; here we use N and M for conciseness). The proposed tiling has also two tile size parameters s_1 and s_2. In principle, there could be a 5th parameter to specify each tile strip, but we chose to derive mappings valid for all tile strips (as for all examples hereafter). After Load/Store analysis and memory folding with successive moduli, we get (after simplification) the following sizes for `A` and `B`, for the sequential schedule:

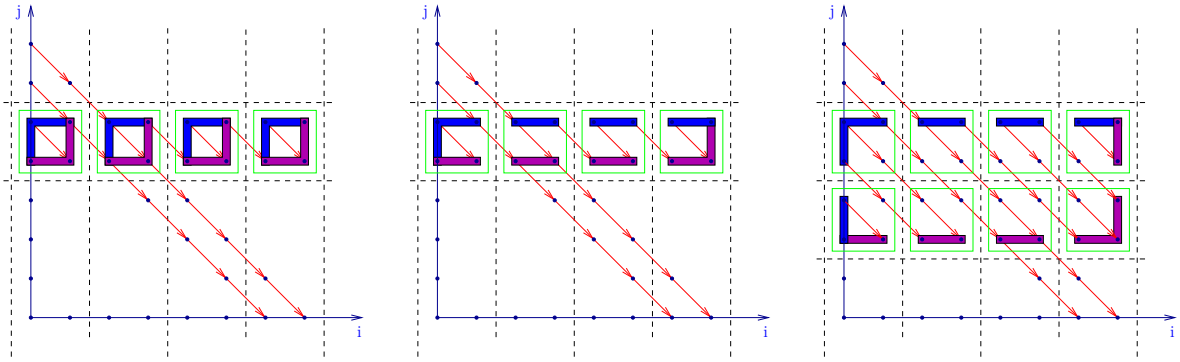
$$\text{size}(\texttt{B}) = \min(\max(0, N-2), 2M + s_2 - 1, 2s_1 + s_2 - 1)$$
$$\text{size}(\texttt{A}) = \min(N, 2M + s_2, 2s_1 + s_2)$$

and with the pipeline schedule:

$$\text{size}(\texttt{B}) = \min(\max(0, N-2), 2M + 2s_2 - 2, 2s_1 + 2s_2 - 2)$$
$$\text{size}(\texttt{A}) = \min(N, 2M + 2s_2, 2s_1 + 2s_2)$$

These expressions are actually expressed as disjunctions, each term that contributes to the minimum being specified by conditions on parameters.

One can also of course easily retrieve (this time in a parametric fashion) the expression of the memory size for the product of two polynomials analyzed in [5]. Let us first illustrate with this example, as this is easy to visualize, how by choosing a different range over which we do data reuse, we can further reduce the amount of communication. Figure 5.5 shows the different communications (for array C, see the code Page 23) depending on the reuse.



(a) No inter-tile data reuse (pipelining forbidden).

(b) Reuse along a single tile strip (requires a barrier between strips).

(c) Reuse over two consecutive tile strips (requires barrier between pairs of strips).

Figure 5.5 – Load and Store sets, depending on the range of the reuse analysis.

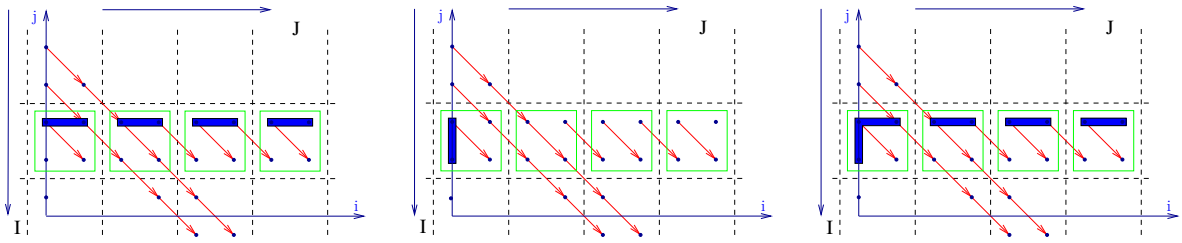
The blue pieces correspond to loads, the violet pieces to stores. While Figure 5.5c seems to be better, there is an inherent cost to such reuse, which is that the data loaded and redefined in the first tile of the first strip has to be kept until reused by the second tile of the second strip (the same happens for array A whose reuse is vertical in the figure). In other words, the local memory needed to execute the code will end up being more or less the memory needed to execute the full domain. We thus chose to only reuse data along a tile strip as it seemed to be a good trade-off. If there are enough tiles in a strip, the necessary barriers between tile strips are sufficiently amortized, while the reuse along only a strip is usually of the same order as the amount of memory needed by a single tile.

By looking in more details at the expressions we obtain, we can see the advantages of a parametric analysis. Figure 5.6 shows the different Load sets for inter-tile reuse along one strip, and below are the associated expressions (b is the tile size):

$$\begin{aligned}
\text{Load}_A(I,J) &= \{m \mid 0 \leq m \leq n-1,\ J \leq m \leq J+b-1\} \\
\text{Load}_B(I,J) &= \{m \mid J=0,\ 0 \leq m \leq n-1,\ n-I-b \leq m \leq n-I-1\} \\
\text{Load}_C(I,J) &= \{m \mid 0 \leq m,\ n-I-b \leq m \leq n-1-I,\ J=0\} \\
&\quad \cup \{m \mid \max(1,J) \leq m+I-n+1 \leq \min(n-1, J+b-1)\}
\end{aligned}$$

These expressions show that a portion of (at most) b elements of A is loaded for each tile (with no inter-tile reuse), that a portion of (at most) b elements of B is loaded for the first tile only (it will be reused for all tiles), and that the loads of C are different for the first tile of the tile strip (roughly $2b$ elements, then only b to complete with new elements).

Now, if we execute our conflict analysis then mapping optimization to get the local size of each array in the case of a double-buffering execution, we get the expected results



(a) Reuse for A (vertical). (b) Reuse for B (horizontal). (c) Reuse for C (diagonal).

Figure 5.6 – Load sets for the different arrays, for inter-tile reuse over a 1D tile strip.

with all the specific cases handled nicely by the polyhedral machinery. Array A and B require, respectively, a local array of size $\min(2b, n)$ (i.e., two blocks of size b, due to the double buffering pipeline) and $\min(b, n)$ (i.e., only one block of size b, kept for the whole tile strip), while C requires a local array of size $3b - 1$ when $n \geq 2b + 1$, of size $b + n - 1$ when $b \leq n \leq 2b$, and of size $2n - 1$ when $n \leq b - 1$. These sizes correspond respectively to the cases where there are at least two full tiles, one full tile and one partial tile, or only one partial tile. The fact that A occupies twice more space than B suggests that it might be interesting to explore rectangular tiles (which our analysis has no trouble with; here we studied square tiles, with a single size parameter b, only for conciseness).

We are still working on an automated implementation of our algorithms with `isl`, to be integrated into the optimizer for GPUs PPCG [75] (see Section 5.5). For the moment, we manually adapted an `iscc` script as presented earlier, for some `PolyBench` [63] examples. The loop transformations to enable tiling were computed by the `isl` scheduler, which gives results similar to those of Pluto [61]. We tiled the largest consecutive tilable dimensions (underlined in Table 5.1) for which dependences are nonnegative. Some examples were omitted, either because the `isl` scheduler did not exhibit any "tileability"[1] – at least without preliminary transformations such as array expansion –, or because they had too many instructions[2] or variables[3] and would not fit in the table (these examples were not tried: they may – but maybe not – reveal complexity issues, which will have to be explored with an automatic implementation in `isl`, as well as different approximation schemes). Moreover, parameters were restricted so that each kernel domain contains at least one strip with at least two consecutive full tiles, and tile sizes are at least 2: this avoids many special cases (their generation is possible however) that, again, would not fit in the table.

The results we provided in Table 5.1 are the array sizes after memory folding. We computed a memory allocation compatible for all tile strips, depending on the program

[1] Kernels `durbin`, `ludcmp`, `cholesky`, and `symm`
[2] Kernels `adi`, `fdtd-apml`, `gramschmidt`, `2mm`, `3mm`, `correlation`, and `covariance`
[3] Kernels `bicg`, `gemver`, and `gesummv`

Table 5.1 – Local memory sizes for PolyBench 3.2.

