HAL
open science

A Combined Language and Polyhedral Approach for Heterogeneous Parallelism

Jie Zhao

To cite this version:

Jie Zhao. A Combined Language and Polyhedral Approach for Heterogeneous Parallelism. Distributed, Parallel, and Cluster Computing [cs.DC]. PSL Research University, 2018. English. NNT : . tel-01988073v1

HAL Id: tel-01988073

https://theses.hal.science/tel-01988073v1

Submitted on 21 Jan 2019 (v1), last revised 26 Feb 2020 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'École Normale Supérieure

Une Approche Combinée Langage-Polyédrique pour la Programmation Parallèle Hétérogène

A Combined Language and Polyhedral Approach for Heterogeneous Parallelism

Ecole doctorale n°386

École doctorale de Sciences Mathématiques de Paris Centre

Spécialité Informatique

COMPOSITION DU JURY :

Mme. HALL Mary W.
University of Utah, Rapporteur

M. BASTOUL Cédric
Université de Strasbourg, Rapporteur

M. MCCOLL Bill
Huawei/University of Oxford, Examinateur

M. ZAPPA NARDELLI Francesco
ENS Paris/INRIA, Examinateur

M. TADONKI Claude
Mines ParisTech/CRI, Examinateur

M. COHEN Albert
Google/INRIA, Directeur de these

Soutenue par Jie **ZHAO**
le 13 décembre 2018

Dirigée par **Albert COHEN**

Acknowledgements

To be honest, I had always been doubting about the impressive performance and the surprising magic of the polyhedral model before I joined the PARKAS group. The obscure mathematical abstractions of the model always prevented me from getting to the bottom of the polyhedral world. I therefore came to a decision to go into this world by pursuing a PhD degree majoring this research direction. Today, I eventually reach the end of this path, with the guidance, advices, supports and assistances from a large number of persons to name.

I would like to show my foremost gratefulness to my supervisor, Albert Cohen, for leading me the way to the world of polyhedral compilation and being patient, keen and passionate to help me dig into practical and interesting research ideas. Albert always provided me very valuable feedbacks via in-person discussions and online exchanges of views, leaving me much freedom and space to explore the issues and seek solutions independently. He also gave me plenty of guidance and loads of suggestions with his forward-looking insight, inspiring me to follow the promising research directions in academy and up-and-coming industry issues. It was a great experience advised by Albert and I cannot wait to work with him again.

Many thanks to Michael Kruse, Oleksandr Zinenko, Chandan Reddy and Riyadh Baghdadi for the interesting and helpful conversations during my doctoral study. Talking with them helps me better understand the mathematical background of polyhedral compilation and their constructive suggestions on my work always help me make progress. Thanks to Marc Pouzet, Francesco Zappa Nardelli and Timothy Bourke for leading the PARKAS team and establishing a harmonious environment for the smooth progress of my thesis. I also acknowledge Ulysse Beaugnon, Lélio Brun, and Guillaume Iooss, who helped me a lot on my registration in the doctoral school and the university as well as many other daily activities.

I would also like to acknowledge Bill McColl and his team, including Chong Li, Zhen Zhang, Sheng Yang, and Yu Xia, for hosting me during my visit at Huawei Research at Paris. We have evolved good friendships and they will last for my whole life. The collaboration work at Huawei for building a connected, intelligent world is an unforgettable experience. Thanks to your support and grant for my visit to the Huawei Research Center at Beijing, China. I am expecting to cooperate with you in the future.

I am very grateful to the valuable time and efforts from the reviewers of my PhD dissertation,

Mary W. Hall and Cédric Bastol. Thanks to Mary W. Hall for the help to a better comprehension about sparse computations, with an in-depth look into your approach and the NVIDIA implementations of the applications. Thanks to the practical comments and criticisms from Cédric Bastol for helping improve the quality of this work.

I also feel very thankful to the contributors of the PPCG code generator and the `isl` integer solving library, Sven Verdoolaege and Tobias Grosser. This work would not be possible without the contributions of their work. I would also like to acknowledge the contributors and developers of a large number of underlying components of polyhedral compilers and optimizers, including `pet`, Pluto, CLooG, Omega, `iscc`, etc., without which the understanding of the polyhedral model would be an unimaginable task.

I have to express my deep gratefulness to some important persons who walked with me the path of undergraduate and graduate study, including Baoliang Li, Jinchen Xu, Zhifeng Chen, Guanghui Liang and Xu Zhao, without some of whom my programming knowledge and skills would be impossible.

Last but not least, I want to acknowledge my family for their endless support to my education from the childhood. Everyone needs a house to live in, but a supportive family is what builds a home.

Paris, 10 December 2018 Jie Zhao

Remerciements

Pour être honnête, j'ai toujours douté des performances impressionnantes et de la magie surprenante du modèle polyédrique avant de rejoindre le groupe PARKAS. Les obscures abstractions mathématiques du modèle m'ont toujours empêché d'atteindre le fond du monde polyédrique. J'ai donc pris la décision d'entrer dans ce monde en poursuivant un doctorat dans le domaine de la recherche. Aujourd'hui, J'arrive finalement au bout de ce chemin, avec les conseils, les soutiens et les assistances d'un grand nombre de personnes.

Je tiens à exprimer ma plus profonde gratitude à mon superviseur, Albert Cohen, pour m'avoir conduit au monde de la compilation polyédrique et pour sa patience, sa volonté et sa passion pour m'aider à approfondir des idées de recherche pratiques et intéressantes. Albert m'a toujours fourni des retours très utiles via des discussions en personne et des échanges de vues en ligne, me laissant ainsi beaucoup de liberté et d'espace pour explorer les problèmes et rechercher des solutions de manière autonome. Il m'a également donné beaucoup de conseils et une foule de suggestions grâce à sa vision prospective, m'inspirant de suivre les orientations de recherche prometteuses dans le monde académique et les questions d'actualité. Ce fut une excellente expérience conseillée par Albert et je suis impatient de retravailler avec lui.

Un grand merci à Michael Kruse, à Oleksandr Zinenko, à Chandan Reddy et à Riyadh Baghdadi pour les conversations intéressantes et utiles que j'ai eues au cours de mon doctorat. Parler avec eux m'aide à mieux comprendre le contexte mathématique de la compilation polyédrique et leurs suggestions constructives sur mon travail m'aident toujours à progresser. Merci à Marc Pouzet, Francesco Zappa Nardelli et Timothy Bourke d'avoir dirigé l'équipe de PARKAS et créé un environnement harmonieux pour le bon déroulement de ma thèse. Je tiens également à remercier Ulysse Beaugnon, Lélio Brun et Guillaume Iooss, qui m'ont beaucoup aidé lors de mon inscription à l'école doctorale et à l'université ainsi que de nombreux autres travaux quotidiens.

Je tiens également à remercier Bill McColl et son équipe, notamment Chong Li, Zhen Zhang, Sheng Yang et Yu Xia, de m'avoir accueilli lors de ma visite à Huawei Research à Paris. Nous avons développé de bonnes amitiés et elles dureront toute ma vie. La collaboration chez Huawei pour créer un monde connecté et intelligent est une expérience inoubliable. Merci pour votre soutien et votre subvention pour ma visite au centre de recherche Huawei à Pékin, en Chine. Je compte coopérer avec vous à l'avenir.

Acknowledgements

Je suis très reconnaissant au temps précieux et aux efforts des rapporteurs de ma thèse, Mary W. Hall et Cédric Bastol. Merci à Mary W. Hall pour l'aide à une meilleure compréhension des calculs clairsemés, avec un examen approfondi de votre approche et des implémentations NVIDIA des applications. Merci aux commentaires pratiques et aux critiques de Cédric Bastol pour avoir aidé à améliorer la qualité de ce travail.

Je suis également très reconnaissant aux contributeurs du générateur de code PPCG et de la bibliothèque de résolution intégrale d'`isl`, Sven Verdoolaege et Tobias Grosser. Ce travail ne serait pas possible sans les contributions de leur travail. J'aimerais également remercier les contributeurs et les développeurs d'un grand nombre de composants sous-jacents de compilateurs et optimiseurs polyédriques, y compris `pet`, Pluto, CLooG, Omega, `iscc`, etc., sans lesquels la compréhension du modèle polyhédral serait une tâche inimaginable.

Je dois exprimer ma profonde gratitude à certaines personnes importantes qui ont suivi avec moi le chemin des études de premier et deuxième cycles, notamment Baoliang Li, Jinchen Xu, Zhifeng Chen, Guanghui Liang et Xu Zhao, sans lesquelles mes connaissances et mes compétences en programmation seraient impossible.

Enfin, je tiens à remercier ma famille pour son soutien sans faille à mon éducation dès mon enfance. Tout le monde a besoin d'une maison, mais c'est une famille qui apporte son soutien qui construit une maison.

A Paris, le 10 décembre 2018 Jie Zhao

Abstract

After a thirty-year history of development, the polyhedral model has evolved into a powerful solution to exploiting automatic parallelization and locality optimization. As bridging software between the high-level description of programs and the underlying implementation of hardware, polyhedral compilation is increasingly challenged by the diversity of programming languages and heterogeneity of architectures. A long standing limitation of the model has been its restriction to static control affine programs, resulting in an emergent demand for the support of non-affine extensions. This is particularly acute in the context of heterogeneous architectures where a variety of computation kernels need to be analyzed and transformed to match the constraints of hardware accelerators and to manage data transfers across memory spaces.

We explore multiple non-affine extensions of the polyhedral model, in the context of a well-defined intermediate language combining affine and syntactic elements. The thesis is organized as follows.

In the first part, we explain the challenges faced by the polyhedral model with respect to programming languages and architectures, and provide a brief introduction to polyhedral compilation.

In the second part, we present a method to parallelize and optimize loop nests for an important class of programs where counted loops have a dynamic data-dependent upper bound. Such loops are amenable to a wider set of transformations than general `while` loops with inductively defined termination conditions: for example, the substitution of closed forms for induction variables remains applicable, removing the loop-carried data dependences induced by termination conditions.

Our approach relies on affine relations only, as implemented in state-of-the-art polyhedral libraries. Revisiting a state-of-the-art framework to parallelize arbitrary `while` loops, we introduce additional control dependences on data-dependent predicates. Our method goes beyond the state of the art in fully automating the process, specializing the code generation algorithm to the case of dynamic counted loops and avoiding the introduction of spurious loop-carried dependences. We conduct experiments on representative irregular computations, from dynamic programming, computer vision and finite element methods to sparse matrix linear algebra. We validate that the method is applicable to general affine transformations for

locality optimization, vectorization and parallelization.

In the third part, we propose an automatic implementation of non-affine transformations by revisiting overlapped tiling in polyhedral compilation. Polyhedral frameworks implement classical forms of rectangular/parallelogram tiling affine transformations, but these forms lead to pipelined start and rather inefficient wavefront parallelism. Some experimental branches of existing polyhedral compilers evaluated sophisticated shapes such as trapezoid or diamond tiles, enabling concurrent start along the axes of the iteration space, but leading to custom scheduling and code generation methods insufficiently integrated with the general framework. Overlapped tiling is a technique designed to eliminate pipelined start by modifying tile shapes obtained from existing frameworks, but no implementations in a general-purpose polyhedral framework has been available until now, preventing its application in general-purpose loop-nest optimizers and hampering the fair comparison with other techniques.
We revisit overlapped tiling in polyhedral compilation and demonstrate how to derive tighter tile shapes with less redundant computations, by enabling overlapped tiles in a schedule-tree-based algorithm. Our method allows the generation of both acute and right trapezoid shapes. It goes beyond the state of the art by avoiding the restriction to a domain-specific language or introducing post-pass rescheduling and custom code generation. We conduct experiments on the PolyMage benchmarks and representative iterated stencils, validating the effectiveness and general applicability of our technique on both general-purpose multicores and accelerators.

Finally, we summarize our work and present concluding remarks as well as future research directions. We believe the contributions collected in this dissertation extend the reach fof the polyhedral model to wider ranges of real-world programs. We also believe this work contributes to the integration of polyhedral methods with other compilation techniques.

Résumé

Après trente ans de développement, le modèle polyédrique est devenu une solution puissante pour exploiter la parallélisation automatique et l'optimisation de la localisation. En tant que logiciel de transition entre la description de haut niveau des programmes et la mise en œuvre sous-jacente du matériel, la compilation polyédrique est de plus en plus remise en cause par la diversité des langages de programmation et l'hétérogénéité des architectures. Un défaut de longue date du modèle est sa restriction aux programmes affines de contrôle statique, entraînant une demande émergente pour la prise en charge des extensions non affines, en particulier à l'ère des architectures hétérogènes.

Nous étudions les extensions non affines dans le modèle polyédrique en le combinant avec un langage intermédiaire bien défini. La thèse est organisée comme ci-dessous.

Dans la première partie, nous expliquons les défis rencontrés par le modèle polyédrique en ce qui concerne les langages de programmation et les architectures, et décrivons une brève introduction à la compilation polyédrique pour aider les lecteurs à comprendre le principe du travail.

Dans la seconde partie, nous présentons l'approche du traitement des applications non affines en étudiant la compilation parallélisante et l'optimisation d'imbrication en boucle d'une classe importante de programmes où les boucles comptées ont une limite supérieure dynamique dépendante des données. De telles boucles se prêtent à un ensemble de transformations plus large que les boucles générales `while` avec des conditions de terminaison inductives : par exemple, la substitution des formes fermées par les variables d'induction reste applicable, éliminant les dépendances induites par les conditions de terminaison. Nous proposons une méthode de compilation automatique pour paralléliser et optimiser les boucles comptées dynamiques.

Notre approche repose uniquement sur des relations affines, mises en œuvre dans des bibliothèques polyédriques à la pointe de la technologie. En revisitant un cadre de pointe pour paralléliser des boucles arbitraires `while`, nous introduisons des dépendances de contrôle supplémentaires sur les prédicats dépendant des données. Notre méthode va au-delà de l'état de la technique en automatisant complètement le processus, en spécialisant l'algorithme de génération de code au cas des boucles comptées dynamiques et en évitant l'introduction de dépendances parasites en boucle. Nous effectuons des expériences sur des calculs irréguliers

représentatifs, allant de la programmation dynamique, de la vision par ordinateur et des méthodes par éléments finis à l'algèbre linéaire à matrice fragmentée. Nous validons que la méthode est applicable aux transformations affines générales pour l'optimisation de la localité, la vectorisation et la parallélisation.

Dans la troisième partie, nous proposons une implémentation automatique des transformations non affines en revisitant les mosaïques superposées dans la compilation polyédrique. Les structures polyédriques mettent en œuvre des formes classiques de transformations affines de carrelage rectangulaire/parallélogramme, mais ces formes conduisent à un démarrage en pipeline et à un parallélisme de front d'onde plutôt inefficace. Certaines branches expérimentales de compilateurs polyédriques existants ont évalué des formes sophistiquées telles que des carreaux trapézoïdaux ou diamantés, permettant un démarrage simultané sur les axes de l'espace d'itération, mais conduisant à des méthodes de planification et de génération de code insuffisamment intégrées au cadre général. Le pavage superposé est une technique conçue pour éliminer le démarrage en pipeline en modifiant les formes de pavés obtenues à partir de structures existantes, mais aucune implémentation dans une structure polyédrique polyvalente n'était disponible jusqu'à présent, empêchant son application dans les optimiseurs de boucle comparaison avec d'autres techniques.
Nous revisitons les mosaïques superposées dans la compilation polyédrique et montrons comment obtenir des formes de mosaïques plus étroites avec des calculs moins redondants, en activant des mosaïques superposées dans un algorithme basé sur un calendrier. Notre méthode permet de générer des formes trapézoïdales aiguës et droites. Cela dépasse l'état de la technique en évitant la restriction à un langage spécifique à un domaine ou en introduisant une reprogrammation post-pass et une génération de code personnalisée. Nous effectuons des expériences sur les repères PolyMage et les gabarits itératifs représentatifs, validant ainsi l'efficacité et l'applicabilité générale de notre technique sur les multicœurs et les accélérateurs polyvalents.

Enfin, nous résumons notre travail et discutons de quelques remarques de conclusion pour les futures directions de recherche. Le travail de cette thèse met le modèle polyédrique en application dans des programmes réels, en étendant les champs applicables du modèle et en soutenant l'intégration avec d'autres algorithmes de compilation.

Contents

Acknowledgements (English/Français) **iii**

Abstract (English/Français) **vii**

List of figures **xiii**

List of tables **xvi**

Part I : Introduction & Background **1**

1 Introduction **3**
1.1 From General-purpose Languages to Domain-specific Languages 4
1.2 Architecture Diversity 5
1.3 Beyond Parallelization and Locality Optimization 7
1.4 Combining Languages and the Polyhedral Model 8

2 Background **11**
2.1 Polyhedral Compilation 11
2.1.1 An Overview 12
2.1.2 Polyhedral Representations 16
2.1.3 Loop Transformations 19
2.1.4 Schedule Trees 20
2.2 A Platform-Neutral Compute Intermediate Language 22

Part II : Handling Non-affine Applications **25**

3 Dynamic Counted Loops **27**
3.1 Background and Motivation 27
3.1.1 Limitations of Previous Work 28
3.1.2 Static Control Parts 29
3.1.3 Our Solution 30
3.2 Extension Nodes in Schedule Trees 30
3.3 An Overview of Our Approach 31
3.4 Related Work 33

4 Scheduling Dynamic Counted Loops in the Polyhedral Model **35**
4.1 Preparation . . . 35
4.2 Deriving a Static Upper Bound . . . 37
4.2.1 Static Approaches . . . 37
4.2.2 Dynamic Approaches . . . 38
4.3 Modeling Control Dependences . . . 38
4.4 Scheduling . . . 39
4.4.1 Schedule Construction . . . 39
4.4.2 Schedule Transformation . . . 41

5 Generation of Imperative Code **45**
5.1 Extending the Schedule Tree . . . 45
5.2 Generating Early Exits . . . 46
5.3 Changing Back to Dynamic Conditions . . . 47
5.4 Code Generation for a Single Loop . . . 48
5.5 Flat and Nested Parallelisms . . . 49
5.5.1 Flat parallelism within a band . . . 50
5.5.2 Nested parallelism across bands . . . 52
5.6 General Applicability to Loop Transformations . . . 53
5.6.1 Loop Transformations of Unimodular Matrices . . . 53
5.6.2 Loop Transformations in Code Generation . . . 55
5.6.3 Other Loop Transformations . . . 56

6 Experimental Results **57**
6.1 Dynamic Programming . . . 57
6.2 HOG Benchmark . . . 59
6.3 Finite Element Method . . . 60
6.4 Sparse Matrix-Vector Multiplication . . . 61
6.5 Inspector/Executor . . . 64
6.6 Performance on CPU Architectures . . . 64

Part III : Automating Non-affine Transformations **67**

7 Overlapped Tiling **69**
7.1 Background and Motivation . . . 70
7.1.1 Loop Tiling . . . 70
7.1.2 Image Processing Pipelines . . . 71
7.1.3 Limitations of the Hyperplane-based Technique . . . 72
7.1.4 Our solution . . . 73
7.2 Expansion Nodes in Schedule Trees . . . 73
7.3 Related Work . . . 74

8 Acute Trapezoid Tiling and Right Trapezoid Tiling **77**
8.1 Representations in Schedule Trees . . . 77
8.2 Acute Trapezoid Tiling . . . 78
8.3 Right Trapezoid Tiling . . . 80
8.4 Schedule Generation . . . 81
8.5 Removal of Control Overheads . . . 82
8.6 Comparing the Two Trapezoid Tile Shapes . . . 83
8.7 Handling Multi-statement/-dimensional Cases . . . 85
8.7.1 Multiple Statements . . . 85
8.7.2 Multi-dimensional Statements . . . 85
8.8 Complementary Transformations . . . 86
8.8.1 Alignment and Scaling . . . 86
8.8.2 Fusion . . . 86
8.8.3 Reducing Memory Footprint . . . 87
8.8.4 Hybrid Tiling . . . 87

9 Evaluations **89**
9.1 Experimental Setup . . . 89
9.2 Image Processing Pipelines . . . 89
9.2.1 Bilateral Grid . . . 90
9.2.2 Camera Pipeline . . . 91
9.2.3 Harris Corner Detection . . . 92
9.2.4 Local Laplacian Filter . . . 93
9.2.5 Multiscale Interpolation . . . 94
9.2.6 Pyramid Blending . . . 94
9.2.7 Unsharp Mask . . . 95
9.3 Iterated Stencils . . . 96
9.4 Performance on GPU Architectures . . . 98

Part IV : Conclusions **101**

10 Conclusions and Perspectives **103**
10.1 Conclusions . . . 103
10.2 Future Work . . . 104

Bibliography **120**

List of Figures

1.1 The trend of clock frequency and number of cores per chip in the past 50 years 3

2.1 A general workflow of polyhedral compilation 12
2.2 1D iterated stencil and its iteration space before and after scheduling 14
2.3 An illustrative diagram on different code generation algorithms 16
2.4 An illustrative loop nest and its iteration space in polyhedral compilation . . . 18
2.5 The schedule tree of the code in Figure 2.4(a) 22
2.6 A high level overview of the PENCIL compilation flow 24

3.1 Example with dynamic counted loops 29
3.2 Dependence graph of the example 32
3.3 Original schedule tree of the example 33

4.1 A sparse matrix computation and its normalized format 35
4.2 A `while` loop and its normalized format 36
4.3 Conditional abstraction 39
4.4 A correct schedule tree of the example 40
4.5 Inserting mark nodes in the schedule tree 42
4.6 Mark nodes with split band nodes 43
4.7 Replace each mark node with an extension node 43

5.1 Fusing two dynamic counted loops 48
5.2 Code generation with loop tiling for CPU 49
5.3 The schedule tree representation of code shown in Figure 5.2 50
5.4 An interchange example 51
5.5 Sinking the dynamic definition and its schedule tree representation 53
5.6 Code generation with loop tiling for GPU 54
5.7 The schedule tree representation of the code shown in Figure 5.6 55

6.1 Performance of change-making on GPU 58
6.2 Performance of the bucket sort on GPU 59
6.3 Performance of the HOG descriptor on GPU 60
6.4 Performance of `equake` on GPU 61
6.5 Performance of the CSR SpMV on GPU 63

6.6 Performance of the ELL SpMV on GPU 64
6.7 Performance of the HOG descriptor on CPU 65
6.8 Performance of `equake` on CPU 65
6.9 Performance of the CSR SpMV on CPU 66
6.10 Performance of the ELL SpMV on CPU 66

7.1 Comparison of different tile shapes 71
7.2 A simple image processing pipeline 72
7.3 Shaded regions of acute trapezoid tiling 72
7.4 Shaded regions of right trapezoid tiling 74

8.1 The original schedule tree of the code in Figure 7.2 78
8.2 Rectangular tiling regardless of the correctness 79
8.3 Acute trapezoid tiling 79
8.4 Rectangular tiling after shifting 80
8.5 Right trapezoid tiling 80
8.6 Schedule tree of acute trapezoid tiling 81
8.7 Schedule tree of right trapezoid tiling 81
8.8 Code generated by acute trapezoid tiling 84
8.9 Code generated by right trapezoid tiling 84
8.10 Schedule tree of multiple statements 85
8.11 Full and partial dimensional overlapped tiling 87

9.1 Performance of Bilateral Grid on CPU 91
9.2 Performance of Camera Pipeline on CPU 92
9.3 Performance of Harris Corner Detection on CPU 93
9.4 Performance of Local Laplacian Filter on CPU 93
9.5 Performance of Multiscale Interpolation on CPU 94
9.6 Performance of Pyramid Blending on CPU 95
9.7 Performance of Unsharp Mask on CPU 95
9.8 Performance of heat-1d stencil on CPU 97
9.9 Performance of heat-2d stencil on CPU 97
9.10 Performance of heat-3d stencil on CPU 97
9.11 Performance of the PolyMage benchmarks on GPU 98
9.12 Performance of the iterated stencils on GPU 99

List of Tables

1.1 The memory hierarchy of NVIDIA Tesla GPUs in the past five years 6
1.2 List of xPUs used for AI Accelerators 7

6.1 Summary of the input sparse matrices 62

9.1 Summary of the PolyMage benchmark 90
9.2 Problem sizes and tile sizes of the iterated stencils 96

Part I

Introduction & Background

1 Introduction

In the early stage of high-performance computing, increasing clock frequencies was the main source of performance gain. Since the breakdown of Dennard scaling a dozen years ago, most CPU manufacturers have been focusing on multicore processors as an alternative of raising clock frequencies from one generation to next. Figure 1.1 shows the semi-centennial trend of clock frequency and number of cores per chip.

Multicore processors are nowadays ubiquitous on almost all platforms, ranging from supercomputers ranked on the TOP500 list to personal laptops and mobile devices. In addition, their pervasiveness in the embedding computing domain of multimedia and image processing due to the recent process of neural networks also validates the dominance of multicore processors in all realms of computing.

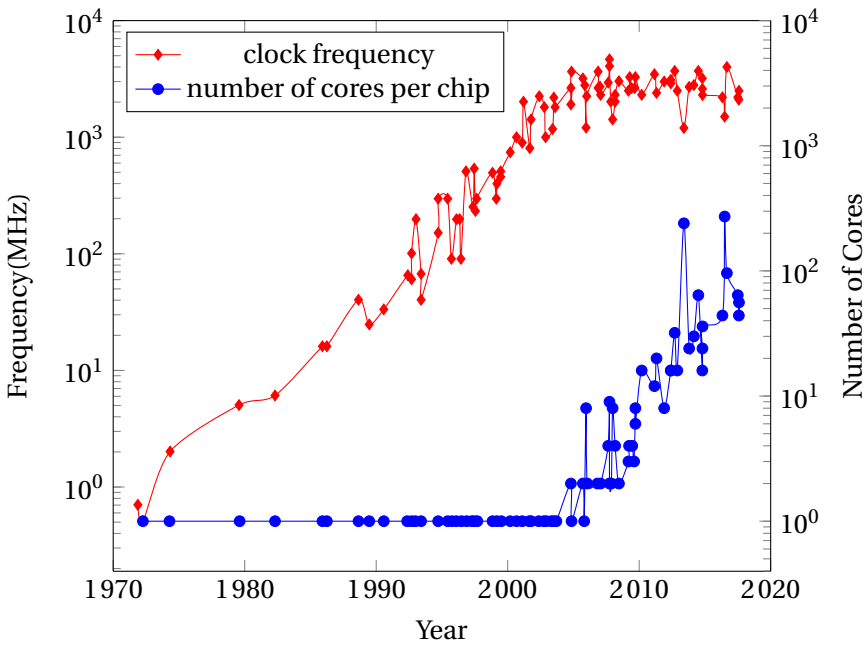


Figure 1.1 – The trend of clock frequency and number of cores per chip in the past 50 years

Apart from the resulting improvements in performance due to the introduction of multicore processors, one of the most challenging issues is the difficulty to effectively exploit parallelism on such devices. On the one hand, a multicore processor is allowed to implement multiprocessing freely by coupling the cores on the device either tightly or loosely, leading to a variety of memory hierarchies and resulting in the diversity in architectures. On the other hand, the evolution of parallel architectures also calls for the innovation of programming languages amenable to the memory hierarchy, giving rise to the design of both general-purpose and domain-specific parallel programming languages and complicating the programmability issue further.

Even though end users may be equipped with the knowledge of the high-level programming language of a platform after a long-term study or training, it is still a complex and error-prone task to deploy the code written by end users on the target architecture. An optimizing compiler is not only responsible for translating the code implemented by a high-level general-purpose/domain-specific language into a low-level executable program, but is also expected to automatically apply both high-level and low-level transformations, especially those performance-critical loop transformations, for exploiting parallelism and improving locality, thus releasing the burden of end users from taking the hardware information into consideration at the beginning of programming.