Sample	Schedule	Sequential Memory Size	Pipelined Memory Size
Stencils			
fdtd-2d	$S_0(t,j)\mapsto(\underline{t},t,t+j,0)$ $S_1(t,i,j)\mapsto(\underline{t},t+i,t+i+j,1)$ $S_2(t,i,j)\mapsto(\underline{t},t+i,t+i+j,3)$ $S_3(t,i,j)\mapsto(\underline{t},t+i+1,t+i+j+1,2)$	$\texttt{hz}[s_1+s_2,\min(s_1,s_2)+s_3]$ $\texttt{ex}[s_1+s_2,\min(s_1,s_2)+s_3]$ $\texttt{ey}[s_1+s_2,\min(s_1,s_2-1)+s_3]$ $\texttt{\_fict\_}[\min(s_1,s_2)]$	$\texttt{hz}[s_1+s_2,\min(s_1,s_2)+2s_3]$ $\texttt{ex}[s_1+s_2,\min(s_1,s_2)+2s_3]$ $\texttt{ey}[s_1+s_2,\min(s_1,s_2)+2s_3]$ $\texttt{\_fict\_}[\min(s_1,s_2)]$
jacobi-1d-imper	$S_0(t,i)\mapsto(\underline{t},2t+i,0)$ $S_1(t,j)\mapsto(\underline{t},2t+j+1,1)$	$\texttt{A}[2s_1+s_2]$ $\texttt{B}[2s_1+s_2-1]$	$\texttt{A}[2s_1+2s_2]$ $\texttt{B}[2s_1+2s_2-2]$
jacobi-2d-imper	$S_0(t,i,j)\mapsto(\underline{t},2t+i,2t+i+j,0)$ $S_1(t,i,j)\mapsto(\underline{t},2t+i+1,2t+i+j+1,1)$	$\texttt{A}[2s_1+s_2,\min(2s_1,s_2+1)+s_3]$ $\texttt{B}[2s_1+s_2-1,\min(2s_1,s_2)+s_3-1]$	$\texttt{A}[2s_1+s_2,\min(2s_1,s_2+1)+2s_3]$ $\texttt{B}[2s_1+s_2-1,\min(2s_1,s_2+1)+2s_3-2]$
seidel-2d	$S_0(t,i,j)\mapsto(\underline{t},t+i,2t+i+j)$	$\texttt{A}\begin{bmatrix}s_1+s_2+1,\\ \min(2s_1+2,s_1+s_2,2s_2+2)+s_3\end{bmatrix}$	$\texttt{A}\begin{bmatrix}s_1+s_2+1,\\ \min(2s_1+2,s_1+s_2,2s_2+2)+2s_3\end{bmatrix}$
Medley			
floyd-warshall	$S_0(k,i,j)\mapsto(k,\underline{i,j})$	$\texttt{path}\begin{bmatrix}\max(k+1,n-k),\\ \max(k+1,n-k)\end{bmatrix}$	$\texttt{path}\begin{bmatrix}\max(k+1,n-k),\\ \max(k+1,n-k,2s_2)\end{bmatrix}$
reg-detect	$S_0(t,j,i,cnt)\mapsto(\underline{t},j-i,t+i,t+cnt,2)$ $S_1(t,j,i)\mapsto(\underline{t},j-i,t+i,t,4)$ $S_2(t,j,i,cnt)\mapsto(\underline{t},j-i,t+i,t+cnt,3)$ $S_3(t,j,i)\mapsto(\underline{t},j-i,t+i,len+t,0)$ $S_4(t,i)\mapsto(\underline{t},-i,t+i,len+t,5)$ $S_5(t,j,i)\mapsto(\underline{t},j-i,t+i,len+t,1)$	$\texttt{diff}\begin{bmatrix}s_1+s_2+s_3-3,\\ \min(s_1+s_3-2,s_2),\\ \min(s_1,s_3)+s_4-1)\end{bmatrix}$ $\texttt{path}\begin{bmatrix}\min(s_1-1,s_4)+s_2+s_3-1,\\ \min(s_1+s_3-1,s_2,s_3+s_4)\end{bmatrix}$ $\texttt{mean}\begin{bmatrix}s_2+s_3-1,\\ \min(s_2,s_3-1)\end{bmatrix}$ $\texttt{sum\_tang}\begin{bmatrix}s_1+s_2+s_3-2,\\ \min(s_1+s_3-1,s_2)\end{bmatrix}$ $\texttt{sum\_diff}\begin{bmatrix}s_1+s_2+s_3-2,\\ \min(s_1+s_3-1,s_2),\\ \min(s_1,s_3)+s_4\end{bmatrix}$	$\texttt{diff}\begin{bmatrix}s_1+s_2+s_3-3,\\ \min(s_1+s_3-2,s_2),\\ \min(s_1,s_3)+s_4-1)\end{bmatrix}$ $\texttt{path}\begin{bmatrix}\min(s_1,2s_4)+s_2+s_3-1,\\ \min(s_1+s_3,s_2,s_3+2s_4)\end{bmatrix}$ $\texttt{mean}\begin{bmatrix}s_2+s_3-1,\\ \min(s_2,s_3-1)\end{bmatrix}$ $\texttt{sum\_tang}\begin{bmatrix}s_1+s_2+s_3-2,\\ \min(s_1+s_3-1,s_2)\end{bmatrix}$ $\texttt{sum\_diff}\begin{bmatrix}s_1+s_2+s_3-2,\\ \min(s_1+s_3-1,s_2),\\ \min(s_1,s_3)+s_4\end{bmatrix}$
Linear algebra solvers			
dynprog	$S_0(iter,i,j)\mapsto(iter,\underline{i},0,j,4)$ $S_1(iter,i,j)\mapsto(iter,\underline{i},0,\underline{j},3)$ $S_2(iter,i,j,k)\mapsto(iter,\underline{k,j,i+j},1)$ $S_3(iter,i,j)\mapsto(iter,\underline{j,j,i+j},2)$ $S_4(iter)\mapsto(iter,\underline{len,len,len},0)$	$\texttt{sum\_c}\begin{bmatrix}\min(s_1,s_2+s_3-1),\\ s_2+s_3-2,\\ st\end{bmatrix}$ $\texttt{W}\begin{bmatrix}\min(s_1,s_2)+s_3-1,\\ \min(s_1,s_2,s_3)\end{bmatrix}$ $\texttt{c}[len-1,len-2]$	$\texttt{sum\_c}\begin{bmatrix}\min(s_1,s_2+2s_3-1),\\ s_2+2s_3-3,\\ st\end{bmatrix}$ $\texttt{W}\begin{bmatrix}\min(s_1,s_2)+2s_3-1,\\ \min(s_1,s_2,2s_3)\end{bmatrix}$ $\texttt{c}[len-1,len-2]$
lu	$S_0(t,i)\mapsto(\underline{k,k,j},1)$ $S_1(t,i,j)\mapsto(\underline{k,i,j},0)$	$\texttt{A}[n,n]$	$\texttt{A}[n,n]$
Linear algebra kernels			
atax	$S_0(i)\mapsto(\underline{0,i},2)$ $S_1(i)\mapsto(\underline{i,0},0)$ $S_2(i,j)\mapsto(\underline{i,j},1)$ $S_3(i,j)\mapsto(\underline{i,ny+j},3)$	$\texttt{A}[s_1,ny]$ $\texttt{x}[s_2]$ $\texttt{y}[ny]$ $\texttt{tmp}[s_1]$	$\texttt{A}[s_1,ny]$ $\texttt{x}[2s_2]$ $\texttt{y}[ny]$ $\texttt{tmp}[s_1]$
doitgen	$S_0(r,q,p)\mapsto(\underline{r,q,p},0,0)$ $S_1(r,q,p,s)\mapsto(\underline{r,q,p+s},s,1)$ $S_2(r,q,p)\mapsto(\underline{r,q,p+np},np,2)$	$\texttt{A}[s_1,s_2,np]$ $\texttt{sum}[s_1,s_2,s_3+s_4-1]$ $\texttt{C4}[s_4,s_3]$	$\texttt{A}[s_1,s_2,np]$ $\texttt{sum}[s_1,s_2,s_3+2s_4-1]$ $\texttt{C4}[2s_4,s_3]$
gemm	$S_0(i,j)\mapsto(\underline{i,j},0,0)$ $S_1(i,j,k)\mapsto(\underline{i,j,k},1)$	$\texttt{A}[s_1,s_3]$ $\texttt{B}[s_3,s_2]$ $\texttt{C}[s_1,s_2]$	$\texttt{A}[s_1,2s_3]$ $\texttt{B}[2s_3,s_2]$ $\texttt{C}[s_1,s_2]$
mvt	$S_0(i,j)\mapsto(1,\underline{i,j})$ $S_1(i,j)\mapsto(0,\underline{i,j})$	for S_0 for S_1 $\texttt{A}[s_1,s_2]$ $\texttt{A}[s_2,s_1]$ $\texttt{x1}[s_1]$ $\texttt{x2}[s_1]$ $\texttt{y\_1}[s_2]$ $\texttt{y\_2}[s_2]$	for S_0 for S_1 $\texttt{A}[s_1,2s_2]$ $\texttt{A}[2s_2,s_1]$ $\texttt{x1}[s_1]$ $\texttt{x2}[s_1]$ $\texttt{y\_1}[2s_2]$ $\texttt{y\_2}[2s_2]$
syr2k	$S_0(i,j)\mapsto(\underline{i,j},0,0)$ $S_1(i,j,k)\mapsto(\underline{i,j,k},1)$ $S_2(i,j,k)\mapsto(\underline{i,j,k},2)$	$\texttt{A}[ni,s_3]$ $\texttt{B}[ni,s_3]$ $\texttt{C}[s_1,s_2]$	$\texttt{A}[ni,2s_3]$ $\texttt{B}[ni,2s_3]$ $\texttt{C}[s_1,s_2]$
syrk	$S_0(i,j)\mapsto(\underline{i,j},0,0)$ $S_1(i,j,k)\mapsto(\underline{i,j,k},1)$	$\texttt{A}[ni,s_3]$ $\texttt{C}[s_1,s_2]$	$\texttt{A}[ni,2s_3]$ $\texttt{C}[s_1,s_2]$
trisolv	$S_0(i)\mapsto(\underline{0,i},0)$ $S_1(i,j)\mapsto(\underline{j,i},1)$ $S_2(i)\mapsto(\underline{i,i},2)$	$\texttt{A}[s_2,s_1]$ $\texttt{x}[n]$ $\texttt{c}[s_2]$	$\texttt{A}[2s_2,s_1]$ $\texttt{x}[n]$ $\texttt{c}[2s_2]$
trmm	$S_0(i,j,k)\mapsto(i,\underline{j+k,j})$	$\texttt{A}[1,\min(k,s_1+s_2-1)]$ $\texttt{B}\begin{bmatrix}\max(ni-k,k+1),\\ \min(ni,s_1+k,s_2+k)\end{bmatrix}$	$\texttt{A}[1,\min(k,s_1+2s_2)]$ $\texttt{B}\begin{bmatrix}\max(ni-k,k+1),\\ \min(ni,s_1+k,2s_2+k)\end{bmatrix}$