In the domain of scientific and engineering applications, a large number of computationally intensive applications spend most of the execution time on nested loops, making the polyhedral model [FL11] a very competitive and promising approach to solving the above problems. The polyhedral model is a powerful mathematical abstraction of loop nests, providing a way to reason about loop transformations by abstracting each iteration of a statement as an integer point in a "polyhedron" and mapping a multi-dimensional logical execution date [Fea92b] for defining its lexicographic execution order. As a role of bridging the gap between high-level programming interfaces and underlying hardware, the polyhedral model has made a great deal of progress in the past few decades, but it is now facing new emergent challenges brought by both modern architectures and programming languages.

1.1 From General-purpose Languages to Domain-specific Languages

Thanks to the significant advances in dependence analysis [Fea91, Pug91, VBCG06, BCVT13], schedule transformation [Fea92a, Fea92b, LL97, BHRS08, BAC16, UC13, ABC18] and code generation [AI91, Che, QRW00, Bas04, VBC06, GVC15], the polyhedral model has been brought to the front scene in automatic parallelization and locality optimization. There exist a large number of mature polyhedral compilation frameworks and loop optimizers, including both research projects [BHRS08, CCH08, VCJC+13] and commercial productions [TCE+10, GGL12, CSG+05, BGDR10, LLS06]. Such compilers usually take a general-purpose (intermediate) language as input and generate optimized high-level/low-level code amenable to the target architecture as demand. Despite that, the optimality of the code generated by such polyhedral

compilers still remains elusive, falling behind the performance of heavily hand-tuned codes written by an expert.

Part of the reason of performance gap between the generated codes of optimizing compilers and hand-written programs is due to the conservativeness that a compiler has to possess in nature as system software, reducing the optimization space of automatic transformations. Worse yet, the absence of the ability to reason about the domain knowledge about the implementation strategies from a piece of code also constraints such compilers, missing aggressive and/or global optimizations that can be performed by hand.

Domain-specific languages (DSLs) are proposed to obtain high performance and now very prevalent in many application domains. The polyhedral model was successfully integrated with DSLs, such as those for optimizing DSLs for graphical dataflow language [BB13, SSPS16], stencil computations [HVF+13], etc. Recently, due to the revolution in machine learning caused by the success of deep learning, the polyhedral community is also expected to resolve the problem of bridging neural network applications and high-performance hardware accelerators. A DSL may be a standalone language or more often embedded in a general-purpose language, like Halide [RKBA+13] in C++, TensorComprehensions [VZT+18] and TVM [CMJ+18] in Python[1], DeepDSL [ZHC17] in Scala, etc. A domain-specific compiler leverages specialized internal representations for expressing domain-specific knowledge, extending its optimization space by enabling such domain-specific high-level transformations. Representative DSL compilers for such applications include the TensorComprehension framework for automating the deployment of neural network applications on multicore platforms and the PolyMage compiler [MVB15, MAS+16] for Halide [RKBA+13], a DSL for writing high-performance image processing code.

While the polyhedral model eases the translation of both general-purpose languages and DSLs on modern architectures, it often suffers from scalability challenges to various input languages. Even though some internal representations like Hailde IR and PENCIL [BBC+15] were proposed as the solution to this problem, the polyhedral model still faces many painful problems due to its incompetence for dynamic control and non-affine applications.

1.2 Architecture Diversity

Generally speaking, a multicore system is supposed as homogeneous if the system includes only identical cores, or heterogeneous otherwise. The Pluto optimizer [BHRS08] provides a systematic, end-to-end way for automatic parallelization and locality optimization on homogeneous multicore systems, taking into consideration the memory hierarchy problem by automating simply/complex tile shapes [BBP17]. The emergence of Graphics Processing Units (GPUs) brought new challenges not found in homogeneous systems to the polyhedral model, calling for source-to-source polyhedral compilers capable of generating correct codes for both

[1] TensorComprehensions and TVM here are used to refer to the DSLs rather than the compiler stacks.

host processors and device accelerators, further complicating the code generation issue.

Unlike CPUs that can run efficiently when data is resident in Caches, GPUs have a variety of different kinds of processing units, leading to a more complicated memory hierarchy. For instance, Table 1.1 lists the memory hierarchy of fastest NVIDIA Tesla GPUs in the past five years.

Table 1.1 – The memory hierarchy of NVIDIA Tesla GPUs in the past five years

Tesla Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100
GPU	GK180 (Kepler)	GM200 (Maxwell)	GP100 (Pascal)	GV100 (Volta)
Memory Size	Up to 12 GB	Up to 24 GB	16 GB	16 GB
L2 Cache Size	1536 KB	3072 KB	4096 KB	6144 KB
Shared Memory Size	16/32/48 KB	96 KB	64 KB	96 KB*
Register File Size/SM	256 KB	256 KB	256 KB	256 KB
Register File Size/GPU	3840 KB	6144 KB	14336 KB	20480 Kb

* The shared memory size of GV100 (Volta) is confinable up to 96 KB.

PPCG [VCJC+13] is considered as one of the most successful polyhedral compilers for heterogeneous systems, exploiting parallelism and locality optimization as in traditional homogeneous systems but also automating the management of memory system on devices and communications between host and device. Like the diamond tiling technique in Pluto, a hybrid/hexagonal tiling approach [GCH+14] is also implemented in PPCG for further improving the performance of generated code. Some follow-up PPCG-based researches focus on parametric tiling [JGTC14] and the mapping and separation of multi-level parallelisms in the accelerators of a heterogeneous system [SHS17].

Besides shared memory strategy, message passing is also used as the inter-core communication method in distributed systems and heterogeneous systems, requiring the code generator of an optimizing compiler to express the explicit communication with libraries like Message Passing Interface (MPI). Polyhedral compilation frameworks targeting on minimizing communication volume [Bon13, RB14] or handling the mixture of regular/irregular loop nests [RDE+15] were proposed for such multicore systems.

Similarly, accelerators in heterogeneous systems are not restricted to GPUs. For example, configurable devices like Field-Programmable Gate Array (FPGA) [BRS07, PZSC13] can also be the target of an optimizing compiler, followed by some researches in high-level synthesis area [ZLC+13, WLC14]. These together with the above mentioned architectures are calling for a strict portability of the polyhedral model to multiple platforms. Recent work integrating multicore parallelism and Single Instruction Multiple Data (SIMD) vectorization [TNC+09, KVS+13] not only addressed the code generation issue but also implemented a different

scheduling strategy in a tile.

Even though the advances made by the polyhedral community on so many accelerators, there still exits a long way to achieving architectural portability. The emergence of artificial intelligence applications has brought new challenges for this issue. For instance, Table 1.2 summarizes the latest xPUs used for modern artificial intelligence accelerators.

Table 1.2 – List of xPUs used for AI Accelerators

Abbreviation	Full name	Manufacturer	Released year
APU	Accelerated Processing Unit	AMD	2011
BPU	Brain Processing Unit	Horizon Robotics	2017
DPU	Deep Learning Processing Unit	Deephi Tech	2016
	Dataflow Processing Unit	Wave Computing	2017
EPU	Emotion Processing Unit	Emoshape	2017
HPU	Holographic Processing Unit	Microsoft	2017
IPU	Intelligence Processing Unit	GraphCore	2017
	Intelligence Processing Unit	Mythic	2018
	Image Processing Unit	Google	2017
NPU	Neural Network Processing Unit	Vimicro	2016
SPU	Stream Processing Unit	AMD	2006
TPU	Tensor Processing Unit	Google	2016
VPU	Vision Processing Unit	Intel	2016
ZPU	Zylin CPU	Zylin AS	2015

1.3 Beyond Parallelization and Locality Optimization

On modern multicore processors, parallelism due to the increased core numbers on a single chip and locality caused by memory hierarchy are the two main objectives considered by compiler designers. As a result, optimizing compilers like a polyhedral optimizer are usually expected to be capable of automatic parallelization and locality optimization. Unfortunately, parallelization and locality optimization are sometimes contradictory with each other by putting conflict constraints on the objective function of scheduling algorithms, forcing them to make a tradeoff between parallelism and locality for achieving optimality of performance.

As a loop transformation aiming at improving locality while preserving the parallelism that has been exploited by a scheduling algorithm, loop tiling [IT88] has been long considered as foreign to optimizing compilers; even for the polyhedral model, it could not be easily expressed using an affine function a decade ago. Thanks to its recent advances, the polyhedral model has been proved to be promising in automating loop tiling. A cost-model-based scheduling

algorithm like the Pluto scheduler [BHRS08] or its variants put into practice the automation of simple tile shapes in the polyhedral model. A follow-up trend on the automatic tiling technique focuses on more complex tile shapes like diamond [BBP17] and hexagonal [GCH+14] working with arbitrary affine dependences, and overlapped and split shapes [KBB+07] restricted to constant dependence vectors.

Loop fusion [KA02] is another loop transformation to enhance locality and reduce synchronizations across multiple loop nests. There have also been successful advances on loop fusion in the polyhedral model [BGDR10, MLY14, JB18], providing a variety of fusion heuristics to modern optimizers.

Storage optimization is also a research direction of polyhedral compilation. Array contraction, for example, is a long considered automatic memory footprint optimization in the polyhedral world. The applicable domain is still constrained to special cases like stencil computations although researchers made a lot of efforts in this direction, including the universal-occupancy-vector-based [SCFS98], lattice-based [DSV05] and storage-hyperplane-based [BBC16] techniques, etc.

In spite of the exciting progresses made by the polyhedral community on automatic parallelization and locality optimization, there still exist a large number of opening issues awaiting supports and efforts. The latest research trend also tried to integrate the polyhedral model with dynamic/runtime techniques [KPP+15, SRC15, BKP+16, SPR17] for extending the scope of the tool, leaving much room for the extensions in this field.

1.4 Combining Languages and the Polyhedral Model

The polyhedral model so far is successful in so-called "static control parts" (SCoPs) where loop nests satisfy certain statically predictable restrictions. There is an increasingly emergent demand on its applicability to non-affine domains to cope with the complexity of modern multicore architectures. A notable direction among the open challenges is the incompetence of the polyhedral model to handle non-affine applications and transformations. Such non-affine applications usually involve dynamic data-dependent control flow and/or non-affine expressions that go beyond the scope of the polyhedral model, while non-affine transformations[2] are usually not expressible using existing techniques.

A representative polyhedral-based approach on non-affine applications are the work of handling `while` loops [BPCB10], along with a great deal of work with special focus on sparse matrix computations [SGO13, VSHS14, VHS15, SLC+16, VMP+16]. The former misses more aggressive optimizations when handling less expressive dynamic conditions than a general `while` loop, while an inspector/executor scheme is usually constrained to a subset of sparse matrix computations.

[2] An affine transformation should be "signle-valued", i.e., an one-to-one mapping function of the integer points on iteration space.

With regard to non-affine transformations, overlapped tiling [KBB+07] is a representative technique gaining much attentions recently due to its compatibility with other optimizations like fusion, scratchpad memory allocation, etc. when optimizing image processing pipelines. Unlike a standard tile shape exploited by current polyhedral compilers, additional overlapped regions are introduced for exploiting inter-tile parallelism in such transformation by jointing consecutive tiles. Unfortunately, no implementations of overlapped tiling in a general-purpose polyhedral framework have been reported except PolyMage [MVB15] , a DSL compiler for image processing pipelines.

This dissertation describes a combined language and polyhedral approach to extend the application domain of the polyhedral model in non-affine applications and express non-affine transformations in the model. On the one hand, we study the parallelizing compilation and loop nest optimization of an important class of programs where counted loops have a dynamic data-dependent upper bound. Such loops are amenable to a wider set of transformations than general `while` loops with inductively defined termination conditions: for example, the substitution of closed forms for induction variables remains applicable, removing the loop-carried data dependences induced by termination conditions; such loops can also be viewed a generalization of sparse matrix computations using compressed data layout stores nonzero elements only as the latter can be easily generalized by subtracting the lower bound from the upper bound.

On the other hand, we revisit overlapped tiling in polyhedral compilation and demonstrate how to derive tighter tile shapes with less redundant computations, by enabling overlapped tiles based on a well-defined general-purpose intermediate representation. It releases the overlapped tiling in polyhedral model from being restricted to a domain-specific language while not introducing sophisticated rescheduling and custom code generation in a polyhedral framework.

Given the diversity of multicore architectures and the difficulty of programming on these platforms, a polyhedral compilation approach has become a compelling alternative for writing parallel code on these targets. Our approach is driven by combining an intermediate language and the polyhedral model, not only removing the conservativeness caused by using a general-purpose language hindered by the difficulty of static analysis but also avoiding the implementation of a DSL compiler for the portability to different architectures. By coupling with such an intermediate language, one may define coding rules predominantly related to restricting the non-statically predictable manners, allowing for better optimizations when translating such programs into the code on target machines using a polyhedral framework. More importantly, leveraging such an intermediate language also eases the code generation for different architectures, making the portability issue a straightforward task.

Our method on counted loops with a dynamic data-dependent upper bound goes beyond the state of the art in fully automating the process, specializing the code generation algorithm to the case of dynamic counted loops and avoiding the introduction of spurious loop-carried

dependences. The experimental results on representative irregular computations ranging from dynamic programming, computer vision and finite element methods to sparse matrix linear algebra validate that the method is applicable to general affine transformations for locality optimization, vectorization and parallelization.

Our algorithm to generalize overlapped tiling allows for tighter overlapped tile shapes than the state of the art, further improving the performance of image pipelines on both general-purpose multicores and heterogeneous accelerators by integrating with transformations including alignment and scaling of stages in the pipeline, loop fusion, scratchpad allocation, hybrid tiling, etc. The experimental evaluation on the PolyMage benchmarks and representative iterated stencils validates the effectiveness and general applicability of our technique on both general-purpose multicores and accelerators.

The organization of the dissertation is as follows. In the first part, we described in this chapter an introduction to the problem we aim at in this dissertation, followed by Chapter 2 providing the technical background on the polyhedral model and the intermediate language used in our approach. The second part presents our method to handle non-affine application, i.e., parallelization and optimization of counted loops with a dynamic data-dependent upper bound, including the motivation and related work in Chapter 3, scheduling algorithm in Chapter 4, code generation method in Chapter 5 and experimental results in Chapter 6. Generalizing overlapped tiling in the polyhedral model regarding the non-affine transformations is introduced in the third part, comprising Chapter 7 describing the motivation and related work, Chapter 8 explaining the polyhedral implementation of the method and Chapter 9 evaluating the proposed technique on both homogeneous and heterogeneous architectures. We finally conclude the dissertation in the last part, Chapter 10, by summarizing the topics studied in the dissertation and discussing directions for further research.

2 Background

After a thirty-year evolution, the polyhedral model has become a powerful optimizer in the domain of automatic parallelization and optimization. There have been a great number of open-source and/or commercial implementations of polyhedral compilation in both research and industry worlds. Comparing with unimodular matrices [Ban93, WL91] used in parallelizing compilers, the polyhedral model is equipped with (1) wider range of applications due to the capability to transform imperfect loop nests, (2) more powerful expressiveness by modeling almost each kind of loop transformations and (3) greater optimization space by compositing more transformations at one time.

As a consequence, polyhedral compilation nowadays is gradually becoming the state of the art of almost each domain of parallelizing compilers. In this chapter, we would first introduce the background of polyhedral compilation for a better understanding of the underlying principle of the polyhedral model. To cope with polyhedral compilation for non-affine applications and transformations, we would next present the intermediate language used in the dissertation.

2.1 Polyhedral Compilation

As we introduced in the previous section, the polyhedral community so far has made a great deal of progress in all realms of computing. Nonetheless, polyhedral compilation is long considered as too abstract for those people outside the polyhedral world. Part of the reason is due to the painfully theoretical descriptions in existing polyhedral publications; more importantly, the underlying principle of the polyhedral model involves a variety of concepts from linear algebra, static analysis, etc., making the use of the tools elusive for end users.

To make it easier to understand the polyhedral model, we introduce the background of polyhedral compilation in this section. One may refer to [FL11] for a much detailed description on fundamental concepts and definitions. In general, we would first give an overview of modern polyhedral compilation and then explain how programs are represented in the model. Next, the transformations that can be modeled in the polyhedral model are presented by comparing

with conventional methods used in parallelizing compilers, followed by an introduction to a well-defined representation of the polyhedral model.

2.1.1 An Overview

The pioneer work of the polyhedral model is the contribution of Karp et al. on systems of uniform recurrence equations [KMW67]. With the development of polyhedral compilation, integer polyhedra [LW97] and Presburger relations [PW94a, PW94c] were also introduced into the model for better expressiveness and flexibility purpose. Polyhedral compilation can either be integrated as a building block into a general-purpose compiler, e.g., Polly in LLVM, or serve as a standalone source-to-source translator like Pluto and PPCG. Such flexibility and compatibility help itself construct a well-defined compilation workflow and extend its application domain to a great extent.

To date, polyhedral compilation has reached maturity and emerged into a fully fledged workflow. Figure 2.1 shows a general compilation workflow of modern polyhedral compilers. Polyhedral compilation can either take as input a high-level programming language when serving as a standalone optimizer or an intermediate representation when embedded in a general-purpose compiler. With regard to generated code, a high-level language wrapping parallel programming APIs executable on target platforms can be generated; otherwise, the optimized intermediate representation of the host compiler would be returned, possibly followed by other transformation passes of the host compiler.

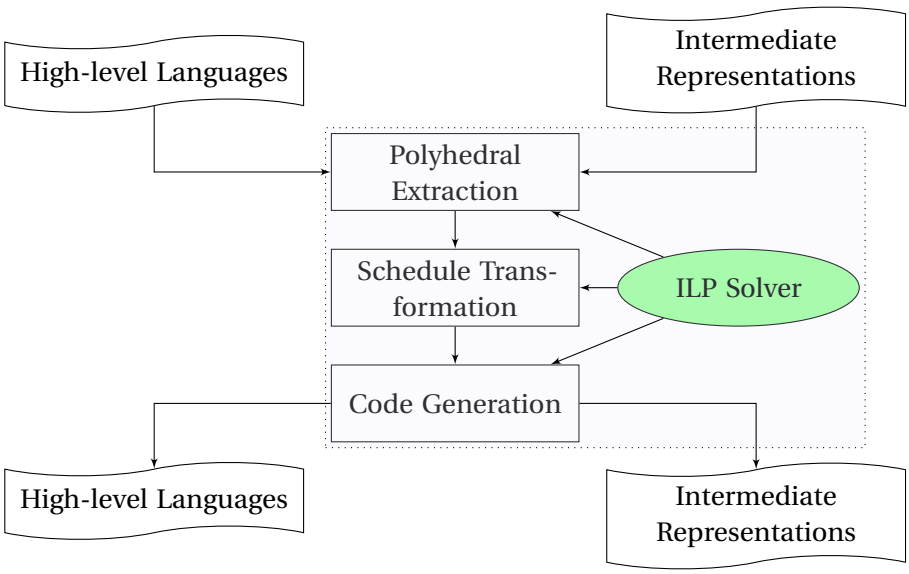


Figure 2.1 – A general workflow of polyhedral compilation

Typically, polyhedral compilation consists of three steps, as the three modules enclosed by the dotted frame in Figure 2.1 show.

Dependence analysis. First, polyhedral extraction is serving as the frontend, parsing the code fragment of input languages and extracting polyhedral representations with regard to statement instances and array elements for the program. The duty of a parser is checking whether the input is a "SCoP", meaning the code fragment is statically predictable and polyhedral compilation would represent the input program with finite internal representations if true. pet [VG12] and clan [cla] are two representative, practical parsers for polyhedral compilation, generally used in a variety of mature polyhedral compilers. Such representations of statement instances and their relationships with the array elements they access are used to compute dependence relations by solving an integer linear programming (ILP) problem.

Polyhedral compilation differentiates dependence analysis from conventional methods by refining the analysis from statement-wise to instance-wise [Fea91]. Dependence relations can be further separated into value-based dependences, the result of data flow analysis [MAL93, Mas94], used for preserving the semantic of programs, and memory-based dependences, studied for the purpose of improving data reuse [PW92, VBCG06, BCVT13].

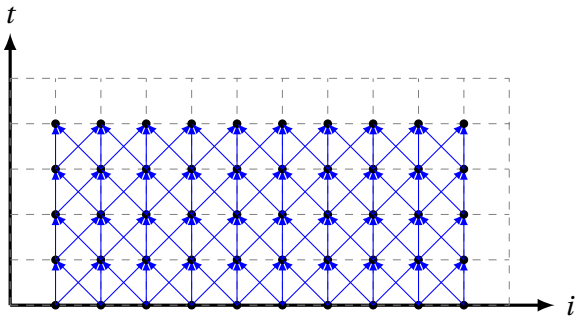
Schedule transformation. Secondly, schedule transformation is the core of polyhedral compilation, producing a new schedule by taking into consideration target architectures. In other words, schedule transformation is the process of mapping a new logical execution date for each integer point in a polyhedron, accomplished by invoking the underlying ILP solver. The process of schedule transformation could also be considered as a composition of different loop transformations with the purpose to fully exploit parallelism and data locality.

We take the 1D iterated stencil shown in Figure 2.2(a) as an illustrative example. Iterated stencils are a class of computations updating an array element using its neighbors, commonly found in computational fluid dynamics, image processing, partial differential equations, etc. The original iteration space of the 1D stencil code is shown in Figure 2.2(b), indicating the computation proceed first along t axis and then i axis. Instead, the transformed iteration space after schedule transformation in Figure 2.2(c) implies the computation should first follow t axis and then $t+i$ axis. The instances of the statement are represented by integer points in iteration space, coordinated with each other by a blue arrow denoting a dependence relation.

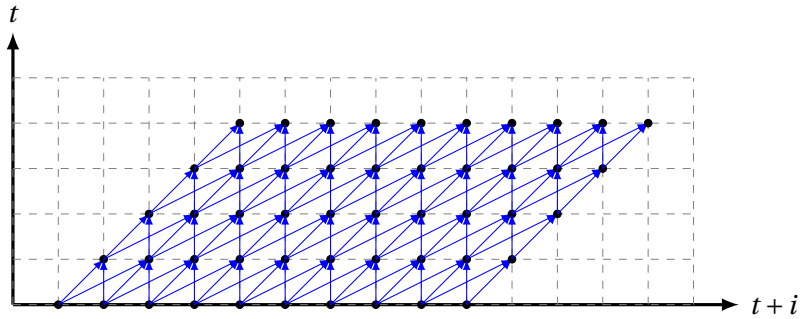
Tiling along t axis and i axis in Figure 2.2(b) is illegal since such tiling may produce dependence cycles between tiles along i axis, prohibiting the data locality along t axis of the original iteration space. On the contrary, one may benefit from the data locality along both axes on the transformed iteration space as tiling along the axes may not result in dependence conflicts. In fact, schedule transformation could be understood as the reconstruction of the basis of iteration space, attained by a scheduling algorithm like [Fea92a, Fea92b, LL97, BHRS08, BAC16, UC13, ABC18] and their variants in libraries. While the scheduling algorithm proposed by Feautrier [Fea92a, Fea92b] was complained due to the missing of considering communication

```
for (t=0; t<T; t++)
    for (i=1; i<N-1; i++)
        A[t+1][i]=0.25*(A[t][i+1]+2.0*A[t][i]+A[t][i-1]);
```

(a) $1d$ iterated stencil



(b) Original iteration space of the code in Figure 2.2(a)



(c) Transformed iteration space of the code in Figure 2.2(a)

Figure 2.2 – 1D iterated stencil and its iteration space before and after scheduling

overhead, the hyperplane[1] partitioning technique [LL97] also failed to minimize the order of synchronization even though it takes into account communications. The cost-model-based scheduler [BHRS08] developed in the Pluto compiler was designed to overcome such flaws and has been demonstrated as effective in practice by a variety of implementations.

Schedule transformation is considered as the most difficult component of polyhedral compilation. A scheduling algorithm is tightly coupled with dependence relations produced by the frontend: while it should preserve the semantic of the program by being constrained to dependence relations, it is also expected to minimize dependence distances for reuse purposes, thereby improving data locality. A scheduling algorithm is not only responsible for exploiting different compositions of loop transformations, e.g., the composition of loop tiling and skewing could be triggered in the example of Figure 2.2, but also obligated to exploit both fine-grained and coarse-grained parallelisms. Moreover, a scheduling algorithm should

[1] A hyperplane is the projection of an n dimensional space on its $n-1$ dimensional sub-space.

also take into consideration the requirements from follow-up code generation: for example, schedule transformation should allow the insert of thread-/warp[2]-level synchronizations when generating CUDA code on GPUs.

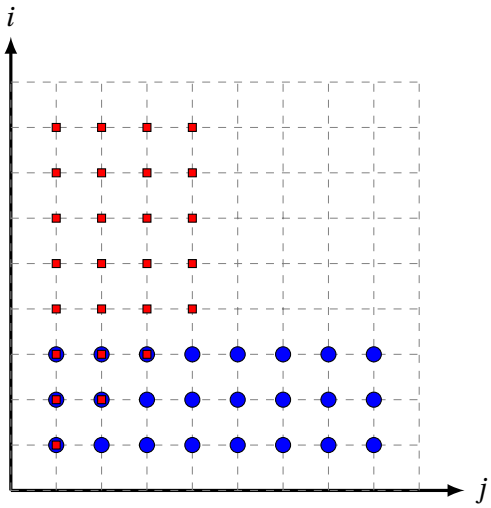
Code generation. Finally, code generation constitutes the backend of polyhedral compilation, made up by two phases with one building abstract syntax trees (ASTs) from the results of polyhedral extraction and schedule transformation while the other generating expected high-performance code executable on target platforms. As code generation would not change the execution order of programs, it is allowed to not take into consideration dependence relations, scanning the iteration space and generating code according to execution dates defined by the transformed schedule. As a consequence, code generation is also referred to as polyhedral scanning.

When building ASTs from the results of previous steps, a code generator manages to determine loop bounds and conditionals of control flow by seeking solutions for an optimization problem subject to integers. The generated ASTs could then be passed to emit instructions amenable to different programming models on target machines, facilitating the portability to different architectures.

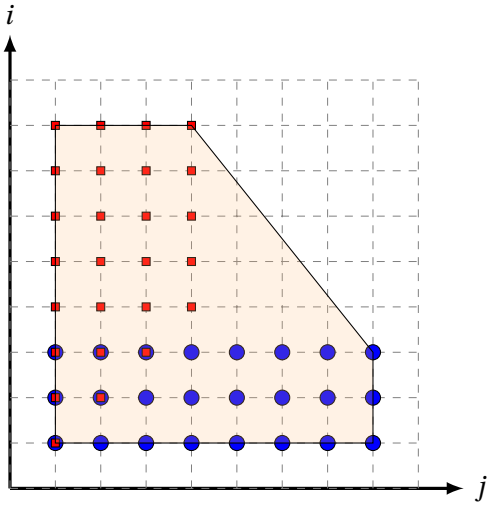
One representative implementation of code generation in polyhedral compilation is the convex-based algorithm [AI91], generating code by first constructing a convex for all polyhedra in the iteration space. Figure 2.3(a) shows an illustrative iteration space composed of two polyhedra with one comprising all red square points and the other made of blue circle points, followed by a diagram of convex-based algorithm in Figure 2.3(b). One of the flaws of this algorithm is the generated code may include multiple nested if conditionals governing the correct execution of each polyhedron in the iteration space, promoting some code generators like Codegen+ working on hoisting if conditionals [Che]. The other code generation technique was proposed by Quilleré et al. [QRW00] and implemented in the CLooG generator and its variants [Bas04, VBC06, GVC15]. Unlike the convex-based algorithm, the method used in CLooG may first split the polyhedra into distinct regions as shown in Figure 2.3(c), producing code by scanning each of such regions individually.

As one may also find in Figure 2.1, an ILP solver is at the core of polyhedral compilation, providing each step with minimal flexible integer solutions, thereby achieving the manipulation of polyhedral transformations. There exist a variety of libraries for solving ILP, including `isl` [Ver10], Omega [KMP+96], PIP [Fea88], PolyLib [Loe99] and PPL [BHZ08], etc., differing each other by using different algorithms and data structures.

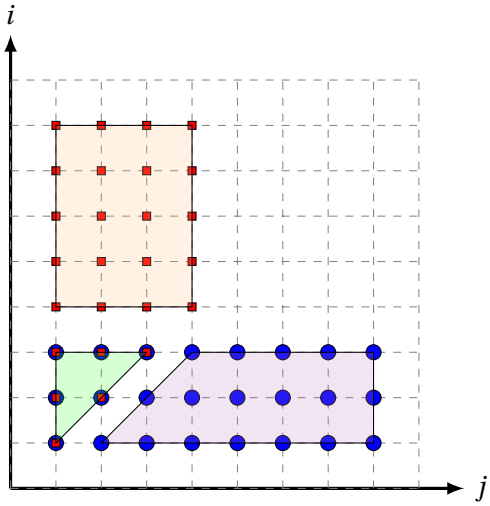
[2] A warp is a set of threads arranged lengthwise on a loom and crossed by the woof.



(a) An illustrative iteration space



(b) A convex-based implementation of the iteration space shown in Figure 2.3(a)



(c) A splitting-based implementation of the iteration space shown in Figure 2.3(a)

Figure 2.3 – An illustrative diagram on different code generation algorithms

2.1.2 Polyhedral Representations

Intuitively, polyhedral compilation models one statement in loop nests as a polyhedron and each instance of the statement as an integer point. One may understand those transformations enabled by polyhedral compilation as the process of reshaping such polyhedra. In terms of implementing the polyhedral model in a parallelizing compiler, however, one may have to resort to some polyhedral representations for both manipulation and optimization purposes. Historically, there have been various compositions of different representations used in polyhedral compilation, but we would like to introduce the following representations used in the dissertation as they are sufficient to model polyhedra and have been implemented in

some widely used optimizers like LLVM/Polly, PPCG, etc. One is without doubt free to choose any other combinations of representations for their implementation, as such representations could always be mutually transformed.

Before the introduction to polyhedral representations, we prefer to first present some mathematical concepts. This is because such mathematical concepts are the underlying expressions in integer manipulation libraries. More importantly, it would be a painful task to explain polyhedral representations if bypassing such mathematical concepts. If the readers are interested in a more detailed description or some more concentrated examples on such mathematical descriptions, we would suggest to refer to the work of Grosser [Gro14].

Integer sets. An integer set is a set of n-tuple integers subject to a group of affine constraints relating such n-tuple integers with m-tuple constant parameters, with n representing the dimensionality of the set, m the dimensionality of the parameters in constraints. Mathematically, an integer set can be written as

$$S = \{(i_1, i_2, ..., i_n) : f((i_1, i_2, ..., i_n), (p_1, p_2, ..., p_m))\} \tag{2.1}$$

An integer set is called named integer set when assigning a name to the integer tuple. In practice, one may think an n-tuple integer set as the collection of loop iterators, m-tuple constant parameters the parameters of programs. A constraint function usually comes as the conjunction of multiple inequalities.

Integer maps. An integer map is a binary relation mapping an n_1-dimensional integer set, i.e., the domain of the map, to another n_2-dimensional integer set, i.e., the range of the map, subject to a set of affine constraints on the integer sets and constant parameters. An integer map can be generalized as

$$M = \{(i_1, i_2, ..., i_{n_1}) \rightarrow (i_1, i_2, ..., i_{n_2}) : f((i_1, i_2, ..., i_{n_1}), (i_1, i_2, ..., i_{n_2}), (p_1, p_2, ..., p_m))\} \tag{2.2}$$

An integer map can be interpreted as the relation between statement instances with their accessed data locations or dependence relations. In the same way, a named integer map represents an integer map between two named integer tuples.

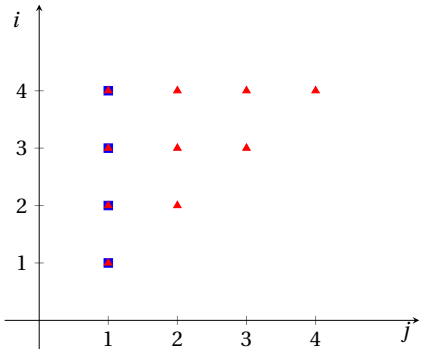
Named union sets and Named union maps. We use named union sets to refer to the union of different named integer sets, and named union maps for the union of different named integer maps. Named union sets can be used to express all statement instances of a SCoP,

while named union maps may either define execution order on statement instances or relate statements and the data they access.

Given the above mathematical concepts, we can explain polyhedral representations in a much easier way. Following the previous subsection, we still use an illustrative example for explanation. One may obtain the iteration space of the loop nest listed in Figure 2.4(a) as shown in Figure 2.4(b), with one polyhedron for statement $S1$, composed of blue points, and the other for statement $S2$, depicted with the collection of all red points. f and g are affine functions of their indices. In our work, we would use the following representations.

```
   for (i=1; i<=4; i++) {
S1:    a[i] = f(i);
       for (j=1; j<=i; j++)
S2:        b[i][j]=g(a[i]);
   }
```

(a) An illustrative loop nest



(b) The iteration space of the loop nest shown in Figure 2.4(a)

Figure 2.4 – An illustrative loop nest and its iteration space in polyhedral compilation

Iteration Domain. Iteration domain is the collection of all statement instances, represented using a named union set with each named component covering all instances of one statement followed by a set of inequalities for bounds. The iteration domain of the code above can be expressed with (2.3).

$$Domain = \{S_1(i) : 1 \le i \le 4; S_2(i,j) : 1 \le i \le 4 \wedge 1 \le j \le i\} \tag{2.3}$$

Access Relations. Access relations are a set of relations coordinating statement instances with the data locations they access, modeled by a set of named union maps together with some inequalities for bounds. (2.4) describes the access relations of the example, consisting of a Write relation and a Read relation. By refining access relations with read and write relations, polyhedral compilation is free to compute dependence relations easily. Furthermore, a write relation can also be split into may-write and must-write relations for the purpose of aggressive optimizations.

$$\begin{aligned} Write &= \{S_1(i) \rightarrow a(i) : 1 \leq i \leq 4; S_2(i,j) \rightarrow b(i,j) : 1 \leq i \leq 4 \wedge 1 \leq j \leq i\} \\ Read &= \{S_2(i,j) \rightarrow a(i) : 1 \leq i \leq 4 \wedge 1 \leq j \leq i\} \end{aligned} \tag{2.4}$$

Schedule. As we already mentioned in previous context, polyhedral compilation would map a multi-dimensional logical execution date to each statement instance. Schedule is such a multi-dimensional execution date assigned to a statement instance, expressed using a binary relation between two different multi-dimensional integer tuples. A lexicographically smaller schedule implies an earlier execution of the statement instance. An original schedule is the execution date assigned to a statement instance before schedule transformation. For example, the original schedule of the code in Figure 2.4(a) can be written as (2.5). A new schedule would be computed after schedule transformation if the scheduling algorithm may find a better execution date with respect to parallelism and data locality.

$$Schedule = \{S_1(i) \rightarrow (i,0); S_2(i,j) \rightarrow (i,1,j)\} \tag{2.5}$$

Dependences. Dependences represent the access conflicts between statement instances, written as named union maps and used for guaranteeing the execution date of a producer be lexicographically smaller than that of the consumer, therefore enforcing the correctness of any transformations enabled by polyhedral compilation. A refinement from traditional statement-level dependences to instance-level dependences in the polyhedral model makes the expression of dependences more complicated, like the constraints after the named maps shown in (2.6) indicating the dependences are described with regard to statement instances.

$$Dependence = \{S_1(i) \rightarrow S_2(i,j) : 1 \leq i \leq 4 \wedge 1 \leq j \leq i\} \tag{2.6}$$

2.1.3 Loop Transformations

Given the polyhedral representations, one may apply any loop transformations and/or their compositions. Loop transformations could be attained by schedule transformations, i.e., reordering the statement instances. Considering the example shown in Figure 2.2, schedule transformation triggers loop skewing by specifying a new execution date to the statement instances of 1D iterated stencil.

If we use the polyhedral representations introduced in the last subsection to express the transformation, we may obtain the original schedule as $\{S_1(t,i) \rightarrow (t,i)\}$ and $\{S_1(t,i) \rightarrow (t, t+i)\}$ for the new schedule after applying loop skewing. As a result, a transformation could be

expressed as $(t, i) \rightarrow (t, t+i)$ for representing loop skewing. Suppose t and i as variables, the underlying principle of schedule transformation could be interpreted as seeking a coefficient matrix and a constant vector such that

$$\begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix} \begin{bmatrix} t \\ i \end{bmatrix} + \begin{bmatrix} c_{10} \\ c_{20} \end{bmatrix} = \begin{bmatrix} t \\ t+i \end{bmatrix} \tag{2.7}$$

It is straightforward to solve the above system of linear equations by hand as the problem is heavily simplified for the sake of illustration. However, polyhedral compilation may have to solve it automatically by resorting to an ILP solver; worse yet, the practical problems faced by the polyhedral model would be much more complicated. We would not go further into the underlying structure of schedule transformation but invent the readers to refer to the work of Bondhugula [Bon08] for a detailed mathematical explanation.

To generalize, schedule transformation can be understood as solving a coefficient matrix and a constant vector such that a transformation between two integer tuples can be accomplished. Each row of the coefficient matrix can be interpreted as a hyperplane. Note that the two integer tuples could differ with regard to dimensionality: for example, a scalar dimension could be introduced to achieve loop fusion. Besides, a more complex example of loop transformation would be loop tiling, increasing the dimension of the input tuple by doubling those components requiring tiling.

Schedule transformation by manipulating integer sets broadens the optimization space of polyhedral compilation and simplifies the composition of different loop transformations compared with traditional compilation models like unimodular matrices [Ban93, WL91], while the latter applies loop transformations by means of elementary matrix operations. Besides, the loop transformations covered by unimodular matrices are also very restricted, including loop interchange, skewing and reversal; the polyhedral model is rather capable for automating a wider set of loop transformations, widening the optimization space by enabling loop fission, fusion, index set splitting [GFL00], peeling, strip-mining [KP95], tiling, unroll and jam [Bon08, BF03], unrolling, unswitching, etc.[3] A recent work [YGK+13] also makes it possible to model algorithmic changes which could not be achieved by other techniques, further enriching the transformations of polyhedral compilation.

2.1.4 Schedule Trees

Apart from index set splitting, all the loop transformations modeled by polyhedral compilation could be facilitated by operating on the schedule representation, i.e., named union maps.

[3] Some of these loop transformations, i.e., loop peeling, unrolling, unswitching, are achieved by code generation rather than schedule transformation, since these transformations change the loop structure rather than reorder statement instances.

This, however, does not mean the operations on named union maps would be ease of use. A typical drawback of such method can be found when comparing the lexicographic order of two integer sets, since such operations can only be applied on integer sets with the same dimension.

More importantly, the above mentioned schedule representation could not be easily extended to handle non-affine transformations as named union maps hold the "injectivity" and "single-valuedness" properties. "injectivity" indicates a schedule representation allows different statement instances to be assigned the same logical execution date for expressing inner parallelism; "single-valuedness" refers to a schedule representation would only assign a single execution date to a statement instance, preventing the statement instance from being executed more than once. Clearly, the latter would not allow the implementation of non-affine transformations like overlapped tiling requiring multiple executions of a statement instance.

In this dissertation, we rely on a well-defined schedule representation that would make the expression of non-affine transformations possible in polyhedral compilation. As the schedule construction may decompose a dependence graph recursively and compute a partial schedule for each component independently, a schedule representation would naturally have the form of a tree [GVC15]. The schedule representation is thus called "schedule tree".

There have been some schedule representations proposed in the past, including the Kelly's abstraction [KPR95], "2d+1"-schedules [GVB+06], etc., that can be viewed as an encoding of schedule trees. Like named union maps, such encoding methods are usually restricted due to missing the ability to facilitate non-affine applications and transformations.

To give an intuitive impression on schedule trees, we depict the schedule tree representation of the code shown in Figure 2.4(a) in Figure 2.5. A schedule tree is constructed by recursively building partial schedule trees which in turn constructed by schedule nodes. For example, the schedule tree in the figure is composed of two sub-trees rooted at one filter node representing statement $S_1(i)$ and the other for statement $S_2(i, j)$. A partial schedule tree comprises one or more schedule nodes for expressing different semantics. We would next introduce the basic node types in schedule trees. For a complete description of schedule trees and nodes, please refer to [GVC15].

Domain. A domain node in schedule trees is a named union set, appearing as the root of a schedule tree and covering the collection of all statement instances that should be scheduled by the schedule tree. For the sake of simplicity, we would sometimes represent a domain node as "domain" like what we have done in Figure 2.5 rather than writing in the form like (2.3).

Context. A context node is used to introduce constraints on symbolic constants of the schedule tree. Symbolic constants introduced by a context node could serve as parameters of programs, usually omitted when they are only referenced by the domain node. In practice, parameters passed to compilers like tiling sizes are usually introduced in a context node for determining bounds of new schedules.

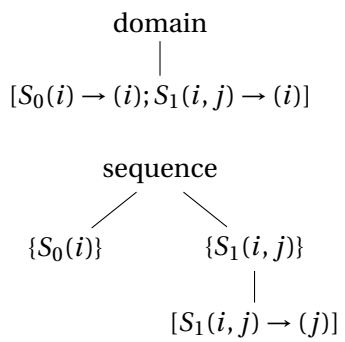


Figure 2.5 – The schedule tree of the code in Figure 2.4(a)

Filter. A filter node can be the root of a partial schedule tree, expressed with a named union set and representing the statement instances to be scheduled by its descendants. The partial schedule tree rooted at a filter node has to be retained to another rooted at a domain node, constructing the final schedule tree recursively. A filter node is guarded by a pair of braces.

Band. A band node is used to express the partial schedule on its parent node, written as a named union map. A band node in schedule trees is also integrated with more operations governing code generation, allowing for flexibility for more complicated cases. Named union maps in a band node could be piecewise quasi-affine for expressing schedules like tiling. We use square brackets to denote band nodes in schedule trees.

Sequence/Set. A sequence/set node always appears as the parent node of a group of filter nodes, forcing the children to be executed in a given/arbitrary order. An explicit support for a sequence/set representation makes it possible to break up the instances of a statement into separated parts.

Mark. A mark node can introduce any kinds of information to schedule trees. The use of mark nodes provides a great compatibility to schedule trees with other intermediate representations. For example, one may use a mark node to attain the information about representation mismatching, informing the follow-up code generator to handle this mark node with a custom implementation.

We introduce the basic node types for schedule trees here because we believe they are sufficient to understand the principle of schedule tree representation. Some other node types would be introduced in the following context, together with their uses in our work. The readers may also find more examples of schedule trees throughout the thesis.

2.2 A Platform-Neutral Compute Intermediate Language

In this subsection, we would introduce a platform-neutral compute intermediate language, PENCIL [BBC+15], that we rely on to facilitate non-affine extensions. Combining languages

with polyhedral approaches have been proved effective in many applications as we introduced in Section 1. We choose PENCIL as an intermediate representation in addition to polyhedral representations by taking into consideration the following properties.

PENCIL provides a sequential semantic in accordance with the philosophy of widely used programming languages by hiding target-specific hardware information, allowing programmers to follow existing way of programming with such languages by only adding lightweight annotations. This allows for the compatibility of integrating with both general-purpose and domain-specific languages.

PENCIL simplifies the analysis of non-affine expressions and eases the implementation of such extensions. With PENCIL, arrays must be declared through the C99 variable-length array syntax [ISO99]. The C99 type qualifiers/keywords `static const restrict` or a macro `pencil_attributes` expending to these type qualifiers/keywords must be used to declare the array function arguments. This allows the polyhedral model know about the length of arrays, and that arrays do not overlap during optimizations.

Pointer declarations and definitions are allowed in PENCIL, but pointer manipulation (including arithmetic) is forbidden except that C99 array references are allowed as arguments in function calls. Pointer dereferencing is neither allowed except for accessing C99 arrays. The restricted use of pointers can essentially eliminate aliasing problems for moving data between different address spaces of hardware accelerators.

A PENCIL `for` loop must have a single iterator, an invariant start value, an invariant stop value and a constant increment (step). Invariant here requires the value must not change in the loop body. To some extent, such structured `for` loops may simplify the polyhedral transformations. Considering the fact that recursive calls are not supported by accelerator programming languages like CUDA and OpenCL, recursive calls are excluded from PENCIL. However, we are allowed to extend the semantic of PENCIL for such extensions as long as they are needed.

As shown in Figure 2.6 is the high level overview of PENCIL compilation flow. PENCIL can be the target of a domain-specific compiler, followed by a polyhedral framework, therefore delivering information between a domain-specific language and polyhedral compilation. A typical representative application in DSLs of PENCIL is its use in the early stage of TensorComprehensions [VZT+18][4]. One is also allowed to write a general-purpose language with PENCIL specifications to model non-affine extensions, extending the polyhedral approaches to handle more complex cases. For example, a combined polyhedral technique with PENCIL was used to handle user-define reductions [RKC16] which would not be possible without PENCIL.

With regard to the code generation of the PENCIL compilation flow, there have already been

[4] PENCIL was introduced for bridging the Halide IR and polyhedral representations in the prototype implementation but was later removed from the framework due to simplification considerations. However, it helps the developers construct the early prototype implementations and lays a solid foundation for follow-up development of the framework.

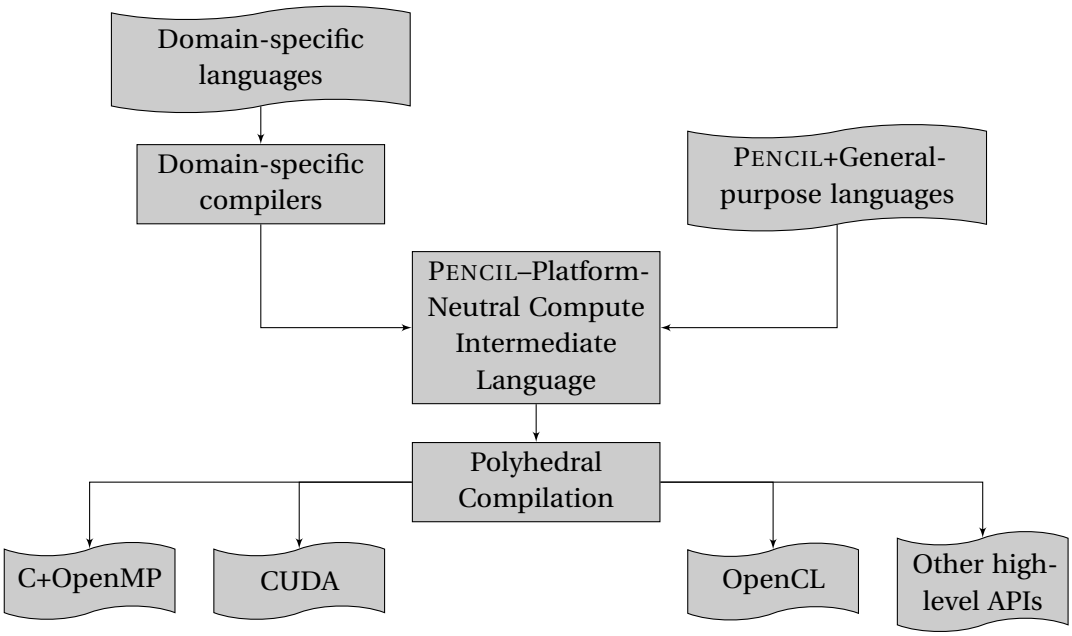


Figure 2.6 – A high level overview of the PENCIL compilation flow

a variety of backends supporting the generation of different parallel programming models, including OpenMP directives and CUDA, OpenCL APIs, etc. Retargeting the compilation flow to support other kinds of high-level APIs is straightforward thanks to the introduction of generating ASTs in polyhedral compilation, providing a great portability to a large number of modern multicore platforms.

Part II

Handling Non-affine Applications

3 Dynamic Counted Loops

While a large number of computationally intensive applications spend most of their time in static control loop nests—with affine conditional expressions and array subscripts, several important algorithms do not meet such statically predictable requirements, going beyond the scope of polyhedral compilation. A well-known non-affine extension to remove the limitation of the polyhedral model in this direction is the work of Benabderrahmane et al. [BPCB10] for handling general `while` loops. We are interested in the class of computational kernels involving *dynamic counted loops*. These are regular counted loops with numerical constant strides, iterating until a dynamically computed, data-dependent upper bound. Such bounds are loop invariants, but often recomputed in the immediate vicinity of the loop they control; for example, their definition may take place in the immediately enclosing loop.

Dynamic counted loops play an important role in numerical solvers, media processing applications, and data analytics, as we will see in the experimental evaluation. They can be seen as a special case of `while` loop that does not involve an arbitrary, inductively defined termination condition. The ability to substitute their counter with a closed form—an affine induction variable—makes them amenable to a wider set of transformations than `while` loops. Dynamic counted loops are commonly found in sparse matrix computations, but not restricted to this class of algorithms. They are also found together with statically unpredictable, non-affine array subscripts.

The purpose of this part is to further extend the ability of polyhedral compilation for handling such non-affine applications by enabling a wider set of loop transformations. We will first present the background along this research direction and then introduce our solution in the next two chapters, followed by some experimental results and discussions.

3.1 Background and Motivation

The polyhedral framework of compilation unifies a wide variety of loop and array transformations using affine (linear) transformations. The availability of a general-purpose method to

generate imperative code after the application of such affine transformations [QRW00, Bas04, GVC15] brought polyhedral compilers to the front scene, in the well-behaved case of static control loops.

3.1.1 Limitations of Previous Work

While significant amount of work targeted the affine transformation and parallelization of `while` loops [GL94, Col94, GC95, Col95, GGL98, GGL99, BPCB10, JCD+14], these techniques face a painful problem: the lack of a robust method to generate imperative code from the polyhedral representation. One representative approach to model `while` loops in a polyhedral framework, and in the code generator in particular, is the work of Benabderrahmane et al. [BPCB10].

This work uses over-approximations to translate a `while` loop into a static control loop iterating from 0 to infinity that can be represented and optimized in the polyhedral model. It introduces exit predicates and the associated data dependences to preserve the computation of the original termination condition, and to enforce the proper termination of the generated loops the first time this condition holds. These data dependences severely restrict the application of loop transformations involving a `while` loop, since reordering of the iterations of the latter is not permitted, and loop interchange is also restricted.

The framework was also not fully automated at the time of its publication, leaving much room for the interpretation of its applicable cases and the space of legal transformations it effectively models. Speculative approaches like the work of Jimborean et al. also addressed the issue [JCD+14], but a general "`while` loop polyhedral framework" compatible with arbitrary affine transformations has yet to emerge. In this dissertation, we make a more pragmatic, short term step: we focus on the special case of dynamic counted loops where the most difficult of these problems do not occur.

There has also been a significant body of research specializing on high-performance implementations of sparse matrix computations. Manually-tuned libraries [BAA+14, BG09, BG11, LBG+12, MCG04, VDY05] are a commonly used approach, but it is tedious to implement and tune for each representation and target architecture. A polyhedral framework that can handle non-affine subscripts has a greater potential to achieve transformations and optimizations on sparse matrix computations, as illustrated by Venkat et al. [VHS15].

As a result, we would like to propose an automatic polyhedral compilation approach to parallelize and optimize dynamic counted loops that can express arbitrary affine transformations and achieve performance portability. We are allowed to make full use of systems of affine inequalities as implemented in state-of-the-art polyhedral libraries [Ver10] for our purpose. Moreover, following what has been implemented in the work [SCF03, SLC+16], we expect not to resort to more expressive first-order logic with non-interpreted functions/predicates such as the advanced analyses and code generation techniques of Wonnacott et al. [PW94b], while

avoiding the complexity and overhead of speculative execution.

3.1.2 Static Control Parts

The polyhedral compilation framework was traditionally limited to static control loop nests. It represents a program and its semantics using iteration domains, access relations, dependences and schedules. The statement instances are included in iteration domains. Access relations map statement instances to the array elements they access. Dependences capture the partial order on statement instances accessing the same array element (one of which being a write). The schedule implements a (partial or total) execution order on statement instances that is compatible with dependences.

Consider the running example in Figure 3.1. The upper bounds, m and n, of the j-loop and k-loop are computed in their common enclosing loop and updated dynamically as the i-loop iterates. As a result, it is not possible to classify the whole loop nest as a SCoP, and traditional polyhedral techniques do not directly apply. Tools aiming at a greater coverage of benchmarks—such as PPCG or LLVM/Polly—will abstract the offending inner loops into a black box, greatly limiting the potential for locality-enhancing and parallelizing optimizations.

```
   #pragma scop    //begin of our scop
   for (i=0; i<100; i++) {
S0:  m = f(i);
S1:  n = g(i);
     //begin of the scop of traditional techniques
     for (j=0; j<m; j++)
       for (k=0; k<n; k++)
S2:      S(i, j, k);
     //end of the scop of traditional techniques
   }
   #pragma endscop  //end of our scop
```

Figure 3.1 – Example with dynamic counted loops

As an alternative, one may narrow the SCoP by only considering the j-/k-loop nest and treating the dynamic upper bounds as symbolic parameters, enabling polyhedral transformations without problems. This, however, either introduces more frequent synchronizations by exploiting fine-grained parallelism when targeting on CPU targets, or misses the data locality along the outermost loop dimension and the opportunity to exploit full-dimensional parallelism on GPU platforms.

3.1.3 Our Solution

To extend the polyhedral framework to dynamic computed loops, we may need to address the following problems.

Modeling Control Dependences. Undeniably, the polyhedral model in its current form cannot handle dynamic counted loops. We would first derive a static upper bound for such dynamic conditions to make dynamic counted loops amenable to polyhedral compilation. To solve this problem, we may rely on the computation of an affine upper bound for all dynamic trip counts that a given loop may reach, using a combination of additional static analysis and dynamic inspection. Revisiting the polyhedral compilation framework [BPCB10] of arbitrary `while` loops, we introduce exit predicates for dynamic counted loops, modeling the control dependence of the original loop through additional data dependences from the definition of these exit predicates to every statement in the loop body.

Achieving Exact Dependence Analysis. Dynamic counted loops are commonly found in sparse matrix computations involving indirect array subscripts, preventing polyhedral model from achieving exact dependence analysis. We leverage the PENCIL language for eliminating alias suspicion and ambiguous analysis in the polyhedral model, allowing for the exact dependence analysis even in the presence of indirect array subscripts.

Eliminating the Effect of Over-approximations. Due to the over-approximation caused by deriving a static upper bound, we need to eliminate the introduced empty iterations for performance improvement. We extend the schedule-tree-based algorithm [GVC15] to enable the full automation of imperative code generation after the application of affine transformations, targeting both CPU and GPU architectures.

Our method goes beyond the state of the art [BPCB10, JCD+14, VHS15] in fully automating the process, specializing the code generation algorithm to the case of dynamic counted loops, and avoiding the introduction of spurious loop-carried dependences or resorting to speculative execution. We conduct experiments on representative irregular computations, including dynamic programming, computer vision, finite element methods, and sparse matrix linear algebra. We validate that the method is applicable to general affine transformations for locality optimization, vectorization and parallelization.

3.2 Extension Nodes in Schedule Trees

Our work follows the idea behind the work of Benabderrahmane et al. [BPCB10] by using over-approximations and modeling control dependences, the latter, however, misses a systematic code generation algorithm. The difficulty to generate early exits for over-approximations is

because such statements are not included in the iteration domain modeled by polyhedral compilation. It is difficult to model such statements at the time of the publication of the approach on general `while` loops. Fortunately, we may now leverage the schedule tree representation for such purpose.