parameters and the counters of the loops surrounding the tiled loops. Another choice could have been to compute a memory allocation depending on the strip, potentially saving space for boundary strips. The memory size was computed for both sequential and pipelined (double buffering) execution with inter-tile data reuse, using the successive modulo approach [54]. Future work will be needed to consider approximations, not provided in the table, as well as techniques to speed up and simplify both the expressions of intermediate or final (such as load) sets and the memory sizes.

Double buffering, as expected, usually doubles the local memory size in terms of the innermost tile size. Some arrays require almost all data to be live during a strip, thus causing the whole array to be stored into local memory (e.g., `x` in `trisolv`). Furthermore, modulo allocation has limitations. It is really apparent on `floyd_warshall` where memory conflicts are spread in such a way that only a modulo bigger than $k+1$ and $n-k$ on both dimensions is valid. Thus, while the number of conflicting memory addresses is proportional to the tile area, the allocation is not. A tighter memory allocation could be obtained with a piece-wise modulo allocation scheme, allocating accesses to `path`$[i,k]$ and `path`$[k,j]$ differently from the accesses to `path`$[i,j]$. More generally, it is more likely that automating such schemes, with pipelining, parallelism, and hierarchical transfers, will require more advanced communication and allocation strategies.

5.5 Targeting GPGPU

This section describes the design choices we made on the (unfinished) implementation of a code generator targeting GPU accelerators, and more specifically CUDA devices (but this should apply to OpenCL as well). We decided to build on top of PPCG [75], which already features all the important aspects of CUDA code generation, namely: a polyhedral SCoP extraction (through `pet`), a polyhedral scheduling algorithm (to improve tileability, with GPU considerations), tiling heuristics (for when to tile and what), and a CUDA code generator.

Our goal was to provide, thanks to our parametric tiling with inter-tile data reuse:

- a full parameterization of the tile sizes at grid, block, and thread tile levels;
- a robust reuse analysis for transfers between kernels (including successive execution of instances of the same kernel), and between tiles executed on the same block and instructions on the same thread;
- pipelining of communication between the host and the GPU, as well as between the global memory and the shared memory (which also saves barriers).

The full implementation of inter-tile data reuse (as exposed in Chapter 2) for the special case of GPUs required much more time than anticipated, the theory itself opened new problems (in particular due to the handling of unaligned tiles and of pipelining), and we instead redirected our effort on the design of a better memory-conflict analysis (as exposed in Chapter 3) and memory-mapping strategy (as exposed in Chapter 4). As there is no complete working implementation of the following design at the present time, we cannot guarantee significant performance improvement on the final result compared to PPCG. However, the proximity of our design should guarantee an equivalent (albeit parametric) code if neither inter-tile reuse nor pipelining of communications is done, while we expect these optimizations to provide a significant improvement. They also seem mandatory to be able to run kernels whose necessary data do not fit entirely in the global memory of the GPU and therefore require to be cut into sub-parts.

One of the main design choices is to decide how to use tiling, and possibly hierarchical tiling, to exploit the multiple levels of the memory hierarchy of a GPU architecture. Transfers will be made at the frontiers of tiles, but how this can be done depends on both how tiles are mapped to kernels, blocks, or threads, and what type of synchronization and parallelism is available or required at each level. We address these issues in the next sections, exposing some design considerations that we tried or envision for the future.

5.5.1 Choice of Permutable Loop Band

PPCG has a relatively good heuristic to expose and choose the permutable loop band (a set of loops that are fully permutable with each other) that will be exploited for parallelization on the GPU. As of today, the strategy is as follows: using a modified version of Pluto scheduling [1], find a permutable loop band with at least one parallel loop, and if this fails, use Feautrier's scheduling algorithm [38] (for innermost parallelism) to find a sequential loop that carries as many dependences as possible (which are therefore taken care of) and try again without them. If no parallelism is found at the end, then the kernel is not exported to the GPU. The reason for this choice is that, while Pluto scheduling algorithm is good at maximizing the size of the loop band (and thus tileability), finding parallel loops is an extreme case of its search for good locality and it might not always find some. On the other hand, Feautrier's (or greedy based) scheduling is good at finding parallelism but it produces outermost sequential loops that will end up on the CPU (this is not ideal but is better than nothing).

We kept this heuristic as is, as it was not the part we were focusing on. As a general remark, in the case of sequential loops created by Feautrier's scheduling, the reuse analysis

between kernel instances that we propose should significantly reduce the cost of keeping sequential loops on the CPU. On another direction, some codes might present multiple nested loop bands with parallelism, in which case it might be interesting to switch to inner loop bands when tiling for inner levels, in case the outer level already depleted the parallelism available on the first band. We did not explore any of these design choices.

5.5.2 Handling Sequentiality

As explained in Section 1.1.2 of Chapter 1, the GPU architectures provide a huge amount of parallelism but with rather limited synchronization capabilities. This is one of the reasons why their programming is relatively difficult and why not all kernels can be accelerated by them. More specifically, GPUs cannot synchronize some of the parallel threads at all levels. Threads on different SM (streaming multiprocessors) cannot efficiently interact (except through atomic operations) and should preferably show perfect parallelism, i.e., they should not require any synchronization with another thread of a different block of the same kernel (again, as recalled in Section 1.1, relying on such atomic operations to do synchronizations is risky as it can lead to deadlocks in general).

Because of this, outermost loops that are sequential (due to dependences) cannot be propagated inside a kernel (or the kernel would have to use a single block or atomic operations for "risky" synchronizations) and should thus be implemented as executing a sequence of kernel instances (that is, controlled by the CPU). However, sequential loops inside a band, nested in one or more parallel loops, are fine. Indeed, if tiling is performed on the full band and if only the tile dimensions (i.e., the tile loops) corresponding to the parallel loops of the band are distributed among CUDA blocks, then the sequential loops of the band can be propagated to the inner levels (i.e., inside a block) without the need for synchronization. They may then be tiled further down with the point loops that correspond to the parallel loops. In other words, sequentiality between tiles can be used for offloading between CPU and GPU, while it can be used only inside a block for offloading between global and local memory, or at register level. A block can then execute multiple successive tiles, thanks to the possible full synchronization of threads of the same block with the `__syncthreads()` call (as in the PPCG code given in Figure 5.7).