The schedule tree representation has the same expressiveness with traditional polyhedral representations but it allows for the modeling of non-affine extensions in polyhedral compilation. In the case of dynamic counted loops, we may rely on the extension node of schedule trees to introduce additional domain elements to be scheduled.

Recall the mathematical concepts and polyhedral representations we introduced in Subsection 2.1.2, an extension node can be expressed using a named union map relating the outer schedule dimensions with the set of array elements accessed by the statement. In our case, we may abstract an early exit statement as a virtual statement accessing a scalar data named "exit". As a result, we may express a general extension node for such early exit statements as the following

$$\{(d_1, d_2, ..., d_n) \rightarrow exit()\} \tag{3.1}$$

with $(d_1, d_2, ..., d_n)$ representing the outer schedule dimensions and $exit()$ for the data accessed by the statements.

A similar use of extension nodes in PPCG [VCJC+13] is the creation of data copying statements for locality optimization and the introduction of thread-level synchronization instructions. A statement introduced by an extension node may be scheduled even it is originally excluded by the iteration domain of schedule trees.

3.3 An Overview of Our Approach

We may explain our approach by starting with dependence analysis. As shown in Figure 3.1, statement S_2 does not have data dependences on other statements. However, there are output dependences among definition statements of dynamic parameters m and n. To faithfully capture the scheduling constraints, one should also model the control dependences of S_2 over both headers of the enclosing dynamic counted loops. Such control dependences can be represented as data dependences between the definition statements of dynamic upper bounds and S_2.

To establish such a dependence relation, an exit predicate may be introduced before each statement of the loop body, like in the framework of Benabderrahmane et al. [BPCB10]. The resulting dependence graph is shown in Figure 3.2. The solid arrows represent the original (output) dependences between definition statements of dynamic parameters, and the

dashed arrows represent the data dependences converted from the exit conditions' control dependences.

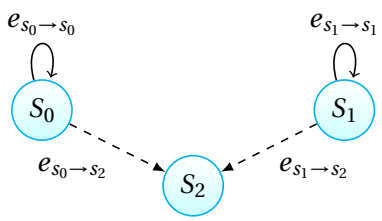


Figure 3.2 – Dependence graph of the example

By capturing control dependences as affine relations from the definition of exit predicates to dominated statements in loop bodies, one may build a sound abstraction of the scheduling constraints for the loop nest. This technique is applicable to arbitrary `while` loops, in conjunction with a suitable code generation strategy to recover the exact control flow protected by the exit predicate, and by over-approximating the loop upper bound as $+\infty$. This is the approach explored by Benabderrahmane et al., but the resulting polyhedral representation is plagued by additional spurious loop-carried dependences to update the exit predicate, removing many useful loop nest transformations from the affine scheduling space. In the more restricted context of dynamic counted loops, it is possible to eliminate those loop-carried dependences as the exit predicate only depends on loop-invariant data.

We base our formalism and experiments on the schedule tree representation [GVC15]. Schedule trees can be flattened into a union of relations form, with each relation mapping the iteration domain of individual statements to a unified logical execution time space.

Since dynamic counted loops cannot be appropriately represented in the iteration domain, a state-of-the-art polyhedral compiler like PPCG may only model the outer loop, abstracting away the j-loop and k-loop, as the schedule tree of Figure 3.3. Following Benabderrahmane's work [BPCB10], we can derive two static upper bounds, u_1 and u_2, that are greater than or equal to m and n. The domain and access relations of statement S_2 can be over-approximated accordingly, and represented parametrically in u_1 and u_2. This representation can be used to compute a conservative approximation of the dependence relation for the whole schedule tree.

Based on this dependence information, one may derive a correct schedule using the Pluto algorithm or one of its variants [BHRS08, VCJC+13], to optimize locality and extract parallelism. The resulting schedule tree may indeed be seen as a one-dimensional external domain and schedule enclosing a two-dimensional inner domain and schedule controlled by two additional parameters, u_1 and u_2, as will be seen in Figure 4.4.

The final step is to generate code from the schedule tree to a high level program. The generation of the abstract syntax tree (AST) follows the approach implemented in `isl` [Ver10], traversing the schedule tree and specializing the code generation algorithm to integrate target-specific

constraints, e.g., nested data parallelism and constant bounds. Before encountering a filter node associated with a dynamic counted loop, the exit predicate and its controlled loop body is seen as a single black-box statement by the AST generation algorithm. When passing the filter node constraining the dynamic upper bound, it is necessary to complement the standard code generation procedure with dedicated "dynamic counted loop control flow". This involves either (on GPU targets) the reconstruction of the exit predicate and the introduction of an early exit (`goto`) instruction guarded by the predicate or (on CPU targets) the replacing the over-approximated static upper bound with the dynamic condition and the removing of the introduced control flow. Our algorithm generates code in one single traversal of the schedule tree[1].

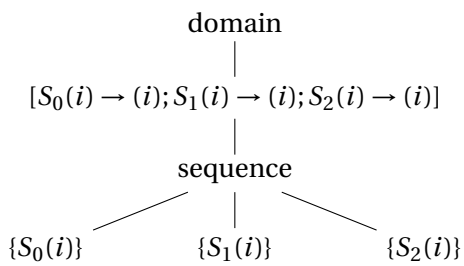


Figure 3.3 – Original schedule tree of the example

3.4 Related Work

The polyhedral framework is a powerful compilation technique to parallelize and optimize loops. It has become one of the main approaches for the construction of modern parallelizing compilers. Its application domain used to be constrained to static control, regular loop nests. But the extension of the polyhedral framework to handle irregular applications is increasingly important given the growing adoption of the technique. The polyhedral community invested significant efforts to make progress in this direction.

A representative application of irregular polyhedral techniques is the parallelization of `while` loops. The polyhedral model is expected to handle loop structures with arbitrary bounds that are typically regarded as `while` loops. Collard [Col94, Col95] proposed a speculative approach based on the polyhedral model that extends the iteration domain of the original program and performs speculative execution on the new iteration domain. Parallelism is exposed at the expense of an invalid space-time mapping that needs to be corrected at run time.

Beyond polyhedral techniques, Rauchwerger [RP95] proposed a speculative code transformation and hybrid static-dynamic parallelization method for `while` loops. An alternative, conservative technique, consists in enumerating a super-set of the target execution space [GL94, GC95, GGL98, GGL99], and then eliminating invalid iterations by determining termination detection on the fly. The authors present solutions for both distributed and shared

[1] Another difference with [BPCB10] where multiple traversals were needed.

memory architectures.

Benabderrahmane et al. [BPCB10] introduce a general framework to parallelize and optimize arbitrary `while` loops by modeling control-flow predicates. They transform a `while` loop as a `for` loop iterating from 0 to $+\infty$. Compared to these approaches to parallelizing `while` loops in the polyhedral model, our technique relies on systems of affine inequalities only, as implemented in state-of-the-art polyhedral libraries. It does not need to resort to the first-order logic such as non-interpreted functions/predicates, it does not involve speculative execution features, and it makes dynamic counted loops amenable to a wider set of transformations than general `while` loops.

A significant body of work addressed the transformation and optimization of sparse matrix computations. The implementation of manually tuned libraries [BAA+14, BG09, BG11, LBG+12, MCG04, VDY05] is the common approach to achieve high-performance, but it is difficult to port to each new representation and to different architectures.

Sparse matrix compilers based on polyhedral techniques have been proposed [VHS15], abstracting the indirect array subscripts and complex loop-bounds in a domain-specific fashion, and leveraging conventional Pluto-based optimizers on an abstracted form of the sparse matrix computation kernel. We ought to extend the applicability of polyhedral techniques one step further, considering general PENCIL code as input, and leveraging the semantical annotations expressible in PENCIL to improve the generated code efficiency and to abstract non-affine expressions.

4 Scheduling Dynamic Counted Loops in the Polyhedral Model

4.1 Preparation

A dynamic counted loop with a dynamic counted upper bound and a static lower bound is referred to as the normalized format of dynamic counted loops. For example, the code shown in Figure 3.1 is such a normalized format.

```
for (i=0; i<M; i++)
    for (j=idx[i]; j<idx[i+1]; j++)
      y[i] += A[j]*x[col[j]];
```

(a) An illustrative code of sparse matrix computation

$$A = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 0 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

$$idx[5] = \{0, 2, 4, 5, 6\}$$

$$col[6] = \{0, 1, 0, 1, 2, 3\}$$

(b) Sparse representations

```
for (i=0; i<M; i++)
    for (j=0; j<idx[i+1]-idx[i]; j++)
      y[i] = A[j+idx[i]]*x[col[j+idx[i]]];
```

(c) Normalized format

Figure 4.1 – A sparse matrix computation and its normalized format

Sparse matrix computations represent an important class of dynamic counted loops. They are a class of computations using compressed data layout stores nonzero elements only. Loops iterating on the compressed layout may have dynamic lower and upper bounds. In practice, such nonzero elements could be stored in different formats, leading to a variation among different formats on the sparse matrix. However, most of such formats could be transformed mutually, as explained by Venkat et al. [VHS15]. Figure 4.1(a) shows an example of such computations using Compressed Sparse Row (CSR) format. One may represent the sparse matrix as in Figure 4.1(b), with the additional arrays for storing the information about dynamic conditions.

However, these loops can be easily normalized by subtracting the lower bound from the upper bound, as shown in Figure 4.1(c). This transformation may introduce non-affine array subscripts since the lower bound may not be affine; we assume the dependence analysis will conservatively handle such subscripts, leveraging PENCIL annotations to refine its precision [CBF95, BBC+15]; we may also symbolically eliminate identical non-affine expressions on the left and right-hand side.

```
for (i=0; i<nodes; i++) {
  Anext=...
  Alast=...
  ...
  while(Anext<Alast) {
    S(i, Anext);
    Anext++;
  }
  ...
}
```

(a) A illustrative `while` loop

```
for (i=0; i<nodes; i++) {
  Anext=...
  Alast=...
  ...
  for (j=0; j<Alast-Anext; j++) {
    S(i, j+Anext);
  }
  ...
}
```

(b) Normalized format

Figure 4.2 – A `while` loop and its normalized format

Some forms of `while` loops may also be modeled, as long as an affine induction variable can be identified and assuming the variant part of the exit condition reduces to this induction variable. For example, the `while` loop shown in Figure 4.2(a) is extracted from the `equake` program of SPEC2000 benchmarks. It could be normalized to the format shown in Figure 4.2(b) without changing the semantic of the program.

4.2 Deriving a Static Upper Bound

To make a dynamic counted loop amenable to a polyhedral representation, our approach assumes that a static control upper bound u on the dynamic number of iterations is available. The general idea is that a dynamic counted loop can always be converted into a static `for` loop enclosing an `if` statement whose condition checks the dynamic bound.[1] One may determine the u parameter statically or dynamically.

4.2.1 Static Approaches

The u parameter can be approximated statically, as the dynamic upper bounds are functions of outer enclosing loop variables: a typical solution relies on Fourier-Motzkin elimination, projecting out enclosing dimensions and eliminating non-affine constraints.

For instance, the following set of dynamic conditions is extracted from the HOG benchmark of the PENCIL benchmark suite, which we may use Fourier-Motzkin elimination for eliminating the max/min operations and finally deriving a static upper bound.

$$\begin{cases} lb_x = max(f_x(x), 1) \\ lb_y = max(f_y(y), 1) \\ ub_x = min(g_x(x), 1) \\ ub_y = min(g_y(y), 1) \end{cases} \tag{4.1}$$

The u parameter can also be determined in other ways, from array size declarations or additional user-defined predicates in PENCIL [BBC+15]. We use the C99 type qualifiers/keywords `static const restrict` when declaring an array argument of a PENCIL function, guaranteeing the array argument do not alias and thereby allowing for the static derivation of the u parameter. When such static methods fail, `MAXINT` or any type-dependent bound remains a valid approximation, but a tighter bound is preferable to avoid lifting induction variables to a wider integral type.

[1] This is easier than a general `while` loop, since the dynamic bound check remains continuously false after its first falsification.

4.2.2 Dynamic Approaches

Besides static analysis, dynamic inspection prior ahead of the loop nest of interest may be practical in some cases. For example, in sparse matrix computations, u may be computed by inspecting the maximum number of non-zero entries in a CSR format. We may infer that the static upper bound of the sparse matrix shown in Figure 4.1(b) is 2 with an inspection. Alternatively, one may think about the transformation changes the sparse matrix into the form of

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & - \\ 6 & - \end{bmatrix}$$

All in all, affine bounds on the u parameter can generally be derived automatically, at compilation or run time, and the tightness of the approximation does not have an immediate impact on performance.

4.3 Modeling Control Dependences

To model control dependences on dynamic conditions, we introduce additional data dependences associated with exit predicates and their definition statements.

An exit predicate definition and check is inserted at the beginning of each iteration of a dynamic counted loop. At code generation time, all statements in the body of the counted loop will have to be dominated by an early exit instruction conditioned by its predicate. This follows Benabderrahmane's method for `while` loops [BPCB10], but without the inductive computation and loop-carried dependence on the exit predicate. Of course, we delay the introduction of `goto` instructions/changing back to the dynamic conditions until code generation, to keep the control flow in a statically manageable form for a polyhedral compiler. For example, the code in Figure 4.3(a) is preprocessed as the version in Figure 4.3(b) before constructing the affine representation.

The control dependences are therefore converted into data dependences between definition statements and the body of dynamic counted loops. Each statement in a dynamic counted loop is associated with a list of exit predicates. These predicates should be attached to the band node dominating the dynamic counted loop, and will be used to guard or terminate the execution within the over-approximation iteration domain bounded by the u parameters.

```
for (i=0; i<100; i++){
  m=f(i);
  n=g(i);
  for (j=0; j<m; j++)
      for (k=0; k<n; k++)
        S(i, j, k);
}
```

(a) Dynamic counted loops

```
for (i=0; i<100; i++){
  m=f(i);
  n=g(i);
  for (j=0; j<u1; j++)
      for (k=0; k<u2; k++)
        if (j<m && k<n)
            S(i, j, k);
}
```

(b) if conditional

Figure 4.3 – Conditional abstraction

4.4 Scheduling

The u parameter and conversion of control dependences make it possible to approximate dynamic counted loops in the polyhedral model, at the expense of traversing a larger iteration space. We may thus apply any affine scheduling on this "approximated static control program", to safely compute a correct schedule tree preserving all dependences. Based on the result of scheduling, we may leverage the mark node and extension node for accomplishing transformations for dynamic counted loops while preserving the correctness of the program.

4.4.1 Schedule Construction

The original domain node, as shown in Figure 3.3 can be expressed as

$$Domain = \{S_0(i); S_1(i); S_2(i) : 0 \leq i < 100\} \tag{4.2}$$

Note that statement S_2 is modeled as an one-dimensional statement without our technique. With the introduction of the u parameter and conversion of control dependences, we may

obtain a new domain node as

$$Domain = \{S_0(i); S_1(i) : 0 \le i < 100; S_2(i,j,k) : 0 \le i < 100 \wedge 0 \le j \le u_1 \wedge 0 \le k \le u_2\} \tag{4.3}$$

and a correct schedule tree representation shown as below.

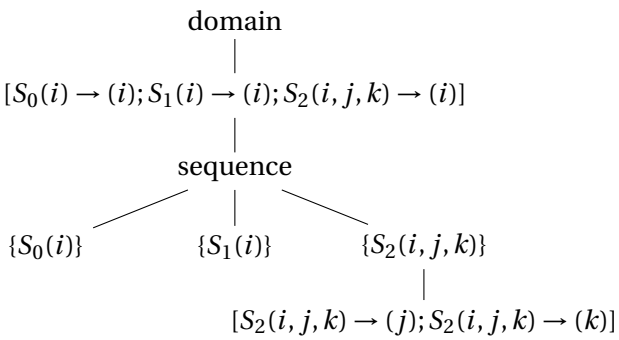


Figure 4.4 – A correct schedule tree of the example

Note that this schedule tree has not been applied any transformations and can be viewed as the original schedule tree representation. The dynamic parameters are assigned at their definition statements, and then virtually read by statement S_2 implicitly guarded by the negation of the exit predicates. This can be modeled as read and write (affine) access relations:

$$\begin{aligned} Read = \{&S_2(i,j,k) \to m[] \mid 0 \le i < 100 \wedge 0 \le j < u_1 \wedge 0 \le k < u_2; \\ &S_2(i,j,k) \to n[] \mid 0 \le i < 100 \wedge 0 \le j < u_1 \wedge 0 \le k < u_2\} \end{aligned} \tag{4.4}$$

$$Write = \{S_0(i) \to n[]; S_1(i) \to m[] \mid 0 \le i < 100\} \tag{4.5}$$

According to the variant of the Pluto algorithm implemented in `isl` [Ver10], one may set the validity dependences, associated with semantics preservation, to

$$\begin{aligned} Validity = (&Read^{-1} \circ Write' + Write^{-1} \circ Read' + Write^{-1} \circ Write') \\ &\cap (Schedule \prec Schedule') \end{aligned} \tag{4.6}$$

and the proximity dependences, associated with locality enhancement, to

$$\begin{aligned} Proximity = (&Read^{-1} \circ Write' + Write^{-1} \circ Read' + Write^{-1} \circ Write' \\ &+ Read^{-1} \circ Read') \cap (Schedule \prec Schedule') \end{aligned} \tag{4.7}$$

where *Schedule* represents the original schedule constructed from the original code according to the procedure above, and the $'$ (primed) maps distinguish iterations in dependence.

We can then compute a new schedule that applies the variant of the Pluto algorithm using

$$\begin{aligned} New_Schedule = &\text{ schedule } Domain \text{ under } Schedule \\ &\text{respecting } Validity \text{ and minimizing } Proximity \end{aligned} \tag{4.8}$$

In other words, the scheduling algorithm may safely compute a new schedule, starting from the original one shown in Figure 4.4, preserving all dependences and attempting to minimize the reuse distance.

4.4.2 Schedule Transformation

Once a correct schedule tree representation can be obtained, we are allowed to leverage any types of schedule nodes to apply schedule transformations. Indeed, it is possible to apply transformations and generate code without any special handlings on the current schedule tree. However, the generated code would waste a great number of iterations due to the over-approximations caused by the u parameters and conversion of control dependences, thereby inhibiting performance improvements.

Marking Dynamic Counted Loops. As we introduced before, a mark node can be used for retaining any pieces of information to schedule trees. We therefore are allowed to insert mark nodes above the nodes representing the dynamic counted loops, implying the child node would be considered as dynamic counted loops. As a consequence, one may obtain a schedule tree with a mark node shown in Figure 4.5. The only information we need to retain to a mark node is a string, i.e., "$dynamic_counted_loops$". The reason behind our intention to leverage a mark node comes from the strategy used in PPCG [VCJC+13] for generating thread-level synchronization instructions.

One may now use the schedule tree with mark nodes shown in Figure 4.5 for schedule transformation. As there are two separated, tilable band nodes in the schedule tree, a scheduling algorithm would identify the outer band node, corresponding the outermost i-loop in Figure 3.1, leading to a coarser parallelism. Without such abstraction, it would be impossible to model the whole program as a SCoP, as the polyhedral model could only obtain a schedule

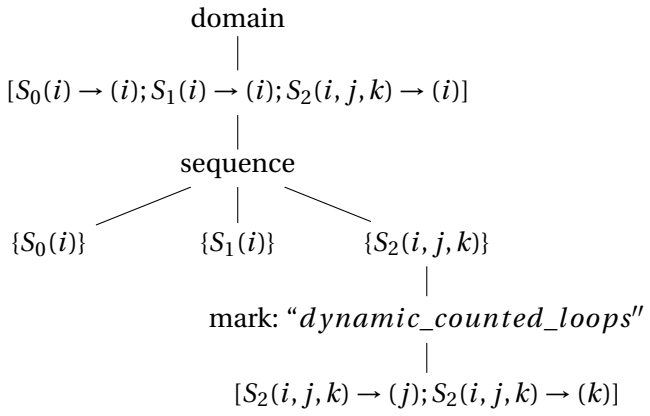


Figure 4.5 – Inserting mark nodes in the schedule tree

tree like Figure 3.3 shows, missing dependence information about S_2 and therefore failing to identify the outer tilable band.

Recall that we have mentioned in the previous section, an alternative way is to narrow the SCoP by only considering the j-/k-loop nest and treating the dynamic upper bounds as symbolic parameters, but such modeling strategy may introduce more frequent synchronizations by exploiting fine-grained parallelism or misses the data locality along the outermost loop dimension.

After applying schedule transformation, we need to handle the introduced mark node. One may notice that there exist two dynamic counted loops in the example, but we only introduce one mark node since the j-loop and k-loop are combined into one band node. The Pluto algorithm or its variants would always try to combine loops into one tilable band because such combination would exploit nested parallelism and hereby creating opportunities for more transformations. As transformations have been applied (for example, the scheduling algorithm may strip-mine the outer band or tile the inner band), we are free to split such combined band node. As a result, the schedule tree would be transformed into the following format (we do not represent the transformations that may have been applied in the figure).

In other words, a mark node would be broadcast to each dimension of a combined band node after splitting no matter whether such dimension is a dynamic counted loop. In case where a normal loop is also marked with such mark node, one can determine by checking whether the loop iterator appears in the predicates introduced for conversion of dynamic control. We will explain this issue further in the next section.

Non-affine Extensions. Extension nodes can now be used to replace each occurrence of the mark node. A mark node can only be used to attach additional information but not for custom implementations. As can be found from Figure 4.6, the loop dimension information is not present in a mark node as such information cannot be determined when introducing mark

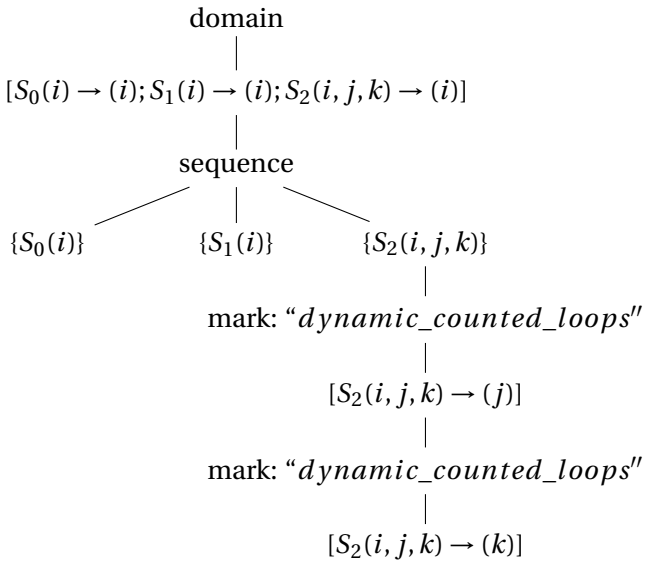


Figure 4.6 – Mark nodes with split band nodes

nodes. Besides, a mark node may also be broadcast after schedule transformation, making the retaining of further information like loop dimensions impractical at that stage.

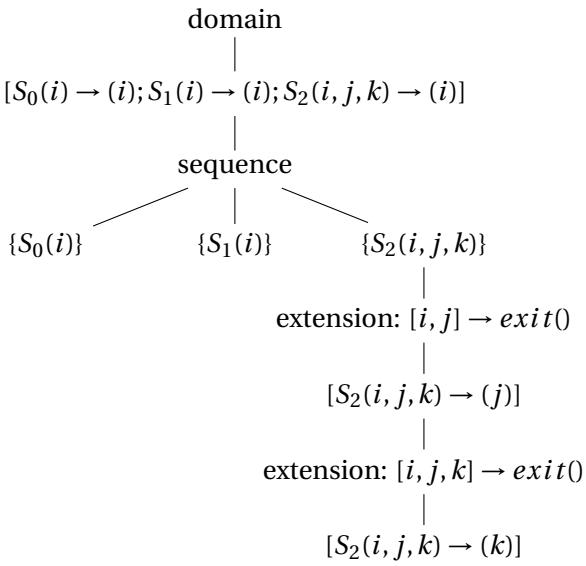


Figure 4.7 – Replace each mark node with an extension node

Figure 4.7 shows the schedule tree with each mark node being substituted by an extension node. Unlike mark nodes, an extension node can be used for recording loop nest information, calling for a custom implementation during code generation. We use an explicit expression in each extension node for the illustrative purpose, followed by the context used in the implementation of schedule trees. An extension node can be represented with a map, relating the

loop nest information to an early exit statement which implies an early exit instruction should be emitted.

Emitting early exit statements could be implemented differently depending on the target architecture. Generally, we force the code generator to change the introduced u parameters back to the original dynamic conditions when targeting CPUs, while a `goto` statement would be introduced for GPUs. The reason behind the different implementations is due to the different programming models used on different targets, as we will explain in detail next section.

5 Generation of Imperative Code

Once a new schedule is produced, additional transformations can be applied on band nodes, to implement loop tiling or additional permutations, strip-mining for vectorization, etc. Eventually, one needs to return to imperative code through a so-called code or AST generation algorithm. AST generation is a performance-critical step in any polyhedral framework. We extend the code generation scheme of Grosser et al. [GVC15], itself derived from the algorithm by Quilleré et al. [QRW00] and its CLooG enhancements and implementation [Bas04].

When the Grosser et al. algorithm traverses the band nodes in a schedule, it projects out the local schedule constraints from the domain node. As the dynamic upper bounds are not modeled in the iteration domain (the domain node in the schedule tree and subsequent filter nodes), the generated loops will iterate from 0 to u. It is thus necessary to emit an early exit statement (for GPU architectures) or change the over-approximated static upper bound back to the original dynamic condition (for CPU architectures). Besides, the introduced control flow can also be removed when generating code for CPU targets, reducing the control overhead.

5.1 Extending the Schedule Tree

Let us first recall the extensions we made to the schedule tree in the last section. The Grosser et al. algorithm is not able in its original form to generate semantically correct code for our extended schedule tree. However, it can be easily modified to handle the special case of exit predicates that are homogeneous over all statements in a sequence or set node of the schedule tree (e.g., all statements in a band of permutable loops).

This is facilitated through the syntactic annotation of dynamic counted loops using so-called mark nodes in the schedule tree. A mark node may attach any kind of information to a subtree; we used it here to specify which band nodes and which dimensions in those bands involve dynamic counted loops. To account for affine transformations combining static and dynamic counted loops (e.g., loop skewing), mark nodes are inserted at every dimension.

One may insert an extension node in a schedule tree to extend its iteration domain, e.g., to

insert a new statement with a specific iteration domain. In our case, we replace each mark node with an extension node, inserting a guard statement with the proper exit predicate. In a first pass, all exit predicates are attached to the band node; a follow-up traversal through the predicate list lets the AST generator detect whether a dimension of the band node is a dynamic counted loop, and position early exits at the right level.

In fact, the code generation scheme is designed to complement the standard code generation procedure with a dedicated "dynamic counted loop template". This template serves as a post-processing step after code generation, involving the reconstruction of the exit predicate. To this stage, we may have to consider the features of target architectures and branch the code generation template.

We generate CUDA code when targeting on GPU architectures. However, it does not mean our technique is only restricted to such programming APIs. When launching a CUDA kernel, a fix-length loop bound should be specified for execution configuration. We may therefore emit an exit statement every time when an extension node is encountered to force the over-approximated loops terminate correctly. When we implement a custom code generation scheme for CPU architectures, we may generate loops amenable to OpenMP directives, implying an exit statement is not allowed to jump out of the parallel region but a dynamic loop bound is permitted. As a result, we choose to change the introduced u parameters back the original dynamic computed bounds and remove the associated predicates for eliminating control overheads.

5.2 Generating Early Exits

When scanning the schedule tree to generate early exits for GPU targets, the AST generator creates a `goto` AST node for each of the above-mentioned extension nodes. A `goto` statement can be generated from the AST node using the following steps.

Positioning a `goto` Statement. As shown in (3.1) and Figure 4.7, an extension node is expressed with a named union map, relating the outer schedule dimensions with the introduced exit statement. Such outer schedule dimensions could be used for positioning the `goto` of each exit statement.

Generating the Guard. During modeling control dependences, a predicate was introduced at the beginning of each iteration of a dynamic counted loop. Such predicates would be recorded and passed over to the AST generator. A `goto` statement should also be guarded by a predicate by negating one of the conditions of the introduced predicate dominating the body of the corresponding dynamic counted loop, as the conditions may be a conjunction of multiple dynamic counted loops, e.g., the code shown in Figure 4.3(b). This can be attained by checking the appearance of the loop iterator of current dimension in the conditions.

Determining Whether the Loop is a Dynamic Counted Loop. As we explained in subsection 4.4.2, the band node with a mark node specifying dynamic counted loop would be split after schedule transformation, with each dimension marked with the same mark node by broadcasting the latter. This broadcast may also mark a static loop as dynamic. Of course, all of the mark nodes would be replaced with extension nodes. When generating a `goto` statement by checking each occurrence of such extension nodes, the AST generator should determine whether the current dimension is a dynamic counted loop. Following the idea of generating the guard, this can also be achieved by inspecting the appearance of the loop iterator of current dimension in the conditions of the predicate. One may determine the current dimension is a static, normal loop provided the loop iterator is not present in the conditions.

Counting Labels. A `label` destination is required when using a `goto` statement. As a result, the AST generator should maintain a global `label` counter, enforcing the exit statements jump to the corresponding destinations. The `label` counter is incremented each time a dynamic counted loop is encountered, enforcing uniqueness.

5.3 Changing Back to Dynamic Conditions

When targeting on CPU architectures, it may not be allowed to jump in or out of the parallel region using an early exit statement like `goto`, but one may change the over-approximated static upper bound u back to the original dynamic condition. The information to facilitate such replacement can be attached to an AST annotation node and be the same with those of the `goto` AST node in GPU case except the `label` counter. The code generation is similar to the case of GPU case, being accomplished by the following steps.

Positioning Dynamic Counted Loops. Like what we have explained in GPU case, the AST generator has to pick out dynamic counted loops from the band node due to the broadcast of mark nodes. The method has been introduced above, and the AST generator can follow the same scheme used in GPU case for recognizing such dynamic counted loops.

Substituting the u Parameters. The AST code generator may look up the predicate list and extract the condition corresponding to the current dimension. The right-hand side of the condition can be taken out to substitute the introduced u parameter of a dynamic counted loop.

Removing Control Overheads. Similarly, each occurrence of the earlier introduced dynamic conditions at the beginning of each iteration can now be degenerated for eliminating such control overheads.

5.4 Code Generation for a Single Loop

The final step is converting the AST to a high level program. When a `goto` AST node of a dynamic counted loop is captured, a `goto` statement conditioned by its predicates is enforced after the loop body, as well as a `label` destination after the loop itself. The associated predicates are gathered in a conjunction and wrapped as one conditional, with loop iterators instantiated according to the loop level. A `label` is inserted after each dynamic loop as a target for a `goto` statement.

```
for (i =0; i<N; i++) {
  for (j =0; j<u1; j ++) {
    m=f(i);
    if(j<m)
      S1(i,j);
  }
}
for (i =0; i<N; i++) {
  for (j =0; j<u2; j ++) {
    n=g(i);
    if(j<n)
      S2(i,j);
  }
}
```

(a) Before fusion

```
for (i =0; i<N; i++) {
  for (j =0; j< max (u1 ,u2); j ++) {
    m=f(i);
    n=g(i);
    if(j<m)
      S1(i,j);
    if(j<n);
      S2(i,j);
    if(j >=m && j >=n)
      goto label0 ;
  }
  label0 : ;
}
```

(b) After fusion

Figure 5.1 – Fusing two dynamic counted loops

Changing back to the dynamic condition for a dynamic counted loop is straightforward, but special cares have to be taken to handle cases with multiple associated predicates. One may construct a *max* operation comprising all the associated predicates as the upper bound of a dynamic counted loop, without removing these introduced control flow since they have to be there to preserve the semantic of the code.

This schedule-tree-based code generation algorithm enables all kinds of loop transformations, with the most challenging one being loop fusion. When fusing two dynamic counted loops, the two sets of predicates are considered, and the early exit statements/*max*-operation-based dynamic upper bounds are guarded by/composed of their statementwise conjunction/them. As shown in Figure 5.1 is the original and fusion result of two dynamic counted loops. One may conclude from the figure for CPU architectures easily. A normal loop can be treated as a specific case of dynamic counted loop by reasoning on its static upper bound as a predicate.

Unfortunately this scheme efficiently supports a single dynamic counted loop only, and does not deal with the expression of parallelism in these loops.

5.5 Flat and Nested Parallelisms

As shown in Figure 4.4, the canonically constructed schedule tree isolates two nested band nodes to represent different levels of the loop nest. This works fine when the target architecture is a shared memory multiprocessor. As an illustrative example, Figure 5.2 is the generated code for a shared memory multiprocessor after the application of loop tiling on the code in Figure 3.1 with the outermost *i*-loop being parallelized. We also depict the corresponding schedule tree representation in Figure 5.3 for reference.

```
#pragma omp parallel for
for (i=0; i<100; i++) {
  m = f(i);
  n = g(i);
  for (jj=0; jj<m/BB+1; jj++)
    for (kk=0; kk<n/CC+1; kk++)
      for (j=0; j<min(m, jj*BB+BB); j++)
        for (k=0; k<min(n, kk*CC+CC); k++)
          S(i, jj, kk, j, k);
}
```

Figure 5.2 – Code generation with loop tiling for CPU

However, when targeting GPU accelerators or producing fix-length vector code, we usually expect to combine nested bands to express parallelism at multiple levels, and a constant iteration count may also be required for data-parallel dimensions. We therefore consider two

cases depending on the need to extract parallelism across more than one band.

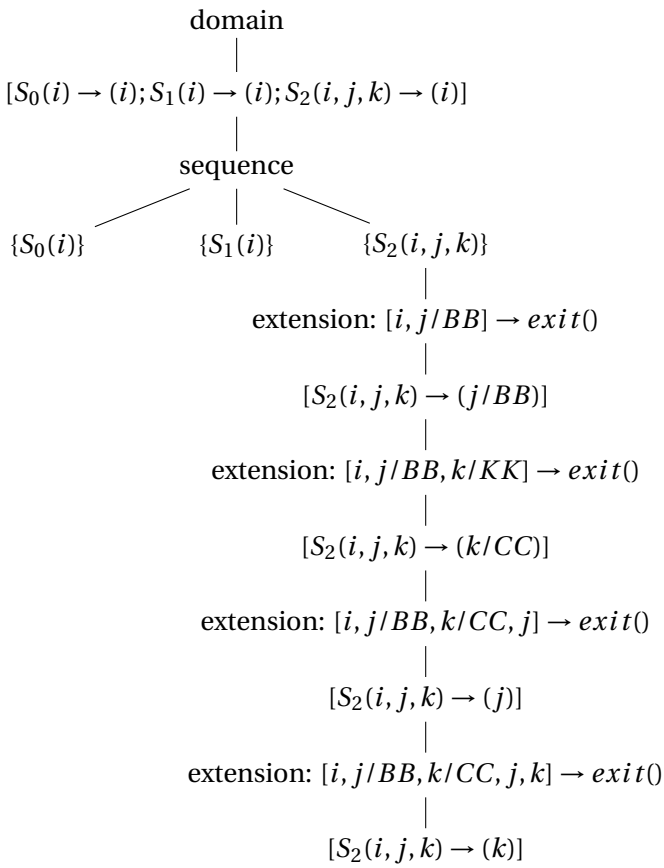


Figure 5.3 – The schedule tree representation of code shown in Figure 5.2

5.5.1 Flat parallelism within a band

Let us first discuss the case of regenerating imperative code for one or more nested dynamic counted loops within a single band. As a first step, one may systematically generate conditional statements on exit predicates at the innermost level. Figure 4.3(b) shows an example illustrating this approach. The predicates of both loops are included in a single conditional, and generated under the inner loop. Notice that this approach is compatible with affine loop transformations such as loop interchange, not expressible in [BPCB10] due to the presence of spurious loop-carried dependences.

Our approach is generally applicable in the context of loop interchange except when one attempts to permute a dynamic counted loop with its enclosing affine loop governing the dynamic condition. As we introduce redundant, empty iterations to dynamic counted loops, we inject early exits for eliminating the effect of such over-approximation. In this special interchange case, the introduction of early exits may not guarantee the semantic of original programs. We therefore leave out the introduction of such early exits in this special interchange

case. As an illustrative example, Figure 5.4(b) shows the result of permuting the dynamic counted loop with its governing affine loop in the example of Figure 5.4(a). Such treatment may not remove the empty iterations introduced by the over-approximation but the semantic of the program could be preserved. In summary, our approach always tries to remove or minimize the introduced empty iterations but puts the correctness of the program in the first place, implying that we may have to sacrifice the performance by skipping the injection of early exits in some rare cases.

```
for (i=0; i<N; i++) {
  m = f(i);
  for (j=0; j<m; j++)
    S(i,j);
}
```

(a) Original code

```
for (j=0; j<u; j++) {
  for (i=0; i<N; i++) {
    m = f(i);
    if (j<m)
      S(i,j);
  }
}
```

(b) After interchange

Figure 5.4 – An interchange example

Yet one still needs to generate early exits in order to avoid traversing a potentially large number of empty iterations. We may extract the iterators one by one from the predicate list and generate the corresponding exit statements from the innermost outwards. The exit predicates are generated in the form of multiple conditionals rather than else branches, as shown in Figure 5.4 and 5.6. Unlike Jimborean et al. [JCD+14], we do not need speculation on the number of iterations, since we do not deal with general `while` loops; our technique always executes the same number of iterations as the original programs.

Loop tiling is a special case that should be taken into account. Loop tiling involves the insertion of one or more additional schedule dimensions through strip-mining. When strip-mining a dynamic counted loop, there should be an exit statement at both levels. For the point loop—iterating within a tile—the common case above applies. For the tile loop—iterating among tiles—we align its bounds and strides to follow the structure of the inner loop, so that its counter can also be compared systematically with the same bound.

5.5.2 Nested parallelism across bands

Targeting GPU accelerators or producing fix-length vector code motivates the exploitation of data parallelism within dynamic counted loops, in combination with other nested loops. Since dynamic counted loops result in nested bands in the schedule tree, the combined exploitation of multiple levels of parallelism including one or more dynamic counted loops requires special treatment that is not directly modeled by affine sets and relations. The constraints on the grid of multi-level data parallelism require the collection of bound information across nested bands: when launching a kernel, the parameters of the grid must be known and may not evolve during the whole run of the kernel.

Unfortunately, the statements between nested bands that occur in dynamic counted loops are used to initialize dynamic upper bounds. Statements in the body of these dynamic counted loops depend on those definition statements, through the added dependences modeling the original dependence of the dynamic loop. Still, one can sink these definition statements inside, within the dynamic counted loops, as a preprocessing step. Figure 5.5(a) shows the code after sinking the definition statements of the example in Figure 3.1, followed by a depiction on its schedule tree in Figure 5.5(b).

Note that both dynamic definition statements change into 3-dimensional statements due to the inward movement. In the schedule representation, we use a "band" to represent the band node after such operation, which can be expressed as a piecewise schedule, $[\{S_0(i,j,k) \to (i); S_1(i,j,k) \to (i); S_2(i,j,k) \to (i)\}, \{S_0(i,j,k) \to (j); S_1(i,j,k) \to (j); S_2(i,j,k) \to (j)\}, \{S_0(i,j,k) \to (k); S_1(i,j,k) \to (k); S_2(i,j,k) \to (k)\}]$. As a result, the nested bands can be combined again, with no intervening computation or control flow.

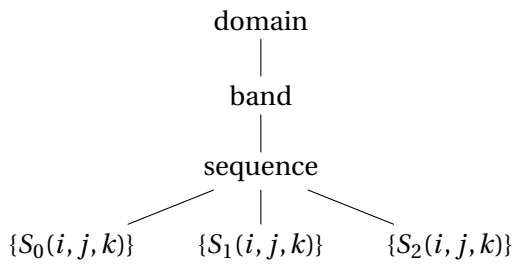
The inward movement of these definition statements is safe with the introduction of the upper bound u-parameter. Yet as a side-effect of this movement, each definition will be redundantly evaluated as many times as the number of iterations of the dynamic counted loop itself. This is the price to pay for a fixed upper bound on the iterations.

Once again, this overhead may be mitigated with additional strip-mining of the outer loops, to better control the value of u, effectively partitioning the loop nest into coarse-grain sub computations amenable to execution on a heterogeneous target. Figure 5.6 shows an example after the application of loop tiling on the code in Figure 3.1, and one may also refer to Figure 5.7 for the schedule representation.

As the nested bands are combined into a single one, a polyhedral framework would identify it with multiple dimensions of parallelism, partitioning the loop nest into coarse sub-problems that can be solved independently on heterogeneous platforms.

```
for (i=0; i<100; i++)
  for (j=0; j<u1; j++)
    for (k=0; k<u2; k++){
      m=f(i);
      n=g(i);
      if (j<m && k<n)
        S(i, j, k);
    }
```

(a) Sinking the dynamic definitions of the example in Figure 3.1



(b) The schedule tree of the code shown in Figure 5.5(a)

Figure 5.5 – Sinking the dynamic definition and its schedule tree representation

5.6 General Applicability to Loop Transformations

One of the benefits of our approach with respect to Benabderrahmane et al.'s work [BPCB10] is its compatibility to various loop transformations. We analyze in this subsection the general applicability of our approach for each loop transformation presented in Subsection 2.1.3 and their combinations.

5.6.1 Loop Transformations of Unimodular Matrices

We first analyze the case of loop transformations covered by unimodular matrices [Ban93, WL91], i.e., loop interchange, skewing and reversal. One may view unimodular matrices as loop transformations for a single statement because the loop body may always be abstracted as a black box and the structure stays unchanged under such transformations.

Our approach is compatible to loop interchange, as we explained in Subsection 5.5.1 with a special treatment designed for permuting dynamic counted loops with the enclosing affine loop. Similarly, such strategy can also be applicable to loop reversal since the iterations of a dynamic counted loop would be traversed in a reversed order, guaranteeing the correctness of our technique by introducing over-approximations. Fortunately, such cases rarely happen in practice as the control dependences caused by the dynamic conditions prevent such permutation and they are not seen in our experiments.

```
for (ii=0; ii<100/AA+1; ii++) {
  for (jj=0; jj<u1/BB+1; jj++) {
    for (kk=0; kk<u2/CC+1; kk++) {
      for (i=ii*AA; i<min(100, ii*AA+AA); i++) {
        for (j=jj*BB; j<min(u1, jj*BB+BB); j++) {
          for (k=kk*CC; k<min(u2, kk*CC+CC); k++) {
            m = f(i);
            n = g(i);
            if (j<m && k<n)
              S(j, k);
            if (k>=n)
              goto label0;
          }
          label0: ;
          if (j>=m)
            goto label1;
        }
        label1: ;
      }
      if (kk*CC>=n)
        goto label2;
    }
    label2: ;
    if (jj*BB>=m)
      goto label3;
  }
  label3: ;
}
```

Figure 5.6 – Code generation with loop tiling for GPU

In fact, the injection of early exits may only be affected by the iteration reordering of a dynamic counted loop, i.e., loop reversal, and/or the change of the dynamic condition, i.e., the interchange with the governing affine loop. As a result, the validation of the correctness of our method on loop skewing is straightforward, as no such transformations happen in skewing and the introduced predicates before each iteration of the dynamic counted loop would also be updated with respect to the result of skewing.

To conclude, our method on dynamic counted loops are always correct for loop transformations covered by unimodular matrices, and any combinations of these transformations.

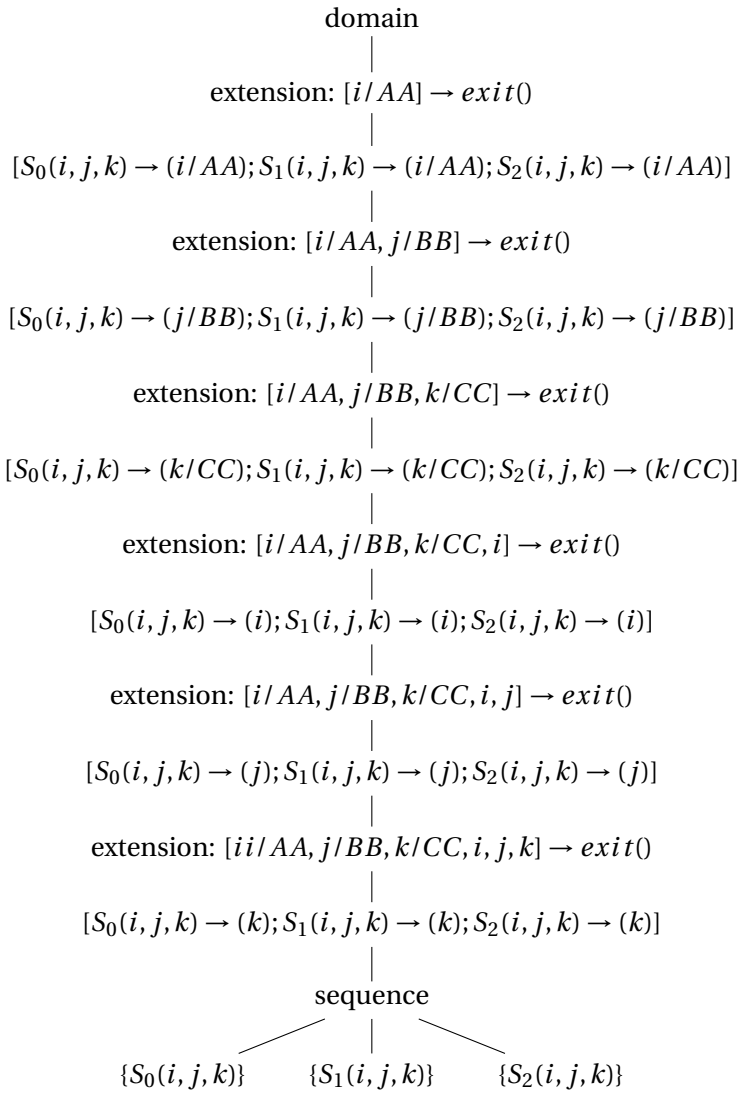


Figure 5.7 – The schedule tree representation of the code shown in Figure 5.6

5.6.2 Loop Transformations in Code Generation

As we explained in Subsection 2.1.3, some loop transformations including loop peeling, unrolling, unswitching, are achieved by code generation. Our approach is valid for such cases since the code generator may only change the loop structure instead of reordering statement instances. For example, we are allowed to apply our method on each version after loop peeling and/or unswitching, and the predicate may be introduced before each instance after unrolling.

Similar to the case of unimodular matrices, loop transformations achieved by code generation can also be viewed as transformations applied on loop nests with a single statement, since the loop body of each version after such transformations stays unchanged.

5.6.3 Other Loop Transformations

The analysis on index set splitting [GFL00] can follow the case of loop transformations achieved by code generation, as we always split the index set by introducing an affine parameter, implying the loop structure after splitting may stay unchanged.

Strip-mining [KP95], unrolling-and-jam [Bon08, BF03] and loop tiling can be put together as the first two transformations can be viewed as special case of loop tiling. The interchange involved in tiling will not change the order of a dynamic counted loop and its governing loop, neither the iterations of the dynamic counted loop, meaning the introduction of early exits should always be correct.

The solution to loop fusion are discussed in Subsection 5.4 and the validation of the correctness is therefore straightforward. One may see loop fission as a reverse transformation of fusion, and the general applicability of our method for fission is also validated. Loop fusion and fission are transformations that apply on multiple statements since they change the body of the loop nest.

As our method is correct on each loop transformation, it should also be correct on all combinations of these transformations.

6 Experimental Results

Our framework takes a C program as input, and resorts to PENCIL [BBC+15] extensions only when dealing with indirect accesses (subscripts of subscripts), implying that all arrays are declared through the C99 variable-length array syntax with the `static const restrict` qualifiers, allowing PPCG to derive the size of the arrays offloaded on the accelerator despite the presence of indirect accesses, and telling that these arrays do not alias.

We use PPCG [VCJC+13] to generate target codes, a polyhedral compiler that performs loop nest transformations, parallelization, data locality optimization, and generates OpenCL or CUDA code. The version `ppcg-0.05-197-ge774645-pencilcc` is used in our work. In a follow-up auto-tuning step, we look for optimal parameter values for tile sizes, block sizes, grid sizes, etc. for a given application and target architecture.

The experiments are conducted on a 12-core, two-socket workstation with an NVIDIA Quadro K4000 GPU. Each CPU is a 6-core Intel Xeon E5-2630 (Ivy Bridge). Sequential and OpenMP code are compiled with the icc compiler from Intel Parallel Studio XE 2017, with the flags *-Ofast -fstrict-aliasing* (*-qopenmp*). CUDA code is compiled with the NVIDIA CUDA 7.5 toolkit with the *-O3* optimization flag. We run each benchmark 9 times and retain the median value. Median rather than the mean, for more stability. Long discussion there, this is not idea either in general, but more suitable here. Note that the median pushes for an odd number of runs.

6.1 Dynamic Programming

Dynamic programming is an alternative method of greedy algorithms to guarantee an optimal solution. In computer science, dynamic programming implies the optimal solution of the given optimization problem can be obtained by the combination of optimal solutions of its sub-problems, by solving the same sub-problems recursively rather generating new ones. Dynamic counted loops are usually involved in these problems. We investigate two representative dynamic programming problems—change-making and bucket sort.

Typically, the change-making problem is used to find the minimum number of coins that can

add up to a certain amount W and to count how often a certain denomination is used, but it has a much wider application than just currency. The algorithm is also used to count how often a certain denomination is used.

Suppose N denominations are provided, each of which is $d_i (0 \leq i < N)$. As long as the given amount $W > d_i$, the frequency of the i-th denomination will be incremented by 1. As a result, d_i appears as a bound of the inner dynamic counted loop, enclosed by an outer loop iterating over the total number of denominations. Our technique successfully parallelizes the inner dynamic counted loop and generates the CUDA code in conjunction with a loop interchange optimization. We show the performance with different number of denominations N under different amount constraints W in Figure 6.1. It can be concluded from the figure that the performance improvement grows with the rise of the the number of denominations.

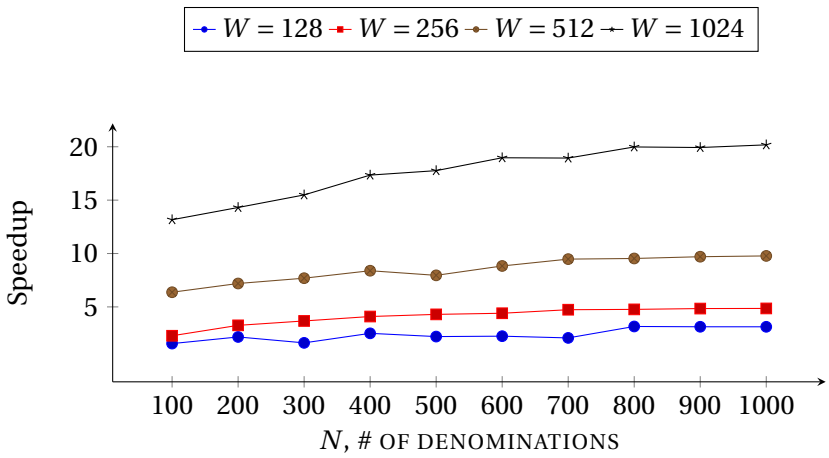


Figure 6.1 – Performance of change-making on GPU

Bucket sort is a generalization of counting sort, sorting by first scattering the N elements of a given array into a set of M buckets, sorting each bucket individually, and finally gathering the sorted elements in each bucket in order. Due to the comparison operations, a sorting algorithm is inherently not the candidate for parallelization. However, it is possible to parallelize and optimize the gathering step of bucket sort.

We consider a uniform random distribution of elements of the input array. The algorithm has to gather $size[i]$ elements in the i-th bucket, whose static upper bound can be set as N. The dynamic counted loop controlled by the bucket size is captured by our method and parallelized in the form of CUDA code on GPUs. The performance with different array sizes N and different bucket numbers M is shown in Figure 6.2, indicating the speedup rises along with the increase of the number of buckets involved.

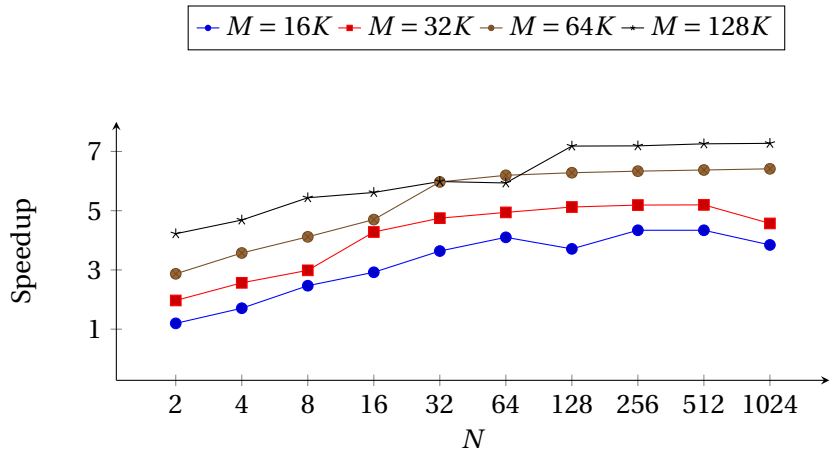


Figure 6.2 – Performance of the bucket sort on GPU

6.2 HOG Benchmark

The HOG benchmark is extracted from the PENCIL benchmark suite[1], a collection of applications and kernels for evaluating PENCIL compilers. The distribution of intensity gradients or edge directions describe the local object appearance and shape within an image. When processing an image, the HOG descriptor divides it into small connected regions called cells. A histogram of gradient directions is then compiled for the pixels within each cell. The descriptor finally concatenates these histograms together. The descriptor also contrast-normalize local histograms by calculating an intensity measure across a block, a larger region of the image, and then using this value to normalize all cells within the block to improve accuracy, resulting in better invariance to changes in illumination and shadowing.

The kernel of the HOG descriptor contains two nested, dynamic counted loops. The upper bounds of these inner loops are defined and vary as the outermost loop iterates. The dynamic parameter is an expression of max and min functions of the outer loop iterator and an array of constants. We derive the static upper bound parameter u from the BLOCK_SIZE constant, a global parameter of the program to declare the size of an image block.

Since we target a GPU architecture, we ought to extract large degrees of parallelism from multiple nested loops. As explained in the previous section, we sink the definition statements of dynamic parameters within inner dynamic counted loops and apply our AST generation scheme for a combined band for GPU architecture. We may then generate the CUDA code with parameter values for tile sizes, block sizes, grid sizes, etc. We show performance results with and without host-device data transfer time, in Figure 6.3, considering multiple block sizes. The detection accuracy improves with the increase of the block size. Our algorithm achieves a promising performance improvement for each block size, and our technique can obtain a speedup ranging from 4.4× to 23.3× while the PENCIL code suffers from a degradation by

[1] https://github.com/pencil-language/pencil-benchmark

about 75%.