This is the current strategy adopted by PPCG and we did not change it. There might be some interest in exploiting wave-front parallelism (which requires to schedule rectangular tiles along a diagonal). But it is difficult to implement in the parametric case, even without inter-tile data reuse ([47, 8]), and the technique of Chapter 2 is not suitable for a diagonal sequence of tiles. In any case, the fact that it induces outer sequential

```
__global__ void kernel0(float *A, float *B, float *C, int n) /* n=12288 */
{
   /* Grid: 192*192 blocks, each with 32*32 threads */
    int b0 = blockIdx.y, b1 = blockIdx.x;
   /* Loops: 384*384*768 tiles, each with 32*32*16 points */
    int t0 = threadIdx.y, t1 = threadIdx.x;
   /* Thus 1 block = 2*2*768 tiles, 1 thread = 1*1*16 points */
    __shared__ float shared_A[32][16];
    __shared__ float shared_B[16][32];
    float private_C[1][1];

    /* 6144 = 32 (tile size) * 192 (number of blocks) */
    for (int g1 = 32 * b0; g1 <= 12256; g1 += 6144)
      for (int g3 = 32 * b1; g3 <= 12256; g3 += 6144) {
        private_C[0][0] = C[(t0 + g1) * 12288 + (t1 + g3)];
        /* 16 consecutive points along k in a thread */
        for (int g9 = 0; g9 <= 12272; g9 += 16) {
          if (t0 <= 15) /* 32*32 threads, only 16*32 do the transfer */
            shared_B[t0][t1] = B[(t0 + g9) * 12288 + (t1 + g3)];
          if (t1 <= 15) /* 32*32 threads, only 32*16 do the transfer */
            shared_A[t0][t1] = A[(t0 + g1) * 12288 + (t1 + g9)];
          __syncthreads();
          /* compute the 16 consecutive points along k */
          for (int c4 = 0; c4 <= 15; c4 += 1)
            private_C[0][0] += (shared_A[t0][c4] * shared_B[c4][t1]);
          __syncthreads();
        }
        C[(t0 + g1) * 12288 + (t1 + g3)] = private_C[0][0];
        __syncthreads();
      }
}
```

Figure 5.7 – PPCG-generated code (GPU part) for matrix product.

loops might make the transformation unsatisfactory. The situation is similar for diamond tiling [6], a particular situation of wave-front tiling, for the same reason (the "past" of such sets of parallel tiles cannot be exactly described with a piece-wise affine relation), while overlap and split tiling [42], with faces parallel to the axis, may be more suitable for our inter-tile reuse analysis. However, we did not dive into these cases so far.

5.5.3 Kernels and Host/Device Memory Transfers

In the CUDA world, kernels are executed on the GPU, but the request for the computation is done on the CPU. Before executing a kernel, it is required to send the necessary data on the GPU with an explicit host-to-device memory transfer, and once the execution is done, to copy back the results with an explicit device-to-host memory transfer. In other words, all the data needed for the kernel is first sent to the global memory of the GPU.

As the requests are done on the CPU side, there is no restriction on the complexity or weirdness of the code executing the requests. PPCG currently implements a simple sequential execution of the kernels, with synchronous memory transfers. It does not pipeline the communication between kernels, and does not split a set of nested loops into multiple kernels, unless there is one or more sequential outer dimensions (such as a sequential loop or imperfectly nested loops). For these external transfers, we suggest applying a supplementary outer level of tiling on each kernel so as to split it into smaller kernel instances and exploit a better overlap between communication and computation. Our inter-tile data reuse analysis should limit the communication overhead to a minimum (there is even no overhead if we look for reuse over the full domain). Decomposing a kernel into smaller pieces (tiles) is also needed when the full kernel requires data that do not fit into the memory of the GPU. In this case, data reuse (on the global memory) has to be limited so as to require less memory than the full data.

To implement our proposed pipeline, we have to use asynchronous memory transfers and CUDA streams. One can first think of two natural strategies, depending on whether we see the task graph of Figure 2.4 by columns or by rows. The first strategy by columns is to use a stream per tile and its associated communications, and then to use events to synchronize loads, computations, and stores of different tiles to take care of other dependences (including those expressing the pipeline of Figure 5.3a). To avoid using an unbounded number of streams (i.e., one per tile), one can actually fold them into only two streams, thanks to the particular structure of the pipeline. The second strategy by rows is to use three streams, one per category of operations (loads, computations, stores), and to use events to synchronize the loads, computations, and stores of a given tile, and the additional synchronizations of the pipeline.

Actually, both needs to be implemented with care and tried on the actual GPU, because a naive implementation revealed to be problematic in practice as the order in which the CPU code fills the different streams influences the effective parallelism that the streams will exhibit at runtime. This is first due to the fact that to be able to wait for an event, this event must have already been not only created but also recorded in a stream (this constraint is a good thing in terms of semantics, as this prevents deadlocks

to be written, but, from a code generation point of view, it forces to records tasks and events in topological order). But this is also due to some implicit serialization, in declared streams of course, but also in the I/O streams of the GPU, and on the way the tasks in user streams are mapped to the actual streams of the GPU. These issues may be different in versions of CUDA above 7.0, but at least for our GPUs, this is what we experienced.

There is another important consideration to take into account for host/device memory transfers, which is pinned memory. CUDA provides ways to allocate memory on the host that use a simple fixed paging strategy. The CPU code is responsible for recording computation and communication tasks into the streams, but the actual transfers are done by the GPU. This memory pinning allows the GPU to read/write from/to the host memory without requesting a page translation for every page. In practice, data that is in pinned memory is transferred twice as fast to and from the device than normal memory. PPCG does not transform the allocations of the host data (`malloc`) into allocations into pinned memory (`cudaMallocHost`), as these allocations are usually done outside the SCoPs. It might be interesting to look into that. So far, we did it by hand by modifying either the source code (before PPCG) or the code generated by PPCG.

Finally, let us mention that CUDA optionally provides unified memory, which avoids the need to specify the memory transfers, but it trades performance for ease of use. The CUDA documentation recommends to use explicit memory transfer over pinned memory when the objective is getting maximal performance. Consequently, we did not use nor recommend the unified memory capabilities in the context of high performance computing.

5.5.4 Tiling Hierarchy

In its original formulation [75], the strategy of PPCG is centered around one tiling (guided by the `tile-size` parameter, number of points in a tile along each dimension), whose tiles and points are distributed respectively to CUDA blocks and threads (guided by the `grid-size` and `block-size` parameters, which give the number of blocks and number of threads in a block along each dimension). The idea is that a tile fits on a SM, a block can execute several of such tiles, in sequence (when grid-size $\times$ tile-size $<$ domain-size), and a thread can execute several points in these tiles (when block-size $<$ tile-size). This could be viewed as two supplementary tiling levels, as we will show later.

The tiles executed by a block, and the points executed by a thread, are allocated in an interlaced (cyclic) manner. For example, if grid-size = 4, the block 0 executes the tiles numbered 0, 4, and 8, the block 1 executes the tiles 1, 5, and 9, etc. Similarly, if block-size = 3, the thread 0 executes the points 0, 3, and 6, the thread 1 executes the

points 1, 4, and 7, etc. This has some advantages, such as there is no need to predict the number of tiles that needs to be executed by a block, or the number of points that needs to be executed by a thread. It is automatically taken care of by the original loop bounds: while the interlacing is computed by using a loop stride equal to the size of the grid (in the case of blocks), or the size of the block (in the case of threads), the iterations simply stop when going out of the domain. See for example the `g1` and `g3` loops in Figure 5.7, which iterate on the multiple tiles assigned to the block identified by (`b0`, `b1`). As blocks are expected to execute in parallel, interlaced execution is relatively good for the L2 cache (as different blocks might share some accessed data thanks to spatial locality). It is also good for the threads, as they are executed in batches of 32 (warp) that run in lock-step, which means that these 32 threads will probably access 32 consecutive data, thus mostly requiring the same lines of the shared memory, at the same time. However, these positive effects seem minor or at least worth reconsidering, as discussed at the end of this section.

In PPCG, when sequential loops are part of the loop band being considered and tiled, they produce sequential tile loops at the block level, and sequential point loops at the thread level. Along these sequential loops, and also along the loops that iterate over the multiple parallel tiles possibly assigned to a given block, data reuse can reduce the amount of memory transfers. PPCG uses a simple but efficient approach for that, which is to hoist a memory transfer to an outer loop if it does not depend on the iterators of the current loop. In other words, if there is perfect obvious reuse of the data, exploit it. As an example, see in Figure 5.7 how the transfer of the elements of `C` is hoisted out of the `g9` loop, i.e., is now used for all tiles along the `k` dimension, for a given thread.