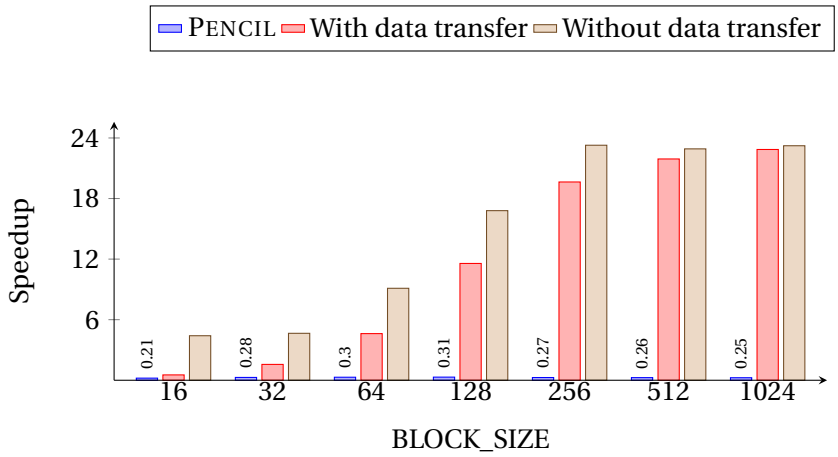


Figure 6.3 – Performance of the HOG descriptor on GPU

6.3 Finite Element Method

`equake` is one of the SPEC CPU2000 benchmarks. Inspired by the 1994 Northridge Earthquake aftershock in the San Fernando Valley of Southern California, this program is designed to model earthquake by simulating the propagation of elastic waves in large, highly heterogeneous valleys, so that the time history of the ground motion everywhere within the valley can be recovered. It follows a finite element method, operating on an unstructured mesh that locally resolves wavelengths. The kernel invokes a 3-dimensional sparse matrix computation, followed by a series of perfectly nested loops. We inline the follow-up perfectly nested loops into this sparse matrix computation kernel to expose opportunities for different combinations of loop transformations.

In the 3-dimensional sparse matrix computation, a reduction array is first defined in the outer i-loop, and every element is repeatedly written by a j-loop that is enclosed by a `while` loop iterating over the sparse matrix. Finally, these reduction variables are gathered to update the global mesh. The `while` loop can be converted to a dynamic counted loop via preprocessing.

One may distribute the three components of the sparse matrix computation kernel, generating a 2-dimensional permutable bands on the dynamic counted loop in conjunction with unrolling j-loop, and fusing the gathering component with its follow-up perfectly nested loops. This case is called "2D band" in Figure 6.4.

One may also interchange the dynamic counted loop with its inner j-loop. As a result, all of the three components of the sparse matrix computation are fused. The loop nest is separated into two band nodes, the outer is a 2-dimensional permutable and the inner is dynamic counted loop. This is called "(2+1)D band" in the figure.

Alternatively, the three components can be distributed instead of being fused. This makes a 3-dimensional permutable band involving the dynamic counted loop, and results in the fusion of the gathering component with the follow-up perfectly nested loops. This case is called "3D band" in the figure.

We generate CUDA code for these different combinations and show the result in Figure 6.4, considering different input sizes. The u parameter is set to the maximum non-zero entries in a row of the sparse matrix. The baseline parallelizes the outer i-loop only, which is what PPCG does on this loop nest; we reach a speedup of 2.7× above this baseline.

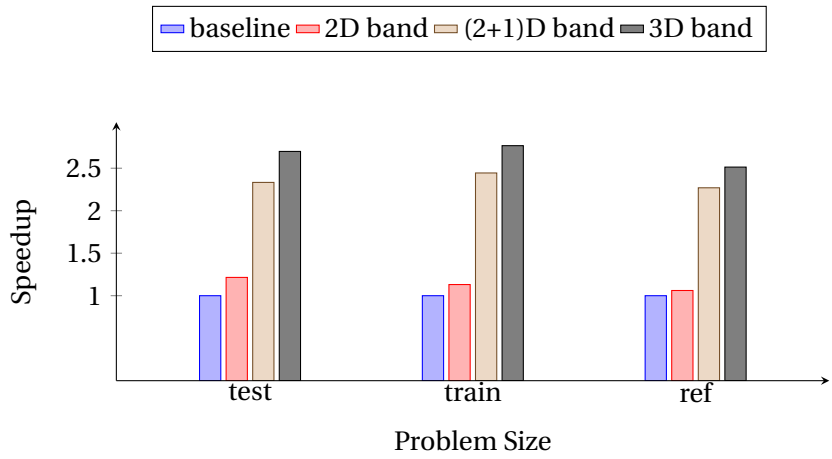


Figure 6.4 – Performance of `equake` on GPU

6.4 Sparse Matrix-Vector Multiplication

Sparse matrix operations are an important class of algorithms frequently in graph applications, physical simulations to data analytics. They attracted a lot of parallelization and optimization efforts. Programmers may use different formats to store a sparse matrix, among which we consider four representations: CSR, Block CSR (BCSR), Diagonal (DIA) and ELLPACK (ELL) [VDY05]. Our experiment in this subsection target the benchmarks used in [VSHS14, VHS15], with our own modifications to suit the syntactic constraints of our framework.

We first consider the CSR representation. The other three representations can be modeled with a make-dense transformation, as proposed by [VHS15], followed by a series of loop and data transformations. BCSR is the blocked version of CSR, its parallel version is the same as that of CSR, after tiling with PPCG. We will therefore not show its performance. Note that Venkat et al. [VHS15] assume block sizes are divisible by loop iteration times, but our work has no such limitation. The inspector is used to analyze memory reference patterns and to generate communication schedules, so we mainly focus on comparing our technique to the executor. The executor of DIA format is not a dynamic counted loop and will not be studied.

In the original form of the CSR format, loop bounds do not match our canonical structure:

we apply a non-affine shift by the dynamic lower bound as discussed earlier. The maximum number of non-zero entries in a row is the static upper bound and may be set as the u parameter. It can be derived through an inspection. As a result, the references of indirect array subscripts can be sunk under the inner dynamic counted loop, exposing a combined band in the schedule tree.

Venkat et al. [VSHS14] generate two optimized versions for the CSR format on GPU target. The first version parallelizes the outer loop by strip-mining the latter into two dimensions with the outer dimension mapped to thread blocks and the inner dimension to threads, mapping each row of the sparse matrix to a thread; the inner loop is executed sequentially. This version is referred to as "scalar". The second version, i.e., "vector", also parallelizes the outer loop but assigns multiple threads to each row of the sparse matrix; in addition, the inner loop is further strip-mined to allow for intra-wrap reduction. In both cases, only the outer loop is identified as a permutable band while the "vector" version exploits intra-wrap reduction by assigning multiple thread to a row, parallelizing the inner loop in without combining it with its outer loop and missing loop transformations across loop nest.

Our technique can identify the inner dynamic counted loop and parallelize both loops, exposing a higher degree of parallelism and allowing for loop tiling rather than (nested) strip-mining when comparing with Venkat et al.'s scalar/vector version. Tiling on CSR format may reduce the impact of thread imbalance issue when the input sparse matrix is unstructured, i.e., the number of non-zero entries in each varies greatly. We introduce an atomic operation in the generated code for preserving the correctness of reduction of sparse matrix computations.

We show the performance in Figure 6.5, using the matrices obtained from the University of Florida sparse matrix collection [DH11] as input, and the properties of the input matrices are listed in Table 6.1. We also show the performance of a manually-tuned library–CUSP [BG09] in the figure for a comparison with hand-written implementations.

Table 6.1 – Summary of the input sparse matrices

Matrices	Symmetric	# of Nonzero Entries	Rows×Columns
cant	yes	4007383	62451×62451
consph	yes	6010480	83334×83334
cop20k_A	yes	2624331	121192×121192
mac_econ_fwd500	no	1273389	206500×206500
mc2depi	no	2100225	525825×525825
pdb1HYS	yes	4344765	36417×36417
Pres_Poisson	yes	715804	14822×14822
pwtk	yes	11634424	217918×217918
rma10	no	2374001	46835×46835
tomographic1	no	647495	73159×59498

Our method beats the scalar version of Venkat et al. in most cases due to the higher degree of parallelism. Benefiting from the intra-wrap parallel reduction, the vector version of Venkat

et al. and the CUSP library outperform our code with the `pdb1HYS` matrix, since the thread imbalance issue could by reduced by the structure of input matrices. However, our code is always better than Venkat et al.'s vector version in the unsymmetric cases and obtains competitive performance for the symmetric cases. We may also obtain a comparable performance with the CUSP library.

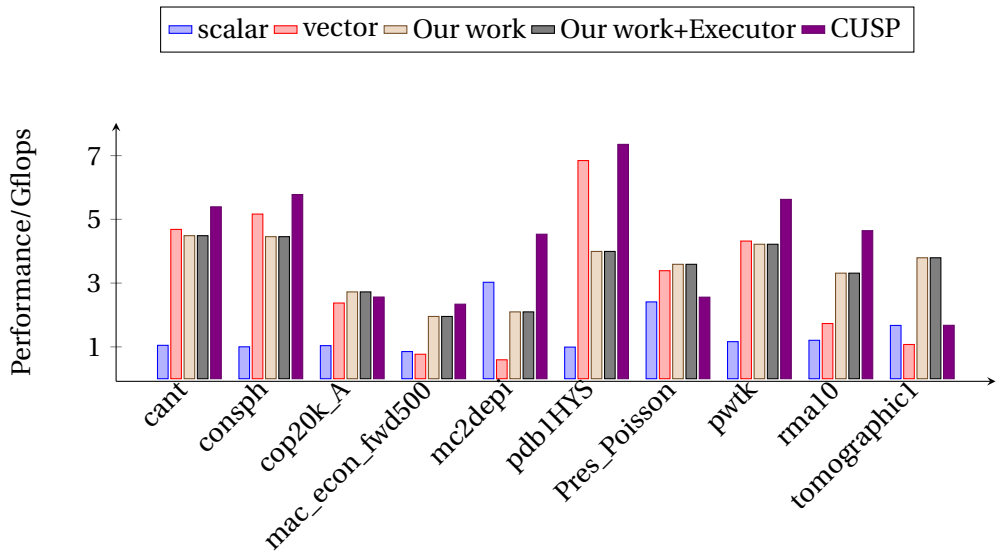


Figure 6.5 – Performance of the CSR SpMV on GPU

In [VHS15], the ELL format is derived from CSR by tiling the dynamic counted loop with the maximum number of nonzero entries in a row. Rows with fewer non-zeros are padded with zero values, implying there will be no early exit statements when parallelizing both loops. It makes their approach effective when most rows have a similar number of non-zeros. Our technique implements a similar idea without data transformation by extending the upper bound of the inner dynamic counted loop to the maximum number of non-zeros, and automatically emitting early exit statements when there are fewer non-zeros in a row, minimizing the number of iterations of the dynamic counted loop. The performance is shown in Figure 6.6 together with that of the CUSP library. A *format_conversion* exception is captured when experimenting the CUSP library with `mac_econ_fwd500`, `mc2depi`, `pwtk` and `tomographic1` while our technique remains applicable on all formats.

Our technique provides comparable or higher performance than the inspector/executor scheme without the associated overhead. Part of the reason why our code falls behind the CUSP library is due to the introduction of atomic operations in the innermost loop which we plan to further imporve in future. Besides, the control overhead caused by our scheme is inevitable, leading to a longer execution time than a library implementation.

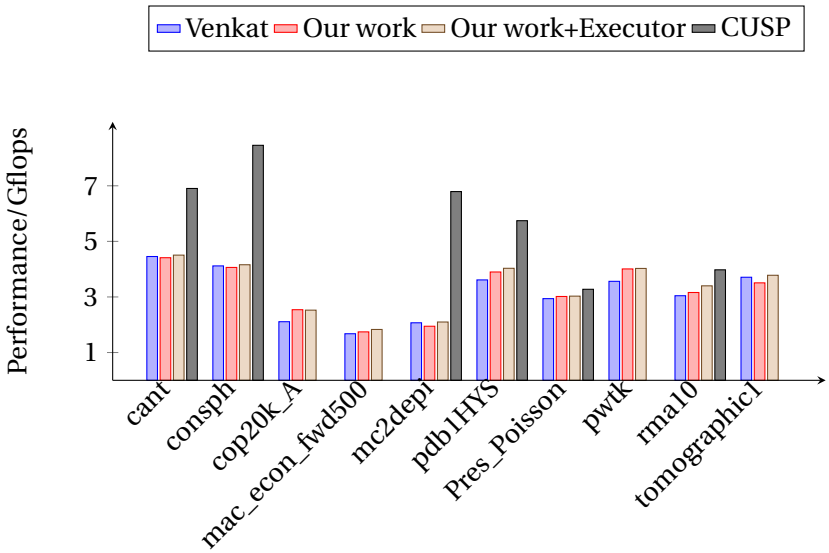


Figure 6.6 – Performance of the ELL SpMV on GPU

6.5 Inspector/Executor

The inspector/executor strategy used in [VHS15] obtains performance gains by optimizing the data layout. Our technique can also apply to the executor of this strategy as a complementary optimization, further improving the performance of the executor. The inspector/executor strategy, however, is not so satisfying as expected for the scalar version of CSR format, since the executor is roughly the same with the original code.

As a result, the performance of our generated code when applying our technique on the CSR executor is also roughly the same with that applying on the orignal code, as shown in Figure 6.5. As a complementary optimization, our technique can speedup the CSR executor by up to 4.6× (from 1.05 Gflops to 4.89 Gflops under `cant` input).

The ELL executor uses a transposed matrix to achieve global memory coalescing, whose efficiency depends heavily on the number of rows that have a similar number of non-zero entries. To get rid of this limitation, our technique may be applied to eliminate the wasted iterations by emitting early exit statements. Experimental results of the ELL executor are shown in Figure 6.6, for which our technique improves the performance by up to 19.7% (from 2.11 Gflops to 2.53 Gflops under `cop20k_A` input).

6.6 Performance on CPU Architectures

We also evaluate our technique on CPU architectures. Unlike generating CUDA code, the original dynamic condition can be taken back when generating OpenMP code on CPU architectures, avoiding the combination of nested bands and the refactoring of the control

flow.

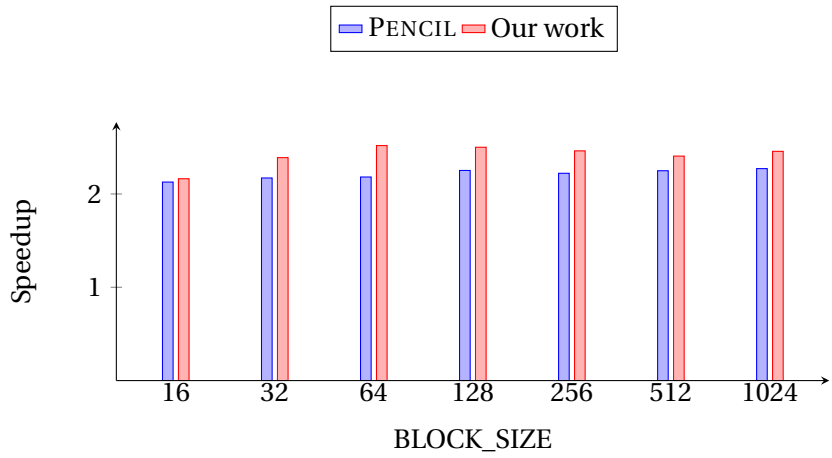


Figure 6.7 – Performance of the HOG descriptor on CPU

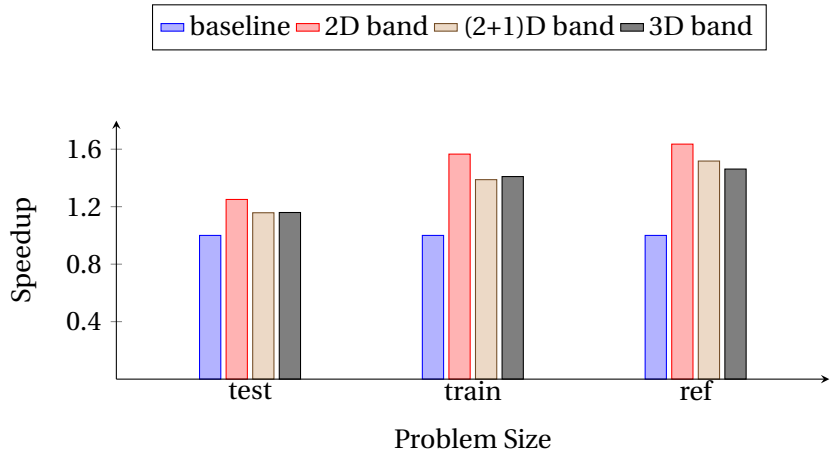


Figure 6.8 – Performance of `equake` on CPU

The performance results are shown in Figures 6.7–6.10. We do not show the performance of dynamic programming examples on CPU architectures since our code generation scheme generates OpenMP code identical with the hand written one. For the remaining benchmarks, our technique enables aggressive loop transformations including tiling, interchange, etc., leading to a better performance when these optimizations are turned on. As the CUSP library is designed for GPU architectures, we only compare the performance of the SpMV code with Venkat et al.'s [VHS15] work.

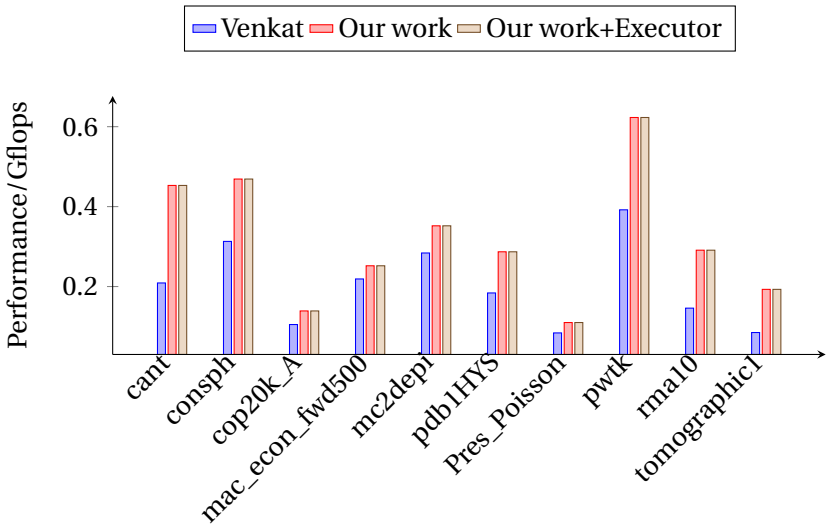


Figure 6.9 – Performance of the CSR SpMV on CPU

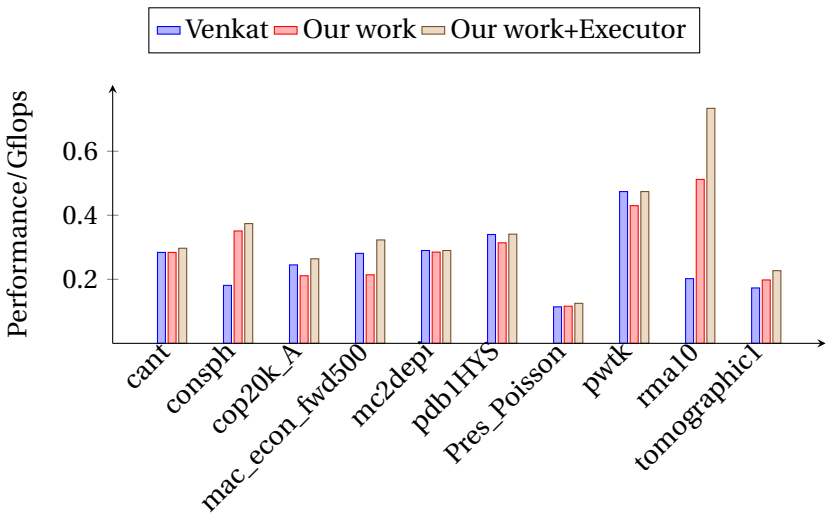


Figure 6.10 – Performance of the ELL SpMV on CPU

Part III

Automating Non-affine Transformations

7 Overlapped Tiling

A large number of efforts have been taken to improve data locality and parallelism for computationally intensive applications especially for iterated stencils, resulting in various loop tiling transformations including simple shapes like rectangular, parallelogram [BHRS08, VCJC+13] and complex shapes like overlapped, split [KBB+07], diamond [BBP17], hexagonal [GCH+14], etc.

The polyhedral framework has been brought to the front scene by its ability to analyze and optimize general-purpose loop nests. Its main scheduling and code generation algorithm remain limited to classical tile shapes however, leading to inefficient wavefront inter-tile parallelism with pipelined startup, while custom solutions with more complex tile shapes exist to exploit inter-tile parallelism along the axes of the iteration domain and data while improving data locality.

There have been several loop tiling techniques specializing on time-iterated stencils, either constrained to constant dependence vectors [KBB+07] or being difficult to extend to other areas like image processing pipelines [BBP17, GCH+14]. Image processing pipelines are a class of computations arising in computer vision and computational photography, consisting of a directed acyclic graph of filtering stages and edges representing dependences between stages. The stages in a pipeline usually exhibit abundant data parallelism but require locality optimization to achieve high performance, making manually exploiting a difficult task.

PolyMage [MVB15] is the state-of-the-art polyhedral compilation framework automatically generating high-performance schedules for such image processing pipelines, benefiting from the full inter-tile parallelism enabled by overlapped tiling [KBB+07]. It takes a DSL inspired by Halide [RKBA+13] as input, computes manually-written-competitive schedules for image processing pipelines, and generates high-performance imperative code. The PolyMage framework implements overlapped tiling by finding bounding hyperplanes of a tile, implying these bounding hyperplanes have to be as inclined as possible to preserve the dependence vectors at all levels of the pipeline, leading to a looser tile shape than expected.

In this part, we intend to implement a schedule-tree-based overlapped tiling technique. Compared with the "DSL+compiler" approach of the PolyhMage framework, we still leverage the PENCIL language and the well-defined polyhedral representation for achieving such non-affine transformation. Due to the introduction of overlapped regions, a statement instance may be assigned multiple execution dates, violating the "single-valuedness" property hold by many previous polyhedral representations. We would show how such violation can be preserved and implemented in the schedule tree representation.

7.1 Background and Motivation

7.1.1 Loop Tiling

Loop tiling has been integrated into polyhedral compilation frameworks, being implemented as a post-scheduling transformation for exploiting data locality and parallelism. Typically, a polyhedral compiler, e.g., PPCG, abstracts the input program as sets and maps defined by systems of affine inequalities, before constructing an affine schedule respecting all dependences carried by statement instances to better exploit data locality and expose parallelism. A follow-up tiling transformation is performed automatically, embedding the computation into a higher dimensional space of tile and point dimensions, but currently limited to classical, rectangular or parallelogram tile shapes.

Decoupling loop tiling from the scheduler may prohibit tile-level concurrent start. An alternative way allowing full inter-tile parallelism involves a tighter coupling of loop tiling and affine scheduling, like the Pluto compiler does by introducing diamond tiling [BBP17]. Tile-level concurrent start along a face of the iteration space is possible if there are no inter-tile dependences parallel to this face, forcing the face to be evicted or linearly independent from the candidate bounding hyperplanes of the scheduler[1].

The schedule found by such evictions could not be better than those found by the standard scheduling algorithm with respect to the dependence distance minimizing cost function, implying a different schedule has to be used for intra-tile parallelism and/or vectorization, complicating the scheduling process and follow-up code generation.

An alternative way to eliminate pipelined startup and drain is to modify the tile shape obtained by existing polyhedral compilation frameworks [KBB+07]. Overlapped tiling is constructed by adding an additional (shaded) region to the left of the tile obtained by existing frameworks, jointing consecutive tiles for exploiting inter-tile parallelism. The shaded regions between consecutive tiles have to be recomputed. Split tiling is obtained by splitting the overlapped tile shape into two sub-tiles, with one being the shaded region and the other consisting of all the remaining points executed in order. Each sub-tile can be executed concurrently with those of neighboring tiles. As an illustrative example, Figure 7.1 shows the comparison between

[1] A hyperplane can also be understood as the higher dimensional analog of a face in 3D space.

different tile shapes for the time-iterated stencil code in Figure 2.2(a).

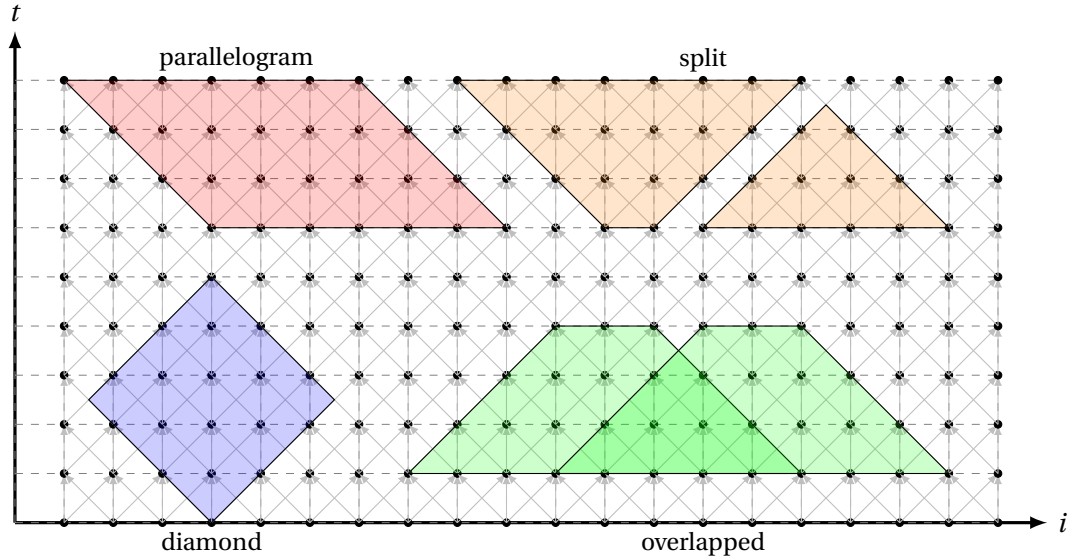


Figure 7.1 – Comparison of different tile shapes

7.1.2 Image Processing Pipelines

Image processing pipelines refer to a set of end-to-end interconnected processing stages conveying an image captured by an image sensor, e.g., a camera or a scanner, to an image renderer like a screen or a printer. Typically, an image processing pipeline can be represented as a directed acyclic graph with each node representing a stage of the computation, related each other by edges for dependence relations.

Image processing pipelines are an important class of computations arising in computer vision, computational photography, medical imaging, etc. They are usually composed of a wide set of operations like point-wise, stencil, up-sampling/down-sampling, exhibiting data parallelism across the pixels of the processed image. While data parallelism exhibited by individual stages is easy to exploit, the high memory bandwidth required by such stages, however, demands locality optimization for achieving high performance.

The PolyMage framework implements an overlapped tiling technique for exploiting data locality for this class of computations, generating high-performance imperative code competitive to those written by experts. Figure 7.2 shows a simple image processing pipeline, and the overlapped tiles constructed by PolyMage on this example is shown in Figure 7.3 with the solid bounds between two tiles representing the bounding faces.

```
for (i=1; i<N; i++)
    A[i] = f(i);
for(i=2; i<N-1; i++)
    B[i] = 0.25*(A[i-1]+2*A[i]+A[i+1]);
for(i=4; i<N-3; i++)
    C[i] = 0.25*(B[i-2]+2*B[i]+B[i+2]);
```

Figure 7.2 – A simple image processing pipeline

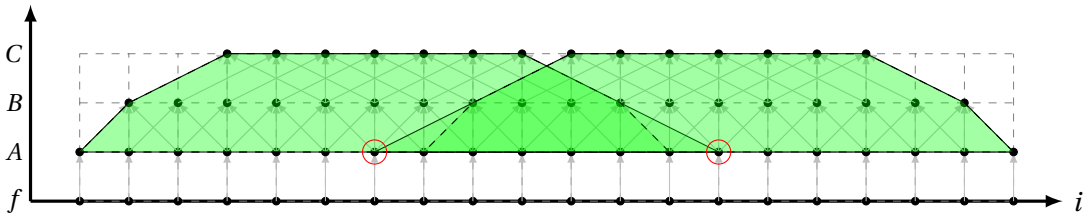


Figure 7.3 – Shaded regions of acute trapezoid tiling

7.1.3 Limitations of the Hyperplane-based Technique

The PolyMage framework is a DSL compiler for addressing the tension between ease of programming and high performance of image processing pipelines. It first leverages a DSL specification to express an image pipeline for extracting its domain-specific knowledge, and then delivers such information to a polyhedral optimizer for automatic parallelization and locality optimization. A follow-up autotuner is used to seek best configurations for achieving high performance.