Our analysis should be able to improve on this restriction, i.e., exploit reuse even when there is only partial reuse (as in the case of the polynomial product example). The difficulty is that our reuse analysis (see Chapter 2) relies on the fact that the tiles executed before a given tile can be summarized as the points in a contiguous region of space that can be described with a finite union of polyhedra. This is not the case with an interlacing distribution (unlike for sequential loops where our technique directly applies). To solve this issue, we propose to use a non-interlaced execution of tiles, i.e., an assignment of successive tiles or points to blocks and threads (what is called block, or possibly block-cyclic, distribution, and not just cyclic). The problem is that now, for code generation, we need to know the number of successive tiles (resp. points) assigned to a given block (resp. thread), while, as previously explained, with the interlaced distribution of PPCG, this was computed implicitly through loop bounds and strides.

To make this possible, we instead switch to another, more direct, tiling hierarchy view, with 5 levels of tiling (most were already proposed by PPCG in some ways, but

expressed differently). Hierarchical tiling means that, for each level we now describe, a tile is actually viewed as an atomic grain of computations (and associated communications), which is itself tiled for the inner levels.

- Each level-5 tile is executed in a new kernel, tiles are executed in sequence, data are transferred from host to device (Load) and from device to host (Store) in parallel of the computation of the previous and/or next tile if reuse analysis is performed. This inter-tile reuse was not provided by PPCG.

- Each level-4 tile is executed in a new block, only parallel loops are tiled, the data required for this block does not necessary fit into the shared memory of a SM. This tiling corresponds to the implicit CUDA grid of blocks. The number of blocks is then the ratio between level-4 and level-5 tile sizes. If the level-5 size is not known (e.g., no level-5 tiling), we can still rely on a block-cyclic distribution (with fixed block size) and a number of tiles per block that depends on the iteration domain.

- Each level-3 tile is executed in its assigned block (the block of the level-4 tile to which it belongs), tiles are executed in sequence, the local buffer fits into the shared memory of the block, transfers from device memory to shared memory can be pipelined (saving barriers) if reuse analysis is performed.

- Each level-2 tile is executed in a new thread, only parallel loops are tiled, the data does not necessary fit into registers. This corresponds to the implicit block of CUDA threads. The number of such tiles (ratio between level-2 tile sizes and level-3 tile sizes) should fit into the limit of CUDA threads per block.

- Each level-1 tile is executed in its assigned thread (the thread of the level-2 tile to which it belongs), tiles are executed in sequence, local buffer fits into the registers (preferably), transfers from shared memory to registers are pipelined by the instruction pipeline. If reuse analysis and memory mapping are performed, allocating local arrays to registers might improve performance. This tile size should be a constant for parallel loops as they will be unrolled (see later why).

This is summarized in Table 5.2. There is a recurring scheme where we interlace fully parallel tiles and pipelined tiles. This ensures that sufficient parallelism is offered while, at the same time, the amount of communications is controlled and they can overlap with the computations (and communications of inner tiles). Notice that it might be interesting to use parallel loops over kernels (from the highest tile level) to distribute work over multiple GPUs. Such a scheme differs from PPCG in the sense that the successive tiles of a given

Table 5.2 – Proposed tiling hierarchy for CUDA code generation

Tile Level	CUDA	Parallel	Pipelined	Footprint	Details
5 (outer)	Kernel	No	Yes	Device memory	CUDA stream
4	Grid	Required	No		Implicit (grid size)
3	Block	No	Yes	Shared memory	Explicit
2	Thread	Required	No		Implicit (block size)
1 (inner)	Instr.	Yes	Yes	Registers	Partially unrolled

block come from a common super-tile, and thus are contiguous, and their union can be expressed with a polyhedron, even with parametric tiling. This is the same for successive points of a given thread. We do not expect this contiguity to degrade performance for two reasons as we now explain (anyway, we could explore a block-cyclic distribution).

Concerning the block level, the L2 cache is small, its total size is the same as the sum of the scratchpads and L1 caches of all SM. This means that if we efficiently use the scratchpad and L1 caches, and each SM works on a different memory, the L2 cache will only mirror these caches and be of little use to us. We suspect the interest of the L2 cache is for codes that behave in unpredictable manner and require the SMs to share a lot of data and abuse of atomic operations to avoid stepping on each other's feet. This is not our case, our codes are well behaved and relatively simple. Nevertheless, we may indeed lose some spatial locality among blocks, but we think this should be acceptable (also, our preliminary experiments did not show evidence of degradation).

On the thread level however, this is another story. Let us consider some actual numbers for our GPU: our SM can execute 6 instructions (of width 32) in parallel thanks to the number of floating point computation units. However, there are only 4 warp schedulers per SM (extracting up to 2 instructions per cycle, in each warp). To get full performance, this means that at least 2 of these 6 instructions have to come from the same warp. In the PPCG schedule, these two instructions would be on data far away due to the interlacing. In our scheme, these instructions are next to each other, but the 4 considered warps are further apart. All in all, there is no clear benefit to such interlacing, and our preliminary experiments did not show significant difference in performance either way.

5.5.5 Blocks and Global/Shared Memory Transfer

Once a kernel is launched, after its data has been copied from global memory or was already there due to a previous kernel, the computation is split into a grid of blocks. Each SM of the GPU picks a block from this grid, allocates the requested shared memory in the scratchpad (if needed), and launches the necessary amount of warps (and thus allocates

their registers) to match the number of threads per block. If a block under-utilizes the SM, it can launch other parallel blocks until reaching one of its limits. It is important to carefully choose the number of threads per block and the amount of shared memory so as to fill the register file and the scratchpad at their maximum, at the same time.

As a reminder, PPCG is controlled by several parameters in this regard: tile size, block size, and maximum shared memory. The grid size does not influence the amount of shared memory allocated per block, or the number of threads (except for the control of outer loops when a block executes multiple tiles). The tile size is the size of PPCG tiles, and the maximum shared memory is the limit of what we allow PPCG to allocate in shared memory for a given block. It mostly depends on PPCG tile sizes. The block size only influences the amount of instruction-level parallelism (ILP), which will influence the number of registers. This will be discussed in the next subsection.

The allocation on the scratchpad should be made easy thanks to the techniques of Chapter 4 (a simple successive modulo approach may be sufficient however). Modulo allocation has the added benefit of allowing a turnover of the data. Some data from previous tiles stay when we overwrite the parts that are not needed anymore, without any copy. It can however increase the cost of the addresses computation as modulo operations can be expensive in general. On the subject, all the techniques of Chapter 4 compute lower bounds of moduli in such a way that any increase in the moduli still produces a valid allocation. This allows us to use powers of two, whose modulo can be implemented with simple Boolean logic.

The transfers from global memory to local (i.e., shared) memory are done by the block itself and, as the only active elements of a block are the threads, they have to share the work of executing these transfers. PPCG uses a simple strategy, which consists in tiling (paving here) the data (portion of arrays) to be transferred with tiles of the size of the block (i.e., number of threads), so that each thread then download the data cells that are mapped to its corresponding point. For example, see in Figure 5.7 how the arrays `A` and `B` are moved from global to shared memory for each tile, through explicit transfers shared by the threads. This, again, corresponds to performing copy operations following an interlaced scheme. But, in this case, the interlacing is not a problem to us, as we were assuming parallel transfers of such data anyway, so their order is not significant. There are however two other remarks to be done related to these transfers.

First, the computation of the address to be loaded or stored, while not taking a significant amount of time compared to the transfers, introduces bubbles in the pipeline of these transfers. If there are enough threads to fill those bubbles, there is no problem (i.e., sufficient occupancy), but as Volkov [77] demonstrated, increased occupancy means

a lower number of registers per thread. As these registers constitute the memory with the highest throughput available, it is important to make an efficient use of them, through reuse. This means that bigger tile sizes at the instruction level (what we called level-1 tiles), increasing instruction-level parallelism and register reuse, and a reduced number of threads, is to be preferred (and this is true for the computations themselves too, not just for the code responsible for the transfers, see for example the PPCG code of Figure 5.8 and the discussion in the next subsection). As suggested by Volkov, transferring multiple consecutive values by each thread, by transferring quad-floats, reduces the number of address computations (one per quad-float) and increases the throughput without requiring more threads. We suggest exploring this direction, and this was one of the reasons for exploring approximations in Chapter 2. That is, we may transfer slightly more data than needed (multiple of quad-floats for example) but at a higher throughput. The difficulty is that the reuse analysis then becomes tricky (but can be addressed as we explained), since loading slightly more data may overwrite the data produced by a previous tile while it was not yet committed to memory as per the reuse.