The significant advance made by the PolyMage framework is the automation of schedule specification, releasing the burden of users from manually specifying the schedules of such image pipelines and attaining high performance by automating a complex tiling shape with some complementary transformations. However, the limitations of such DSL compiler are two-folded.

On the one hand, the PolyMage framework leverages a DSL inspired by Halide [RKBA+13] to express the computations, calling for a domain-specific code generator. Furthermore, the current implementation of the framework misses a backend support of heterogeneous targets.

On the other hand, the PolyMage framework relies on a hyperplane-based approach for exploiting tiling shapes when optimizing data locality for such image pipelines. It constructs schedules for overlapped tiling as follows. The shape of an overlapped tile is determined by checking the dependence vector of each level of the pipeline. To preserve all these producer-consumer relations, PolyMage may have to extend some of the slopes constructed according to these dependence vectors, enforcing a unique bounding face on each side of an overlapped

tile. In the example shown in Figure 7.3, the (dashed) slopes caused by dependences between stages A and B are extended, constructing a looser shaded region between tiles, leading to more redundant computations than one expects.

7.1.4 Our solution

We propose an overlapped tiling technique for eliminating the redundant (circled) points in shaded regions. To construct a tighter overlapped tile, we first fuse the stages in this example and let a general polyhedral framework perform a rectangular tiling on the iteration space regardless of the correctness. We then expand the bounding faces of a tile by taking into consideration the inter-tile dependences, without necessitating further extending the slopes between stages A and B. This can be implemented by modifying the schedule tree intermediate representation [GVC15] in general polyhedral frameworks. As a result, we can construct a bounding face like the dashed slopes shown in Figure 7.3, eliminating the redundant points from shaded regions. Such overlapped tile is referred to as acute trapezoid tile.

One may also construct an overlapped tile by first resorting to the scheduler of a polyhedral compiler, transforming the iteration space into the form shown in Figure 7.4. In this case, a PolyMage-like technique may construct an overlapped tile shape like the solid slopes show, but we can still achieve a tighter shape by (1) resorting to a scheduler to shift the iteration space, (2) performing a rectangular tiling and (3) extending the left bounding face of a tile by operating on the schedule tree representation, still eliminating the redundant points from shaded regions. This shape is referred to as right trapezoid tile.

We design and implement our approach in a source-to-source polyhedral compiler targeting on image processing pipelines written in a general-purpose language. As a result, our method would be neither restricted to a domain-specific language nor does it introduce sophisticated rescheduling and custom code generation in a polyhedral framework. We leverage the schedule tree representation instead, and this allows us to construct tighter tile shapes than the state of the art, minimizing the shaded region of redundant computations across overlapped tiles.

Our technique also goes beyond the state of the art by generating code for both general-purpose multicores and heterogeneous accelerators. We validate its general applicability by conducting additional experiments on iterated stencils, providing a comparison between overlapped tiling and other state-of-the-art techniques.

7.2 Expansion Nodes in Schedule Trees

As we introduced before, the “single-valuedness” property is usually hold by many existing polyhedral representations, preventing such encoding methods from expressing non-affine transformations like overlapped tiling. The reason is the overlapped regions introduced by overlapped tiling require the polyhedral representation may map more than one execution

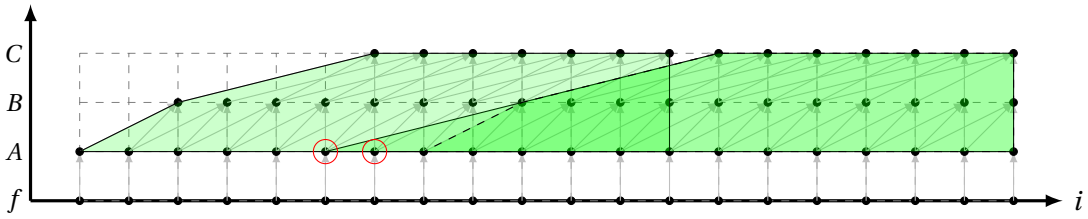


Figure 7.4 – Shaded regions of right trapezoid tiling

dates to a statement instances. Still take Figure 7.3 as an example, all the statement instances of stage A between the two circled points should be scheduled by both tiles, implying they would be assigned two execution dates by the scheduler. This is also the case of all statement instances of stage B in the overlapped region.

We leverage the expansion node in schedule trees for assigning multiple execution dates to a statement instance. In schedule trees, an expansion node is expressed using a named union map, mapping each element of the domain to one or more image in the range. Note that a "domain" here refers to the domain of a map rather than an iteration domain in schedule trees. Similarly, the readers should not be confused the "image" here with that of image processing pipelines. An "image" in this context represents the image of a mapping function.

An expansion node can be generalized with the following named union map

$$\{D(i_1, i_2, ..., i_{n_1}) \rightarrow R(i_1, i_2, ..., i_{n_2}) : f((i_1, i_2, ..., i_{n_1}), (i_1, i_2, ..., i_{n_2}), (p_1, p_2, ..., p_m))\} \tag{7.1}$$

mapping an n_1-dimensional domain statement D to an n_2-dimensional range statement R. One may use an expansion node to map multiple domain statement to the same range statement, implying all the instances of these domain statements are grouped into a single virtual statement. As a result, such statements could always be executed together.

To construct the overlapped regions of overlapped tiling, we can specify the domain and range of an expansion node as the same statement but with a set of specific constraints. The difficulty to implement overlapped tiling is to construct the constraints for such a named union map, not only for assigning multiple execution dates to an instance but also for reducing control overheads. This will be explained in the next section.

7.3 Related Work

Loop tiling was long considered as foreign to the polyhedral model since it could not be easily expressed using an affine function. The Pluto scheduler [BHRS08] is driven by a practical cost function, integrating loop tiling with the polyhedral model. The standard tile shape enabled

by the Pluto scheduling algorithm or its variants is missing concurrent start, limiting the performance of the generated code especially those stencil-based applications.

Overlapped tiling and split tiling [KBB+07] were proposed targeting on concurrent start by modifying the tile shape obtained by a Pluto-like scheduler. The latter should be implemented by either splitting the shaded region of overlapped tiles or introducing more difficult scheduling process for finding different bounding faces of a tile. No implementations of overlapped tiling on general-purpose platforms have been reported though it was implemented in domain-specific compilers for image processing pipelines [MVB15, RKBA+13] or stencil code generator [HPS12] with OpenCL [ZGG+12]. There was also an implementation of split tiling for iterated stencils on GPU architectures [GCK+13]. Comparing with these approaches, our technique covers a wider application domain, improving performance over the state of the art by constructing a much tighter overlapped tile.

Davis et al. [DSO18] proposed to construct the fusion-based and shifting-based overlapped tiles for improving performance of applications involving stencil computations. The image processing pipeline used in their work only involves stencil operations but not sampling nor histogram operations. Our work covers all the basic operations involved in image processing pipelines. The work of Davis et al. was neither trying to minimize the redundant computation of overlapped tiling.

Bondhugula et al. [BPB12, BBP17] proposed a generalizing formalism for diamond tiling in the polyhedral model by introducing a rescheduling step in the Pluto compiler. There has been a great amount of work [EM90, GV15, MHLK17, OG09, SGM+15, SSPS11] reported on the evaluation of diamond tiling. It was also generalized to handle iterated stencils defined over periodic data domain with index set splitting technique [BBC+14] and Lattice-Boltzmann method [PAVB15]. Unlike overlapped tiling and split tiling, diamond tiling may work with arbitrary affine dependences. The introduction of scheduling to find tiling dimensions does not only complicate the scheduling but also increase the code generation time in practice. We show in our evaluation section that our technique could achieve competitive performance on iterated stencils with diamond tiling by carefully selecting tile sizes. Our technique is also applicable to image processing pipelines.

Hybrid hexagonal/classical tiling was proposed by Grosser et al. [GCH+14] to exploit full inter-tile parallelism of iterated stencils on GPU architectures. It can be seen as a generalization of diamond tiling, allowing for partial concurrent start by constructing a hexagonal tile shape along the time and the first space dimension and classical tiling along the other space dimensions. Grosser et al. [GVCS14] also show a comparison between diamond tiling and hexagonal tiling. We compare our technique with hexagonal tiling in our experiments on GPU architectures.

By revisiting overlapped tiling in polyhedral compilation frameworks especially for image processing pipelines, we construct a much tighter overlapped tile shape for improving the performance of such applications. Halide [RKAP+12, RKBA+13] is a domain-specific language

for image processing pipelines, decoupling algorithms from schedules for easy optimizations for such benchmarks, allowing users to experiment with schedules without touching the algorithms. Manually or autotuning approaches [AKV+14] to finding schedules usually takes a long time to facilitate fascinating performance. The polyhedral model is a promising solution to automatically search schedules for image processing pipelines by integrating with transformations like overlapped tiling, fusion, scratchpad allocation, etc.

8 Acute Trapezoid Tiling and Right Trapezoid Tiling

Overlapped tiling is an efficient tiling technique allowing for tile-level concurrent start. As we described above, one may choose to obtain an acute or a right trapezoid tile by fusing or shifting first, with both achievable by operating on the schedule tree representation.

8.1 Representations in Schedule Trees

As we would leverage schedule trees to express both acute trapezoid tiles and right trapezoid tiles, let us recall the schedule tree representation and explain how we can make use of it. In polyhedral compilation, schedules in the polyhedral model are used to define the execution order of programs, including both the original and those generated by scheduling algorithms like the Pluto scheduler or its variants. A schedule has in nature the form of a tree, making an explicit tree representation have the same expressiveness with previous encoding methods but simplify the implementation of non-polyhedral operations.

A statement instance is expressed by a named multi-dimensional vector with the name identifying the statement and the coordinates corresponding to iteration variables of the enclosing loops. The collection of all statement instances, i.e., the iteration domain, is expressed using Presburger formulas [PW94c], retained in a domain node. For example, the iteration domain of the code shown in Figure 7.2 can be expressed as

$$\{S_A(i) : 1 \leq i < N; S_B(i) : 2 \leq i < N-1; S_C(i) : 4 \leq i < N-3\}$$

with loop boundaries included.

A statement instance is also mapped to a multi-dimensional logical execution date [Fea92b] for defining its lexicographic execution order. Such mapping is referred to as a schedule, expressed by a piecewise multi-dimensional quasi-affine function over the iteration domain

and contained in a band node. A band node is derived from tilable band in the Pluto framework [BHRS08], defining permutability and/or parallelism properties on a group of statements as well. A rectangular tiling regardless of the correctness of the example in Figure 7.2 is written as

$$[\{S_A(i) \to (i/T); S_B(i) \to (i/T); S_C(i) \to (i/T)\}, \{S_A(i) \to (i); S_B(i) \to (i); S_C(i) \to (i)\}]$$

with the former piece representing tile loops (iterating among tiles) and the latter representing point loops (iterating within tiles).

A filter node selects a subset of statement instances introduced by an outer domain/filter node. Filter nodes usually appear as children of a sequence/set node expressing a given/arbitrary order on its children. As an illustrative example, Figure 8.1 is the original schedule tree of the example shown in Figure 7.2, indicating each filter node selects a subset (a stage) of the domain node. The sequence node defines the three stages should be executed in order, followed by a loop iterating over the iteration space of each stage.

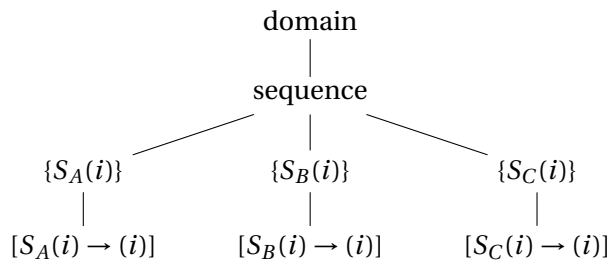


Figure 8.1 – The original schedule tree of the code in Figure 7.2

There have been many other node types existing in schedule trees [GVC15], among which we focus on the expansion node as introduced in subsection 7.2. An expansion node can expand a statement instance to one or more instances, constituting a new set of statement instances to be scheduled by the schedule tree. A standard loop tiling would partition the iteration space into smaller blocks, each of which is disjoint with each other, making it difficult to construct an overlapped tile without an expansion node. With an expansion node, we are free to choose one statement instance of a stage in a tile and expand it to as many instances as we expect. These expanded statement instances would joint with those of neighboring tiles, resulting in overlapped tiles over the whole iteration space.

8.2 Acute Trapezoid Tiling

Let us first consider implementing overlapped tiling by modifying both sides of a tile. Figure 8.2 shows the result of rectangular tiling on the iteration space of the example listed in Figure 7.2 regardless of the correctness. Such rectangular tile can be obtained by first strip-mining

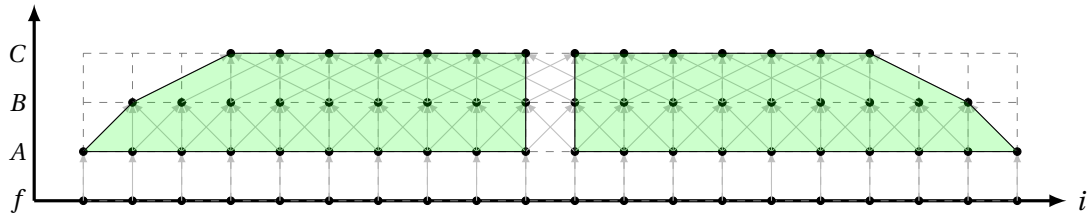


Figure 8.2 – Rectangular tiling regardless of the correctness

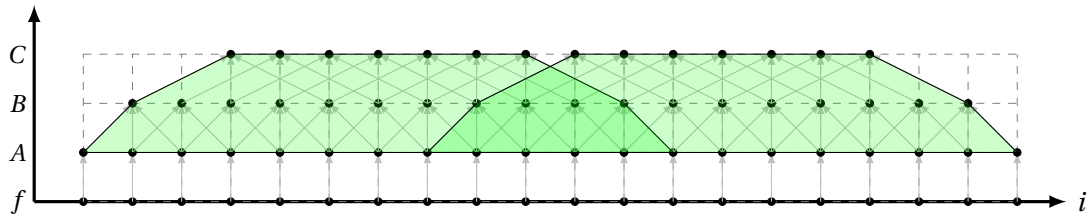


Figure 8.3 – Acute trapezoid tiling

[BGS94] each stage and then fusing the tile loops of each stage. For the sake of simplicity, we call the tile on the left side as L-tile, and the tile on the right side as R-tile.

We may first handle the right tile boundary of L-tile. According to the dependence vector between stage C and B, the last two points of stage C executed by L-tile depend on the first two points of stage B executed by R-tile. As a result, we need to expand the domain elements of stage B executed by L-tile to a set of elements consisting of all the original points executed by L-tile plus the first two elements executed by R-tile. This can be achieved by introducing an expansion node below the schedule of stage B.

In the same way, the last point of stage B executed by L-tile depends on the first point of stage A executed by R-tile. Note that the last point of stage B executed by L-tile should be the second element executed by R-tile due to the introduction of expansion node on the schedule of stage B. The shifting of the last point executed by a tile caused by introducing expansion nodes is called dependence propagation. The images of the expansion node introduced to the schedule of stage A should therefore include all the points of stage A covered by L-tile plus its first three points of R-tile. Finally, we obtain an overlapped tile for L-tile, enforcing its correctness.

We can expand the left tile boundary of R-tile for overlapped tiling in a similar way. The images of the expansion node of stage B consist of the original points plus the last two elements executed by L-tile, and those images of stage A should plus the last three elements executed by L-tile due to dependence propagation. Figure 8.3 shows the acute trapezoid tiling generated by expanding both boundaries of a tile. For each tile other than the first and the last, one may have to expand both the left and right boundaries.

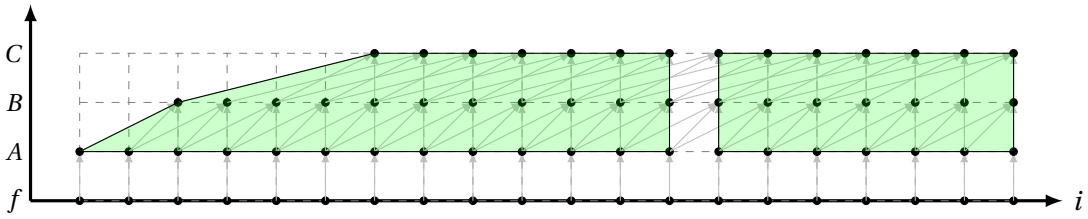


Figure 8.4 – Rectangular tiling after shifting

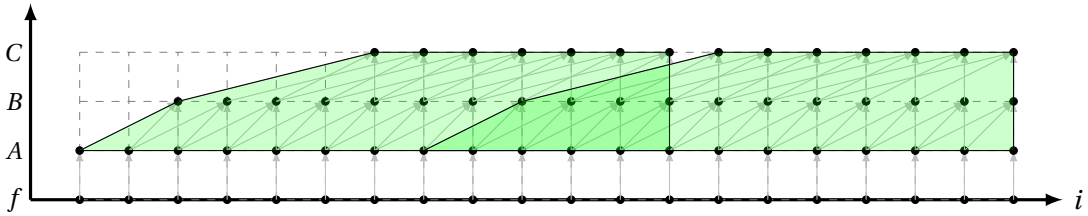


Figure 8.5 – Right trapezoid tiling

We force to insert expansion nodes to the schedule of stages in the top-down manner due to the dependence propagation issue, by which we mean the output stage referencing the live-out sets of a stage group (probably after fusion) should be considered first, followed by the stage it depends on and so on.

8.3 Right Trapezoid Tiling

Decoupling tiling transformation from the scheduler is the usual way tools aiming at a greater coverage of benchmarks–such as PPCG or Pluto follow. Tiling is performed after a new schedule is generated by these tools with respect to dependence distance minimizing cost function. Such scheduling algorithms would generate a new schedule by shifting the iteration domain of stages B and C, followed by rectangular tiling on the transformed iteration space, as shown in Figure 8.4.

We still let L-tile represent the tile on the left and R-tile for the right. Unlike the PolyMage framework, we only need to expand the left boundary of a tile. Considering R-tile, the first point of stage C depends on two elements (the last but one and last but three) of stage B in L-tile, and the second point of stage C depends on another two elements (the last and last but two) of stage B. The left boundary of R-tile on stage B should be shifted to left by four points.

We next turn into stage B. Due to the dependence vector between stage B and A, the images of the expansion node introduced to the schedule of stage A should be expanded to left by two points, forming a final expansion by shifting to left by six points caused by dependence propagation from the above level. Finally, we obtain a right trapezoid tile as shown in Figure 8.5.

8.4 Schedule Generation

We can now generate schedules for both acute and right trapezoid tiling. Considering acute trapezoid tiling, strip-mining is performed on each stage, followed by fusing the tile loops iterating among tiles. Correctness is enforced by introducing expansion nodes below the point loops of stage A and B iterating within a tile, with boundaries updated as explained in the last subsection. We finally get a schedule tree for acute trapezoid tiling as shown in Figure 8.6.

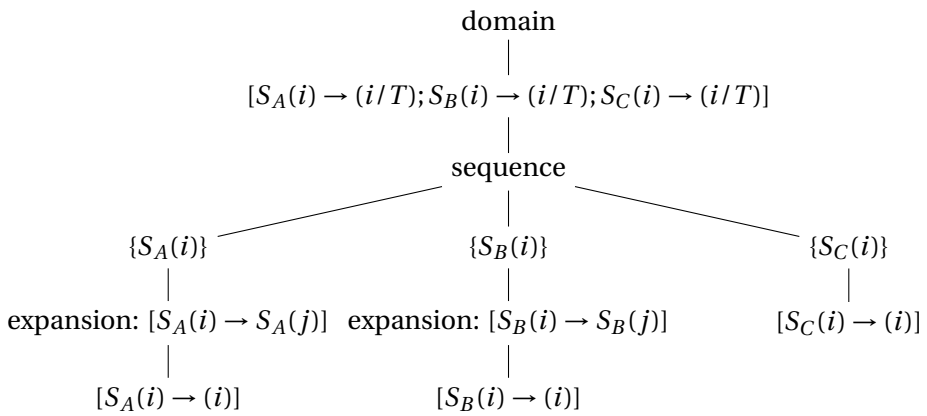


Figure 8.6 – Schedule tree of acute trapezoid tiling

If we let a scheduling algorithm transform the iteration space of the pipeline, we may get a shifted schedule before tiling, leading to a two-dimensional tilable band. However, we need to separate the point loops from the band, making it possible for introducing expansion nodes to stages A and B, as the generated schedule tree shown in Figure 8.7.

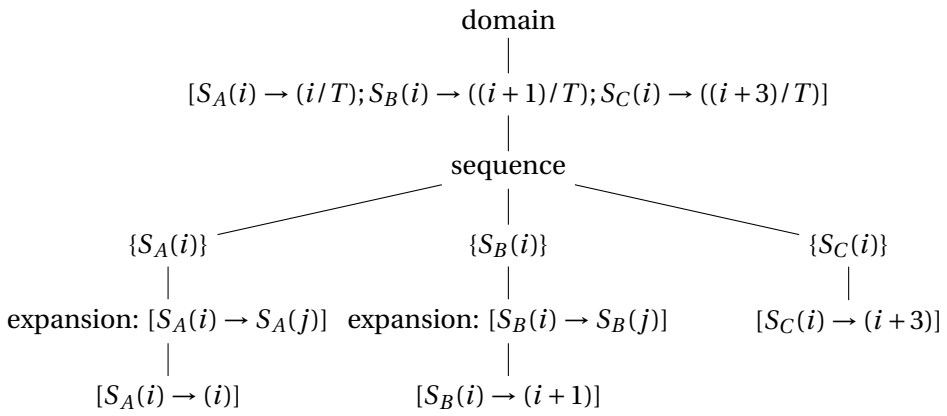


Figure 8.7 – Schedule tree of right trapezoid tiling

The code generator of a polyhedral compilation framework can take these schedule trees for code generation, with overlapped tiling enabled in the generated code.

8.5 Removal of Control Overheads

Let us still consider the stage B of the code in Figure 7.2 and suppose the tile size be T. An expansion node in schedule trees is used to map each domain element of a domain/filter node to one or more images, forming a wider set of elements scheduled by the domain/filter node. Expanding the point loop may change a rectangular/parallelogram tile obtained from existing polyhedral compilation frameworks into an overlapped tile, avoiding necessitating a scheduling algorithm like the PolyMage framework does. We may therefore insert an expansion node below a filter node representing the point loop as shown in Figure 8.6 and 8.7. An expansion node here is represented as a map expanding the original domain $S_B(i)$ to its image set $S_B(j)$.

Introduced for iterating the images of the expansion node, j is an unbounded parameter in the schedule tree, producing an unbounded domain for the subtree of stage B. One may take into consideration the (upper and lower) bounds of j by adding the original bounds on all elements of stage B, guaranteeing the images of the expansion node would not exceed the original bounds. We can write it formally as

$$lb \leq j \leq ub \tag{8.1}$$

where lb and ub represent the lower and upper bounds.

This condition alone cannot produce an overlapped tile, as the boundaries of a tile are still remain unchanged. To expand the boundaries of a rectangular/parallelogram tile as explained in previous subsections, we let $\lfloor (i/T) \rfloor$ denote the greatest integer less than or equal to its parameter, representing the tile number in our case. The original points executed by the $(\lfloor (i/T) \rfloor + 1)$-$th$ tile ($\lfloor (i/T) \rfloor$ is 0 for the first tile) can be expressed by

$$T \times \lfloor (i/T) \rfloor \leq j \leq T \times (\lfloor (i/T) \rfloor + 1) - 1 \tag{8.2}$$

We can change it into

$$T \times \lfloor (i/T) \rfloor - l \leq j \leq T \times (\lfloor (i/T) \rfloor + 1) - 1 + r \tag{8.3}$$

where l represents the number of expanded points introduced by expansion nodes to the left boundary, and r to the right boundary. In practice, l or r may be 0, e.g., right trapezoid tiling.

Expanding the first point of a stage in a tile is straightforward, simplifying schedule transformations when changing a rectangular/parallelogram tile into an overlapped one. We find, however, that the selection of the point from which the images of an expansion node should be generated may have heavy impact on the control overheads in generated code.

One may obtain a bounded map representing an expansion node by conjuncting constraints

(8.1) and (8.3), implying the point at $T+2$ for L-tile and the point at T for R-tile are selected for expansion, isolating a partial tile (L-tile) from full tiles (R-tile and all the remaining if they exist)[1] and introducing more control overheads in generated code. This isolation may lead to a performance degradation when there are much more stages in a group.

A better solution to remove control overheads in generated code is integrating partial tiles with full tiles. Each starting point of stage B executed by a full (other than the first) tile can be expressed as $T \times \lfloor (i/T) \rfloor$, while $T \times \lfloor (i/T) \rfloor + 2$ ($T \times \lfloor (i/T) \rfloor$ is equal to 0) being used for the partial (the first) tile. One is free to choose any point of a stage covered by a tile for generating images of expansion nodes without changing the meaning of the generated code, implying one may choose any point other than $T \times \lfloor (i/T) \rfloor$ in a full tile. We thus choose the point at $T \times \lfloor (i/T) \rfloor + 2$ as starting point for full tiles, enforcing the uniqueness with the partial tile and removing control overheads.

Finally, one may write another condition constraining the selection of starting point of full tiles as

$$i = T \times \lfloor (i/T) \rfloor + s \tag{8.4}$$

where s represents the shifting of the starting point in each tile due to the bounds on the whole domain. We may finally obtain an expansion node by constraining the map of an expansion node with a set of conditions consisting of (8.1), (8.3) and (8.4) as follows.

$$\{S_B(i) \rightarrow S_B(j) : (8.1) \wedge (8.3) \wedge (8.4)\} \tag{8.5}$$

Similarly, we may also obtain the map of expansion nodes like (8.5) and the schedule tree as shown in Figure 8.7, with r in (8.3) being equal to 0, removing control overheads for right trapezoid tiling.

8.6 Comparing the Two Trapezoid Tile Shapes

Given a schedule tree with expansion nodes written as (8.5), one may obtain an acute trapezoid tile by first fusing the stages of an image processing pipeline or a right trapezoid tile by first shifting the iteration space. One may distinguish the difference between the two trapezoid tile shapes by comparing Figure 8.3 and Figure 8.5.

Apart from the shape, the two tile shapes also differ from each other with respect to data locality. For the sake of simplicity, we first show the generated code with different tile shapes of the example in Figure 7.2 in Figure 8.8 and Figure 8.9.

[1] A full tile is completely contained in the iteration space while a partial one is not but has a non-empty intersection with the iteration space [KRR+07].