Second, the introduction of our reuse analysis at this level should enable pipelining. PPCG currently proceeds as follows: each thread loads one part (but not necessarily the part it will work on) of the data needed by the tile, then the threads synchronize so as to ensure they can start working on correct data, they compute, they synchronize so as to ensure everybody finished computing, they transfer back to shared memory the results of the computation, and they synchronize again. To mitigate the cost of these synchronizations, PPCG detects when they are useless. The two of the most frequent cases are: because some data have been fully placed to registers (because they are local to a thread), in which case the remaining transfers are usually hoisted out of the inner loops, and the synchronizations will happen at the level they end up, or because writes and reads happen on different arrays, thus saving the last synchronization. Our inter-tile reuse analysis can help eliminating more synchronizations, even more when pipelining is used. For that, many pipelines could be envisioned. The triple buffering pipeline of Figure 5.3b for example can be implemented using a single barrier. At each iteration the data from the tile $T - 1$ is stored, while the data for the tile $T + 1$ is loaded, and the computation of the tile T is executed. Experiments with such pipelines have not been done but seems worth trying. However, such a scheme requires more local memory, which might influence performance as it calls for smaller tile sizes.

```
__global__ void kernel0(float *A, float *B, float *C, int n) /* n=12288 */
{
    /* Grid: 192*192 blocks, each with 16*16 threads */
    int b0 = blockIdx.y, b1 = blockIdx.x;
    /* Loops: 384*384*768 tiles, each with 32*32*16 points */
    int t0 = threadIdx.y, t1 = threadIdx.x;
    /* Thus 1 block = 2*2*768 tiles, 1 thread = 2*2*16 points */
    __shared__ float shared_A[32][16];
    __shared__ float shared_B[16][32];
    float private_C[2][2];

    /* 6144 = 32 (tile size) * 192 (number of blocks) */
    for (int g1 = 32 * b0; g1 <= 12256; g1 += 6144)
      for (int g3 = 32 * b1; g3 <= 12256; g3 += 6144) {
        /* 2*2 points unrolled for register usage */
        private_C[0][0] = C[(t0 + g1) * 12288 + (t1 + g3)];
        private_C[0][1] = C[(t0 + g1) * 12288 + (t1 + g3 + 16)];
        private_C[1][0] = C[(t0 + g1 + 16) * 12288 + (t1 + g3)];
        private_C[1][1] = C[(t0 + g1 + 16) * 12288 + (t1 + g3 + 16)];
        /* 16 consecutive points along k in a thread */
        for (int g9 = 0; g9 <= 12272; g9 += 16) {
          /* 2 iterations, as 16*32 to bring with 16*16 threads */
          for (int c1 = t1; c1 <= 31; c1 += 16)
            shared_B[t0][c1] = B[(t0 + g9) * 12288 + (g3 + c1)];
          /* 2 iterations as 32*16 to bring with 16*16 threads */
          for (int c0 = t0; c0 <= 31; c0 += 16)
            shared_A[c0][t1] = A[(g1 + c0) * 12288 + (t1 + g9)];
          __syncthreads();
          /* compute the 16 consecutive points along k */
          for (int c2 = 0; c2 <= 15; c2 += 1) {
            /* unrolled for register usage */
            private_C[0][0] += (shared_A[t0][c2] * shared_B[c2][t1]);
            private_C[0][1] += (shared_A[t0][c2] * shared_B[c2][t1 + 16]);
            private_C[1][0] += (shared_A[t0 + 16][c2] * shared_B[c2][t1]);
            private_C[1][1] += (shared_A[t0 + 16][c2] * shared_B[c2][t1 + 16]);
          }
          __syncthreads();
        }
        C[(t0 + g1) * 12288 + (t1 + g3)] = private_C[0][0];
        C[(t0 + g1) * 12288 + (t1 + g3 + 16)] = private_C[0][1];
        C[(t0 + g1 + 16) * 12288 + (t1 + g3)] = private_C[1][0];
        C[(t0 + g1 + 16) * 12288 + (t1 + g3 + 16)] = private_C[1][1];
        __syncthreads();
      }
}
```

Figure 5.8 – PPCG-generated code (GPU part) for matrix product, with ILP.

5.5.6 Threads and Shared/Register Memory Transfer

The register memory level is implicit in CUDA. A CUDA thread can have access to as many as 63 registers on our experimental setup, but the most recent architectures can provide up to 256 registers per thread. In practice, the highest performance kernels, highly tuned, make a meticulous use of these registers. While being the most important memory level for a performance point of view, as it offers the highest bandwidth and lowest latency possible, it is also the hardest to manage as the control utilizes some of the registers (which is hard to predict in the polyhedral model) and interferes with our allocation, in addition to the fact that the back-end compiler (which decides how registers are used) is not accessible at this level of abstraction. In case too many registers are needed, the GPU will spill to global memory, which might be expensive. These complications mean that benchmarking for different tile sizes is probably inevitable. Nevertheless, parameterization should be useful, as it makes possible to specify the unroll factor without interfering with the remaining of the control.

Finally, one of the strategies of PPCG to make CUDA use registers for an array is to make sure its accesses are with constant subscripts. To obtain such accesses, the parallel loops iterating on the different points assigned to a thread are moved at the innermost position, then unrolled. This can be seen on the code of Figure 5.8 (with 4 iterations per thread, assuming perfect divisibility between each tile level) with the computation of 4 elements of `C` in the loop body. This has the added benefit of increasing the instruction-level parallelism of the threads, and as said earlier, it is important to preserve some parallelism at this level so that the warp schedulers can issue enough instructions to the CUDA cores. Our level-1 tiling has the same effect, and is dedicated to instruction-level parallelism and these registers. We found conceptually simpler to clearly identify these five levels of tiling as opposed to the PPCG formalization where some are an implicitly induced by the interlacing of the "tiles" with the grid and blocks. In our case, all our tile sizes are explicit and not a consequence of the different ratios between the size of the iteration domain, the tile size, the number of blocks (grid size), and the number of threads (block size).

5.6 Conclusion

This chapter illustrated how the techniques exposed in previous chapters can be chained to provide a generic approach to the problem of kernel offloading. We showed how to analyze a double-buffering pipeline with parametric tiles, how to compute the resulting

conflict set (and differences), and how to provide a modulo mapping for the local memory. We detailed the different hurdles that can arise in practice, due to the complexity of either the kernel being analyzed or the underlying architecture and programming framework.

While we initially designed our technique for offloading, we showed that it is well adapted to different usages, such as to handle different cache levels and memory hierarchies. Our analysis was centered around GPUs, with 5 levels of hierarchical tiling, but was based on previous work on FPGAs and should possibly apply to other architectures. Work remains to be done as the implementation targeting GPU was not finished. The cost induced by address computation and control, which influences the resulting performance, should be studied further. Multiple GPUs considerations might be an interesting route to pursue. We finally illustrated how difficult finding the right tile sizes can be, due to multiple factors, thereby motivating the need for parametric tiling and cost models for tile size selection.

Conclusion

> I may not have gone where I intended to go, but I think I have ended up where I intended to be.
>
> Douglas Adams

As heat concerns push architecture design towards mass parallelism and as the gap between computing power and memory throughput get wider, the needs for parallelism and locality are becoming the main barrier to performance and are therefore at the heart of most of our contributions. In this regard, the polyhedral model is a valuable framework because it provides both the sufficient expressiveness and flexibility to model the execution of a wide range of compute-intensive programs, to support exact and approximated parametric analyses of their behavior, to explore and select adequate optimization strategies, and to rigorously implement complex code transformations.

It is recognized as being a promising building block for compilers that undertake automatic parallelization, at least in some domain-specific situations or for some specific architectures, and it recently made major progress with tools that gained the sufficient maturity to reliably implement polyhedral techniques inside modern compilers. There is however a lot of work remaining, both to expand the applicability of the model to programs that are outside the boundaries of the classic framework, and to extend the transformations and analyses that can be applied to them.

We extended polyhedral techniques to provide an analysis of inter-tile data reuse for tiled codes with parametric tile sizes. This should have many applications as tiling is the flagship optimization of the polyhedral model, because it improves both locality and parallelism, providing a trade-off between memory usage and processing power. A pipelined execution of these tiles and their associated data transfers is necessary to make use of the full capability of the underlying architecture and this requires a correct inter-tile data reuse analysis. By designing a parametric analysis of loaded and stored data for each

tile, we opened the road to cost models for memory transfers, for the generation of correct parametric pipelined codes, and for further analysis, such as liveness analysis, necessary for the estimation of the size of local storage buffers as well as their allocation and mapping strategy. Our analysis being applicable to approximated read and write accesses, we can use it for programs that are outside of the classic exact polyhedral framework, as well as for programs that fit the polyhedral model but that would be too expensive to model exactly and for which an approximation could give good results for a cheaper analysis. However, work remains to be done on this aspect. Indeed, while our analysis supports approximations of memory accesses, we did not provide an actual approximation scheme. Also, we gave a parametric description of the communication associated with a tiled execution, but a cost model remains to be developed to choose the right tile sizes.

Liveness analysis of the pipelined code, required for memory mapping, proved to be not so easy as the pipelined schedule we used expresses parallelism beyond the expressiveness of the classic scattering functions. This motivated the generalization of previous approaches to compute conflicts between memory locations over a less constrained form of schedule, one described by a happens-before relation. This allowed us to compute a conservatively-correct conflict analysis between array elements not only for a polyhedral sequential input program but also for programs exhibiting some non-deterministic behavior due to parallelism or some unpredictable control that cannot be caught by the polyhedral model. This study was motivated for both aspects by our pipelined parametric tiling as it exhibits not only complex parallelism (due to pipelining introducing partial parallelism) but also unpredictable/inexpressible control (due to unaligned tiles used to get rid of non-linear constraints coming from the parametric tiling). We believe this understanding of parallel specifications expressible with piece-wise affine happens-before relations can have many usages, notably when applied to languages that provide intrinsic parallel constructs such as OpenMP (loop parallelism but also task parallelism as in OpenMP 4.0), X10 (though async/finish keywords), OpenStream, etc.

While the technique we used for array contraction/memory mapping (that is successive modulo allocation) gave relatively good results at first, the introduction of tiling created examples for which the memory conflict sets was not only non-convex, but sometimes corresponded to the worst-case situation of the technique. A good basis selection is often sufficient to solve this issue and obtain mappings close to optimal, but previous techniques did not automatically choose such a basis. Methods based on universal occupancy vectors could give good results (albeit in a limited setting) but only provide the first axis. We thus designed a new polyhedral technique for array contraction in order to reliably build a compact modulo allocation in cases where a change in mapping direction

is needed. It is an extension to the previous work on lattice-base memory allocation and it gives results that are at least as good as universal occupancy vectors. It handles non-convex conflict sets naturally and finds a basis close to the optimal on every test cases to which we applied it (when our first heuristic, looking for successive directions with small widths, is not sufficient). This should lead to a wide range of usages, as the only required information is the conflict set of memory cells and is thus independent of the underlying schedule. We believe it may have interesting applications for languages that require buffer allocation, such as stream languages, single assignment languages, array languages, where the memory mapping is abstracted away from the programmer.

We applied our techniques to perform kernel offloading to GPU architectures. Recursive tiling can be used to optimize for locality at each level of the memory hierarchy. Previous work showed that this can give excellent results provided good tile sizes are chosen, at each tile level. We showed how an intermediate tiling level can be inserted so as to introduce sequentiality and benefit from inter-tile data reuse.

However, there is still work to be done. Notably, we did not provide an automated tile size selection heuristic, a manual (or automated) search in the tile-sizes space is still required. The implementation into PPCG turned out to require a substantial rewrite, which was not completed. Not only the code generator needed complete redesign, as relying on `isl` code generator would incur the integration of parametric tiling inside `isl` itself, but also most analyses have to be rewritten to take into account our techniques on the edge of the polyhedral model, from the description of unaligned tiles for capturing parametric tiling to the use of pipelined schedules, not directly expressible with `isl` schedule trees. We think such an integration is however possible but, so, far we relied on semi-automatically generated scripts on top of `iscc`. We also ran into difficulties to implement our proposed pipeline using CUDA streams due to the many ways implicit synchronizations can occur. Although feasible, this required to fill these streams in a meticulous way (in topological order), which again complicates code generation.

Besides implementation and evaluation, many further improvements are possible. For example, for multiple kernels (and GPUs), a data reuse analysis can help reducing the memory transfers between consecutive kernels. Optimizing device-to-device communications is an interesting track to follow too. Also, while we focused on GPUs because they are a good target for polyhedral optimizations, we believe this work is general enough to apply to a wide range of accelerators, from multicores to FPGAs. A promising application, among others, is to analyze communications between pipelined tasks and automate the allocation of and access to intermediate buffers. This is a direct application of our liveness analysis and an excellent candidate for our extended lattice-based memory mapping.

Bibliography

[1] Aravind Acharya and Uday Bondhugula. PLUTO+: Near-complete modeling of affine transformations for parallelism and locality. In *20th ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*, pages 54–64, San Francisco, February 2015.

[2] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+Cl@k: An implementation of lattice-based array contraction in the source-to-source translator Rose. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07)*, San Diego, June 2007.

[3] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing DDR-SDRAM communications at C-level for automatically-generated hardware accelerators. An experience with the Altera C2H HLS tool. In *21st IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'10)*, pages 329–332, Rennes, July 2010. IEEE Computer Society.

[4] Christophe Alias, Alain Darte, and Alexandru Plesco. Kernel offloading with optimized remote accesses. Technical Report RR-7697, Inria, July 2011.

[5] Christophe Alias, Alain Darte, and Alexandru Plesco. Optimizing remote accesses for offloaded kernels: Application to HLS for FPGA. In *Design, Automation and Test in Europe (DATE'13)*, pages 575–580, Grenoble, March 2013.

[6] Vinayak Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, pages 40:1–40:11, Salt lake city, Utah, USA, November 2012. IEEE Computer Society.

[7] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 1–10, 2008.

[8] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. Parameterized tiling revisited. In *8th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'10)*, pages 200–209. ACM, 2010.

[9] Muthu Manikandan Baskaran, Nicolas Vasilache, Benoît Meister, and Richard Lethin. Automatic communication optimizations through memory reuse strategies. In *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'12)*, pages 277–278, New Orleans, February 2012.

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

[11] Samuel Bayliss and George A. Constantinides. Optimizing SDRAM bandwidth for custom FPGA loop accelerators. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*, pages 195–204. ACM, 2012.

[12] David A. Berson, Rajiv Gupta, and Mary Lou Soffa. URSA: A unified resource allocator for registers and functional units in VLIW architectures. In *Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism (PACT'93)*, pages 243–254, Orlando, Florida, January 1993.

[13] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Automatic intra-array storage optimization. Technical Report IISc-CSA-TR-2014-3, Indian Institute of Science, November 2014.

[14] Someshekaracharya Bhaskaracharya, Uday Bondhugula, and Albert Cohen. SMO: An integrated approach to intra-array and inter-array storage optimization. In *43rd Annual Symposium on Principles of Programming Languages (POPL'16)*, St Petersburg, Florida, January 2016.

[15] Uday Bondhugula, Muthu Manikandan Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *17th International Conference on Compiler Construction (CC'08)*, pages 132–146, 2008.

[16] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM International Conference on Programming Languages Design and Implementation (PLDI'08)*, pages 101–113, Tucson, June 2008.

[17] Srinivas Boppu, Franck Hannig, and Jürgen Teich. Loop program mapping and compact code generation for programmable hardware accelerators. In *24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP'13)*, pages 10–17, Washington, DC, June 2013.

[18] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation: What does the NP-completeness proof of Chaitin et al. really prove? In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'06)*, volume 4382 of *LNCS*, pages 283–298. Springer Verlag, November 2006.

[19] Mathias Bourgoin, Emmanuel Chailloux, and Jean Luc Lamotte. Efficient abstractions for GPGPU programming. *International Journal of Parallel Programming*, 42(4):583–600, 2014.

[20] Michal Cierniak and Wei Li. Recovering logical data and code structures. Technical Report 591, University of Rochester, 1995.

[21] Alessandro Cilardo and Luca Gallo. Improving multibank memory access parallelism with lattice-based partitioning. *ACM Transactions on Architecture and Code Optimizations (TACO)*, 11(4):45:1–45:25, 2014.

[22] Albert Cohen. *Program Analysis and Transformation: From the Polytope Model to Formal Languages*. PhD thesis, Université de Versailles, December 1999.

[23] Albert Cohen and Vincent Lefebvre. Storage mapping optimization for parallel programs. In *5th International Euro-Par Parallel Processing Conference (Euro-Par'99)*, pages 375–382, 1999.

[24] Béatrice Creusillet and François Irigoin. Interprocedural array region analyses. In *International Workshop on Languages and Compilers for Parallel Computing (LCPC'96)*, volume 1033 of *LNCS*, pages 46–60. Springer, 1996.

[25] Alain Darte. Optimal parallelism detection in nested loops. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[26] Alain Darte and Alexandre Isoard. Parametric tiling with inter-tile data reuse. In *4th International Workshop on Polyhedral Compilation Techniques (IMPACT'14)*, Vienna, Austria, January 2014.

[27] Alain Darte and Alexandre Isoard. Exact and approximated data-reuse optimizations for tiling with parametric sizes. In *24th International Conference on Compiler Construction (CC'15)*, pages 151–170, London, UK, April 2015. Nominated as best paper candidate for ETAPS.

[28] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. Extended lattice-based memory allocation. In *25th International Conference on Compiler Construction (CC'16)*, pages 218–228. ACM, 2016.

[29] Alain Darte, Alexandre Isoard, and Tomofumi Yuki. Liveness analysis in explicitly-parallel programs. In *6th International Workshop on Polyhedral Compilation Techniques (IMPACT'16)*, Prague, Czech Republic, January 2016.

[30] Alain Darte, Leonid Khachiyan, and Yves Robert. Linear scheduling is nearly optimal. *Parallel Processing Letters*, 1(2):73–81, 1991.

[31] Alain Darte, Rob Schreiber, and Gilles Villard. Lattice-based memory allocation. In *6th ACM International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'03)*, pages 298–308, San Jose, CA, USA, October 2003.

[32] Alain Darte, Robert Schreiber, and Gilles Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, October 2005.

[33] Eddy De Greef, Francky Catthoor, and Hugo De Man. Memory size reduction through storage order optimization for embedded parallel multimedia applications. *Parallel Computing*, 23:1811–1837, 1997.

[34] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. Leblanc. Design on ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, SC-9(5):256–268, October 1974.

[35] Paul Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. Corresponding software tool PIP: `http://www.piplib.org/`.

[36] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.

[37] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, 1992.

[38] Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[39] Paul Feautrier. The power of polynomials. In *5th International Workshop on Polyhedral Compilation Techniques (IMPACT'15)*, Amsterdam, The Netherlands, January 2015.

[40] Paul Feautrier and Christian Lengauer. The polyhedron model. In David Padua, editor, *Encyclopedia of Parallel Programming*. Springer, 2011.

[41] Georgios I. Goumas, Maria Athanasaki, and Nectarios Koziris. An efficient code generation technique for tiled iteration spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, 2003.

[42] Tobias Grosser, Albert Cohen, Paul H. J. Kelly, J. Ramanujam, P. Sadayappan, and Sven Verdoolaege. Split tiling for GPUs: Automatic parallelization using trapezoidal tiles. In *6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*, pages 24–31, New York, NY, USA, 2013. ACM.

[43] Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. Optimistic delinearization of parametrically sized arrays. In *29th ACM on International Conference on Supercomputing (ICS'15)*, pages 351–360, New York, NY, USA, 2015. ACM.

[44] Armin Größlinger. Precise management of scratchpad memories for localising array accesses in scientific codes. In *18th International Conference on Compiler Construction (CC'09)*, pages 236–250, 2009.

[45] Serge Guelton, Mehdi Amini, and Béatrice Creusillet. Beyond do loops: Data transfer generation with convex array regions. In Hironori Kasahara and Keiji Kimura, editors, *International Workshop on Languages and Compilers for Parallel Computing (LCPC'13)*, volume 7760 of *LNCS*, pages 249–263. Springer, 2013.

[46] Serge Guelton, Ronan Keryell, and François Irigoin. Compilation pour cible hétérogènes: automatisation des analyses, transformations et décisions nécessaires. In *20ème Rencontres Françaises du Parallélisme (Renpar'11)*, Saint Malo, France, May 2011.

[47] Albert Hartono, Muthu Manikandan Baskaran, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. DynTile: Parametric tiled loop generation for parallel execution on multicore processors. In *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*, pages 1–12. IEEE, 2010.

[48] François Irigoin and Rémi Triolet. Supernode partitioning. In *15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 319–329, San Diego, California, 1988. ACM.

[49] Ilya Issenin, Erik Borckmeyer, Miguel Miranda, and Nikil Dutt. DRDU: A data reuse analysis technique for efficient scratch-pad memory management. *ACM Transactions on Design Automation of Electronics Systems (ACM TODAES)*, 12(2), April 2007. Article 15.

[50] Mahmut Kandemir, Ismail Kadayif, Alok Choudhary, J. Ramanujam, and Ibrahim Kolcu. Compiler-directed scratch pad memory optimization for embedded multiprocessors. *IEEE Transactions on VLSI Systems*, 12(3):281–287, March 2004.

[51] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The Omega calculator and library. Technical report, University of Maryland, nov 1996.

[52] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'11)*, pages 277–288. ACM, 2011.

[53] Seyong Lee and Rudolf Eigenmann. OpenMPC: Extended OpenMP programming and tuning for GPUs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, pages 1–11. IEEE Computer Society, 2010.

[54] Vincent Lefebvre and Paul Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24:649–671, 1998.

[55] David Levinthal. *Performance Analysis Guide for Intel® CoreTM i7 Processor and Intel® XeonTM 5500 processors*. Intel, 2010.

[56] Vadim Maslov and William Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. In Springer Verlag, editor, *CONPAR 94—VAPP VI, International Conference on Parallel and Vector Processing*, number 854 in LNCS, pages 737–748, Linz, Austria, September 1994.

[57] Morris Newman. *Integral Matrices*. Academic Press, 1972.

[58] OpenACC, directives for accelerators. `http://www.openacc.org/`.

[59] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, pages 33–42. ACM, 2012.

[60] Alexandru Plesco. *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*. PhD thesis, École normale supérieure de Lyon, 2010.

[61] PLUTO: An automatic polyhedral parallelizer and locality optimizer for multicores. `http://pluto-compiler.sourceforge.net`.

[62] Louis-Noël Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*, pages 29–38. ACM, 2013.

[63] Louis-Noël Pouchet. PolyBench/C, the polyhedral benchmark suite. `http://sourceforge.net/projects/polybench/`.

[64] Fabien Quilleré and Sanjay Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(5):773–815, 2000.

[65] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay V. Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI'07)*, pages 405–414, San Diego, June 2007.

[66] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification version 2.2, March 2012. `x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[67] Alexander Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, New York, 1986.

[68] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'98)*, pages 24–33, 1998.

[69] Lewis Terman and Mary Y. Lanzerotti, editors. *The Impact of Dennard's Scaling Theory*, volume 12.1 of *SSCS, IEEE Solid-State Circuits Society News*. IEEE, Winter 2007.

[70] William Thies, Frédéric Vivien, and Saman P. Amarasinghe. A step towards unifying schedule and storage optimization. *ACM Transactions on Programming Languages and Systems (ACM TOPLAS)*, 29(6), 2007.

[71] Sid Ahmed Ali Touati. Register saturation in instruction level parallelism. *International Journal of Parallel Programming*, 33(4):393–449, 2005.

[72] Ramakrishna Upadrasta and Albert Cohen. Sub-polyhedral scheduling using (unit) two-variable-per-inequality polyhedra. In *40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'13)*, pages 483–496, Roma, Italy, January 2013.

[73] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software - ICMS 2010*, volume 6327 of *LNCS*, pages 299–302. Springer, 2010. Corresponding library: `http://freecode.com/projects/isl/`.

[74] Sven Verdoolaege. Counting affine calculator and applications. In *1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[75] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[76] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, 2007. Corresponding library: `barvinok.gforge.inria.fr`.

[77] Vasily Volkov. Better performance at lower occupancy. In *GPU Technology Conference (GTC'10)*, San Jose, CA, 2010.

[78] Doran Wilde and Sanjay Rajopadhye. Memory reuse analysis in the polyhedral model. In *Second International Euro-Par Conference (Euro-Par'96)*, pages 389–397, 1996.

[79] Michael Wolf and Monica Lam. A data locality optimizing algorithm. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pages 30–44, Toronto, Ontario, Canada, 1991. ACM.

[80] Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[81] Jingling Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

[82] Tomofumi Yuki, Paul Feautrier, Sanjay Rajopadhye, and Vijay Saraswat. Array dataflow analysis for polyhedral X10 programs. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*, pages 23–34, February 2013.

[83] Tomofumi Yuki and Sanjay Rajopadhye. Memory allocations for tiled uniform dependence programs. In *3rd International Workshop on Polyhedral Compilation Techniques (IMPACT'13)*, pages 13–22, January 2013.