In Figure 8.8, the point loops of individual stages are distributed. In other words, the tiles of individual stages are executed one after another in an acute trapezoid tile shape. On the contrary, the point loops of these stages are fused in Figure 8.9, minimizing the intra-tile producer-consumer relation distances.

```
for (c0=0; c0<N/T+1; c0++){
  for (c1=max(T*c0-3,1);c1<min(T*(c0+1)+3,N); c1++)
    A[c1] = f(c1);
  for (c1=max(T*c0-2,2);c1<min(T*(c0+1)+2,N-1); c1++)
    B[c1-1] = 0.25*(A[c1-1]+A[c1]+A[c1+1]);
  for (c1=max(T*c0,4);c1<min(T*(c0+1),N-3); c1++)
    C[c1-1] = 0.25*(B[c1-2]+A[c1]+A[c1+2]);
}
```

Figure 8.8 – Code generated by acute trapezoid tiling

```
for (c0=0; c0<N/T+1; c0++){
  for (c1=max(T*c0-6,1); c1<min(T*(c0+1),N); c1++){
    A[c1] = f(c1);
    if(c1>=3){
      B[c1-1] = 0.25*(A[c1-2]+2*A[c1-1]+ A[c1]);
      if(c1>=7)
        C[c1-3] = 0.25*(B[c1-5]+B[c1-3]+B[c1-1]);
    }
  }
}
```

Figure 8.9 – Code generated by right trapezoid tiling

The right trapezoid tile should be preferred as it holds a better intra-tile data locality than the acute one. However, such data locality has a very little impact on the performance of the generated code when targeting on image processing pipelines since the tile height is usually not large enough for benefiting from the former. Iterated stencils may exploit such data locality when the tile size along the time dimension is large enough, as we will explain in the experiment.

8.7 Handling Multi-statement/-dimensional Cases

8.7.1 Multiple Statements

The ability to handle multiple statements is important since such cases may happen in practice, as we will show in the evaluation. Suppose we have a similar schedule tree as shown in Figure 8.6 and let the filter node of stage B consist of two statements, S_{B_1} and S_{B_2}. For the sake of simplicity, we do not show the context in expansion node of stage A.

Following the cases of single statement, we schedule S_{B_1} and S_{B_2} as a macro statement S_B, fusing the tile loop with those of stage A and C, and obtain a schedule tree regardless of the correctness similar to Figure 8.6. An expansion node is allowed to insert right underneath the filter node consisting of S_{B_1} and S_{B_2}, followed by a band node representing the map of their point loops and its child sequence node defining their execution order. We finally obtain a schedule tree as shown in Figure 8.10. Right trapezoid tiling can be handled in the same way.

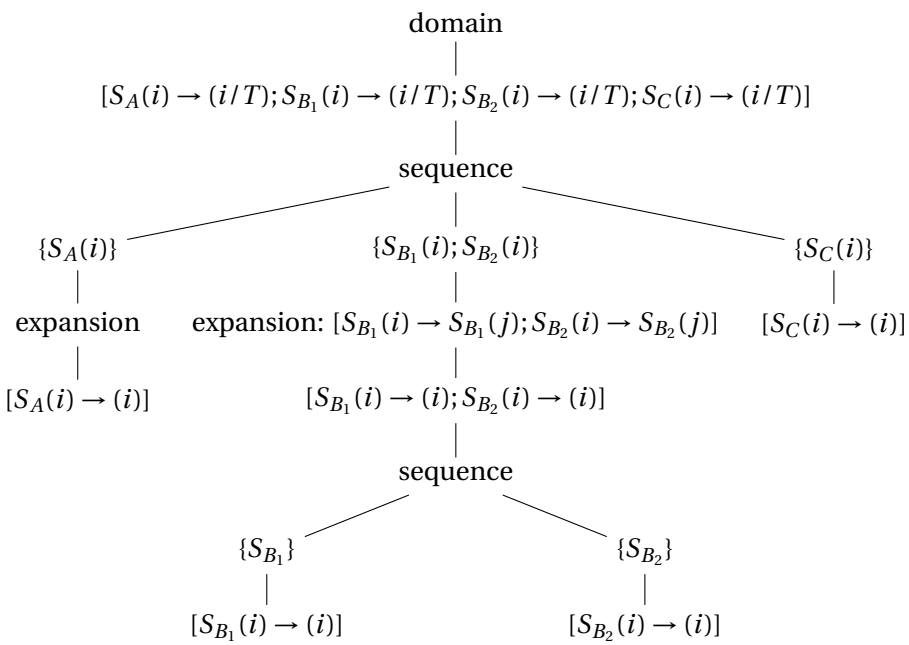


Figure 8.10 – Schedule tree of multiple statements

8.7.2 Multi-dimensional Statements

Considering the cases of multi-dimensional statements, we are allowed to handle a single dimension each time without impacting on other dimensions, implying that handling the cases of multi-dimensional statements is a process of invoking the cases of one-dimensional statement recurrently, with both acute and right trapezoid tiling cases being taken into account.

8.8 Complementary Transformations

One of the main purposes of our technique is to enhance the performance of image processing pipelines by optimizing overlapped tiling in polyhedral compilation frameworks. Following the domain-specific code generator PolyMage, we may also need some other transformations for further improving the performance of such benchmarks.

On the other hand, to cover a wider coverage of benchmarks, we also target on iterated stencil code. We design our technique to handle all cases of iterated stencil code, including multi-dimensional cases and multiple statements as explained in the last subsection. Exploiting full-dimensional parallelism in practice may not be necessary, especially when targeting multi-dimensional cases on general-purpose multicores for single-thread performance. We therefore also need to take it into consideration.

8.8.1 Alignment and Scaling

Constructing overlapped tiling is allowed only in the case of constant dependence vectors, making it not straightforward to exploit inter-tile parallelism in practice when heterogeneous stages exhibiting different dimensions and/or complex access patterns are grouped together. Alignment and scaling of stages can be introduced, following the PolyMage framework, to achieve near-neighbor dependences, changing the dependence vectors into constant.

Alignment can be achieved by introducing a scalar dimension in dependence vectors, followed by shifting among dimensions for eliminating non-constant dependences. Up-sampling and down-sampling are introduced for scaling schedules of stages appropriately, obtained by multiplying a scaling factor for each stage in a group.

8.8.2 Fusion

Loop fusion is an important transformation implemented by polyhedral compilers for exploiting data locality. Benefiting from alignment and scaling, some stages of the pipeline may be fused together, creating opportunities for exploiting overlapped tiling across more stages.

One can make full use of the fusion heuristic adopted by a polyhedral compiler by setting compilation options, but it may not be good enough for image processing pipelines even with the aggressive fusion heuristic. The criteria the PolyMage code generator provides is fusing a successor stage with its only child when it has only one child by viewing the pipeline as a directed acyclic graph consisting of nodes representing stages and edges denoting dependences between stages, followed by iteratively attempts for fusing opportunities until no fusion can be found. We reimplement this heuristic in our technique.

8.8.3 Reducing Memory Footprint

Loop fusion transforms the pipeline into several groups, each of which consists of a set of intermediate stages and an output stage, requiring storage allocation optimization for improving performance.

Those values produced by intermediate stages are only used within a tile, implying they can be discarded when they are not live after the computation of the tile. These intermediate values can therefore be allocated in small scratchpad memory rather than full buffers, leading to better locality and improving the performance when integrating with overlapped tiling and the transformations mentioned above. Indexing expressions generated for such scratchpads can be determined according to the conditions defined in expansion nodes.

8.8.4 Hybrid Tiling

When targeting on multi-dimensional iterated stencils, we are allowed to restrict overlapped tiling only to the time dimension and a subset of space dimensions, leveraging existing polyhedral frameworks to perform rectangular/parallelogram tiling on the remaining space dimensions, just like diamond tiling [BBP17] and hybrid hexagonal/classical tiling [GCH+14]. This lower dimensional overlapped tiling may change the tile shapes for a multi-dimensional case, as we show in Figure 8.11 comparing the difference between a full and partial dimensional overlapped tile shapes for $2d$ stencil code. A full dimensional overlapped tiling would form a base of a pyramid, extending along both dimensions of the space, while a partial one only overlaps along one dimension.

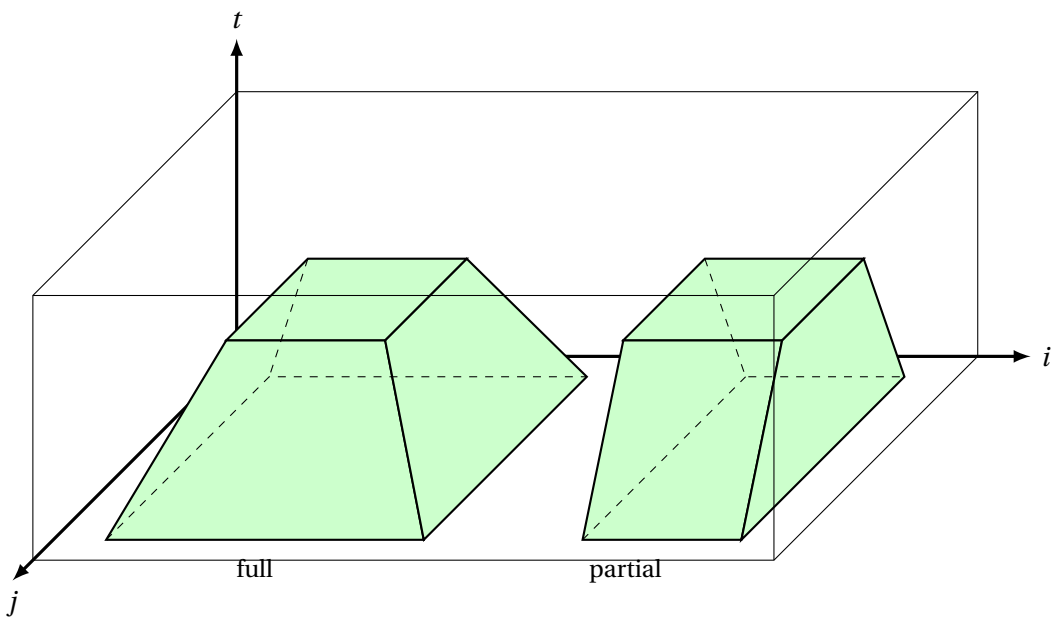


Figure 8.11 – Full and partial dimensional overlapped tiling

9 Evaluations

We conduct experiments on both the PolyMage benchmarks and representative iterated stencils. The image processing pipelines covered by the PolyMage benchmarks are extracted from the Halide benchmarks, varying widely in structure and complexity.

9.1 Experimental Setup

We implement both trapezoid tiling techniques as well as the complementary transformations explained in Section 8.8 in PPCG, a polyhedral compiler that exploits parallelism and data locality of programs automatically. All transformations are applied automatically by default when passing *--acute* and *--right* flags to PPCG for switching between acute and right trapezoid tiling. The PPCG version we use is `ppcg-0.07-26-g236d559`. PPCG can take C programs as input, automatically generating OpenMP code on general-purpose multicores and CUDA code on heterogeneous accelerators.

The experiments are conducted on a 32-core, dual-socket workstation with an NVIDIA Quadro P6000 GPU. Each CPU is a 2.10GHz 16-core Intel Xeon(R) E5-2683 v4. We use the icc compiler (`18.0.1`) from Intel Parallel Studio XE 2018, with the flags *-fast -qopenmp*. CUDA code is compiled with the NVIDIA CUDA (`9.1.95`) toolkit with the *-O3* optimization flag. Each benchmark is executed 11 times, of which the first run is discarded and the average of the remaining is recorded.

9.2 Image Processing Pipelines

The PolyMage framework takes a DSL as input and generates both naïve and optimized OpenMP codes. A naïve version is generated by PolyMage without schedule transformations and overlapped tiling, of which the sequential code is used as the baseline and also as the input PPCG. The image processing benchmarks used in our experiments are listed in Table 9.1, together with the number of stages and the actual execution times of each baseline. One

may obtain the execution time of each case by multiplying the factors shown in Figure 9.1-9.7 and 9.11.

Table 9.1 – Summary of the PolyMage benchmark

Benchmark	Stages	CPU execution time (ms)			Speedup over PolyMage
		naïve (1 core)	PolyMage (32 cores)	Our work (32 cores)	(32 cores)
Bilateral Grid	7	66.01	5.57	5.41	1.03
Camera Pipeline	32	116.32	5.95	5.87	1.01
Harris Corner Detection	11	246.88	5.10	5.10	1.00
Local Laplacian Filter	99	480.48	35.35	27.08	1.31
Multiscale Interpolation	49	209.10	20.07	16.44	1.22
Pyramid Blending	44	350.49	17.18	15.41	1.11
Unsharp Mask	4	142.16	5.01	3.68	1.36

Benchmark	Stages	GPU execution time (ms)		Speedup over PPCG
		PPCG	Our work	
Bilateral Grid	7	4.67	4.25	1.10
Camera Pipeline	32	4.29	2.18	1.97
Harris Corner Detection	11	1.89	1.07	1.77
Local Laplacian Filter	99	16.73	11.12	1.50
Multiscale Interpolation	49	12.86	7.76	1.66
Pyramid Blending	44	8.33	5.78	1.44
Unsharp Mask	4	2.17	1.29	1.68

We compare the performance of our generated code with both the naïve and optimized version generated by PolyMage. Table 9.1 also shows the speedup of our code over the optimized version of PolyMage by only differentiating the overlapped tiling shapes, validating the effectiveness of our tiling techniques. We also show the performance of the Halide[1] manual schedule written by experts and automatic scheduling algorithm [MAS+16] inspired from PolyMage. The OpenCV version we use is 2.4.9.1. We set the overlapped tile sizes the same as the optimized version of the PolyMage framework. Acute trapezoid tiling is used in our experiment, as the data locality exploited by right trapezoid tiling does not make sense in this case, as we have already explained.

9.2.1 Bilateral Grid

Bilateral Grid [CPD07, PKT+09] is a data structure enabling fast edge-aware image processing, localizing operations involved including bilateral filtering, edge-aware painting, local histogram equalization, etc. This benchmark smoothes images while preserving their edges by first constructing a bilateral grid and then sampling the grid along each dimension, producing

[1] Version: commit: 8c23a1970faba9b06bf7145d2653618fb978479e.

a pipeline consisting of a histogram operation and some stencil and sampling operations.

The generated code of our technique is composed of three groups, with the first consisting of the histogram operations, the second constructed by fusing all the stencil and sampling stages, and the final reduction operations. The optimized PolyMage code and the automatically generated code of Halide also claimed having done this way, but PolyMage only fuses the stencil and sampling stages into two groups in practice. The manual schedule of Halide fuses transforms the pipeline into a 3-stage kernel, fusing stencil and sampling stages with the final reduction. Our technique is better than both the PolyMage framework and the automatic scheduling algorithm of Halide but falls behind the hand written schedule of Halide, as shown in Figure 9.1.

To validate the performance gap between our generated code and the manual schedule of Halide is not due to overlapped tiling, we manually tune the fusion strategy of our generated code to keep in line with the latter and evaluate the performance of such tuned code, referred to as "tuned" in the figure. As a result, the performance outperforms the manual scheduling strategy of Halide.

With regard to fusion, the heuristic used in our framework uses a greedy algorithm for fusing stages in image processing pipelines, leaving some fusion cases unevaluated during the optimization that may be exploited by hand. We plan to further improve our fusion heuristic in future.

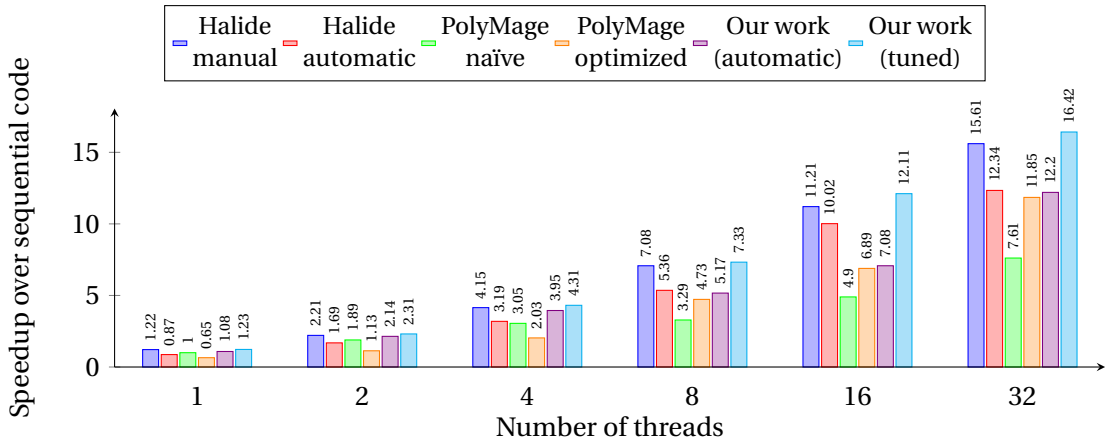


Figure 9.1 – Performance of Bilateral Grid on CPU

9.2.2 Camera Pipeline

Targeting on transforming the raw data captured by a camera sensor into a color image, Camera Pipeline performs a sequence of steps including hot pixel suppression, demosaicking, color correction, and global tone mapping. These tasks are implemented by stencil-like stages consisting of multiple statements and table lookups.

Our technique performs a similar fusion on the benchmark like PolyMage, grouping all stencil-like stages except the lookup table stages. PPCG also performs an automatic inlining of some of the stencil-like stages, outperforming the optimized version of PolyMage slightly. Our technique also beats the automatic scheduling algorithm of Halide, falling a little behind of the manually written schedule because the latter also benefits from unrolling optimization. The performance comparison is depicted in Figure 9.2.

Like the Bilateral Grid benchmark, we may also "fine-tune" the generated code of our framework to eliminate impact of such unrolling optimization, thereby catching up the result of manual scheduling method used in Halide. Complementing the unrolling optimization in our framework is our next research direction.

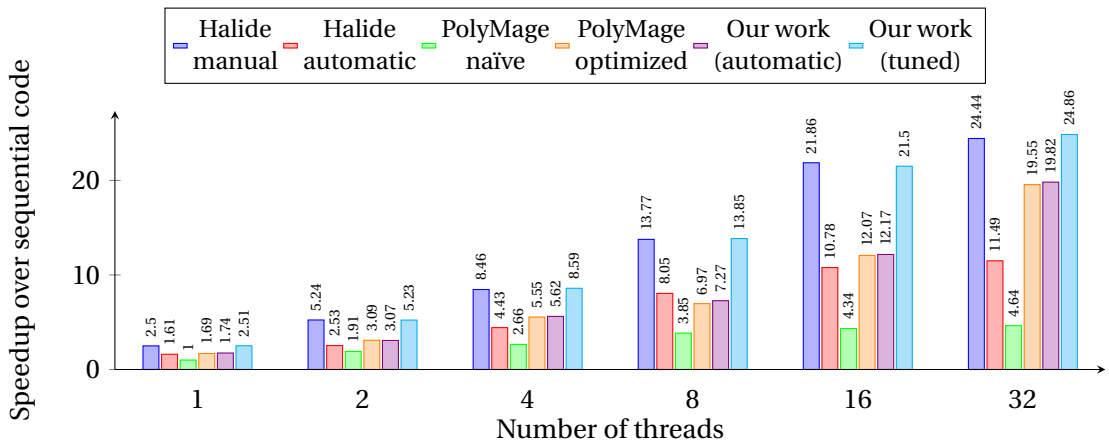


Figure 9.2 – Performance of Camera Pipeline on CPU

9.2.3 Harris Corner Detection

Corner detection is an approach to extracting features and contents of interest from an image. Harris Corner Detection [HS88] is a widely used method to realize corner detection, taking the differential of the corner score into account with reference to direction directly. It has been proved to be more accurate than previous implementations, being improved and adopted in various applications.

The PolyMage framework fuses all stages of the pipeline into one group, benefiting from inlining all point-wise operations which we also implemented in our framework. A follow-up fusion is performed on the inlined version, grouping all stages of the pipeline together. We obtain a similar performance with the optimized PolyMage code as shown in Figure 9.3, beating the hand written schedule of Halide. The automatic scheduling algorithm also follows the PolyMage framework, performing similar to PolyMage and our technique.

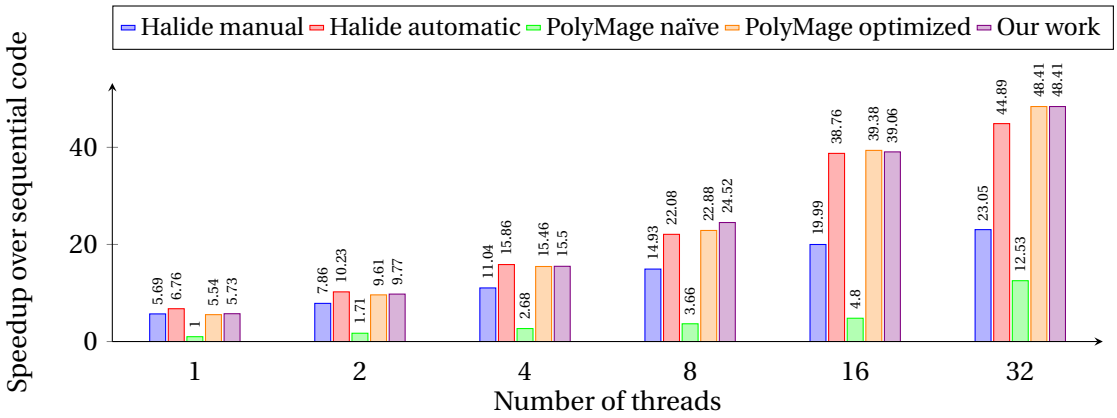


Figure 9.3 – Performance of Harris Corner Detection on CPU

9.2.4 Local Laplacian Filter

Local Laplacian Filter [PHK15, APH+14] is an approach to addressing limited scalability and producing edge-aware filters for manipulating images at multiple scales. The benchmark is the most complicated among all PolyMage benchmarks, consisting of 99 stages and involving both sampling and data-dependent operations.

The experimental result of the benchmark is shown in Figure 9.4. Following the fusion criteria of the PolyMage framework, we fuse some of the stages of the pipeline and generate a much tighter overlapped tile shape than PolyMage, leading to a better performance over the optimized version of PolyMage. The schedules generated by hand and automatic scheduling algorithm of Halide fall behind due to the missing of effective fusion and overlapped tiling.

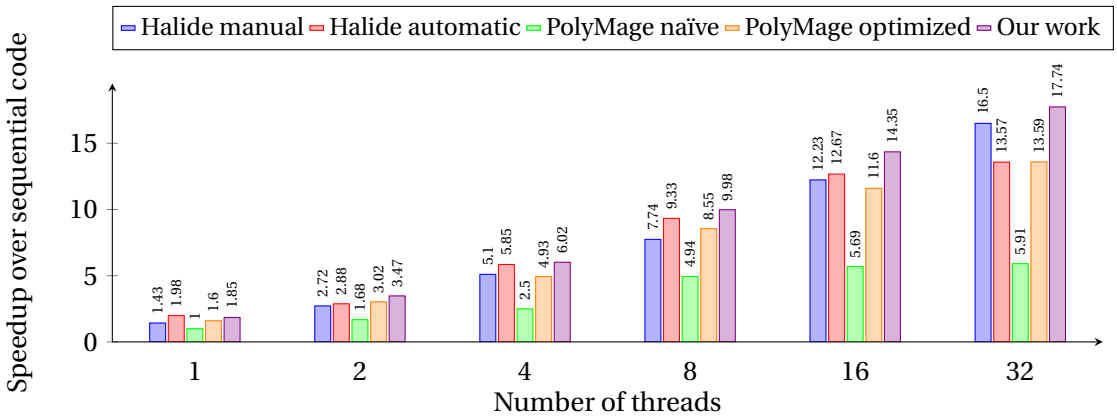


Figure 9.4 – Performance of Local Laplacian Filter on CPU

9.2.5 Multiscale Interpolation

Multiscale Interpolation is designed to preserve local shapes of an image at multiple scales by interpolating pixel values. The benchmark includes 49 stages consisting of sampling and stencil operations.

Like the case of Local Laplacian Filter, we perform the same loop fusion on these stages like PolyMage but minimize the redundant computation in shaded regions of overlapped tiling, obtaining a better performance than the optimized OpenMP code generated by PolyMage. Halide manual schedule specifies the similar schedule of PolyMage. Unfortunately, we fail to get the version of automatic scheduling algorithm of this benchmark from Halide repository, missing a comparison with this version in our experiment. We list the result of Multiscal Interpolation in Figure 9.5.

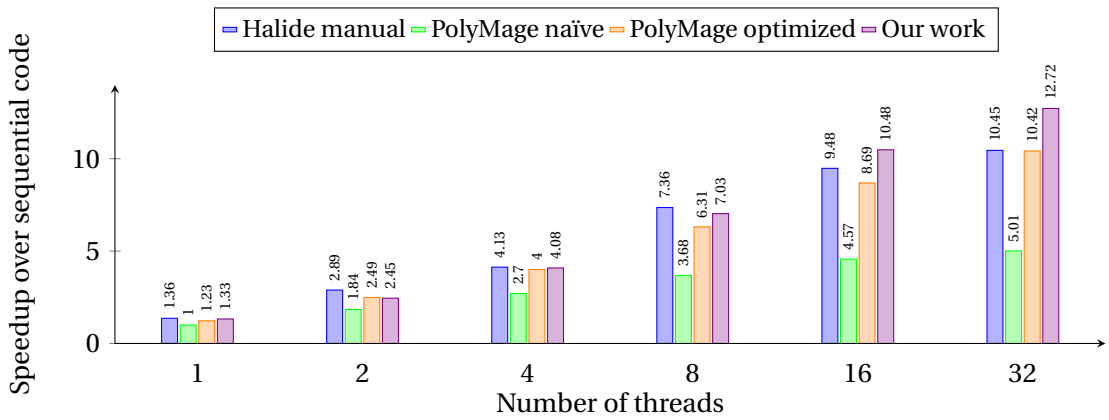


Figure 9.5 – Performance of Multiscale Interpolation on CPU

9.2.6 Pyramid Blending

Pyramid Blending [BA83] combines two images into a larger mosaic by constructing a Laplacian pyramid and performing filtering and splining operations. One may generate a complex grouping result by performing the fusion heuristic of the PolyMage framework, as we do for our technique.

The pipeline comprises 44 stages, constituting several stage groups after fusion and producing an overlapped tile after tiling. Our technique is designed to tighten such overlapped tile shapes for pipelines with a large number of stages, improving the performance of the optimized code of PolyMage, as shown in Figure 9.6. We do not find the Pyramid Blending benchmark in the Halide repository.

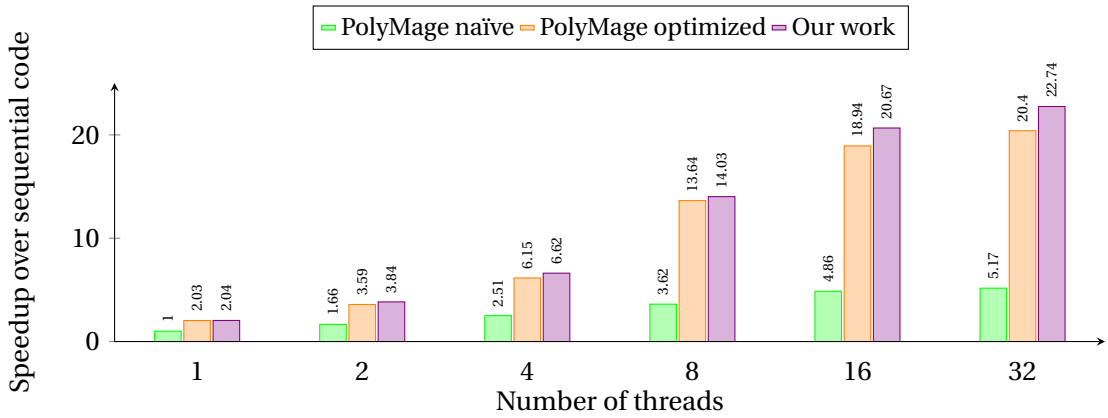


Figure 9.6 – Performance of Pyramid Blending on CPU

9.2.7 Unsharp Mask

Unsharp Mask is an image sharpening technique using an unsharp negative image to create a mask of the original image. This mask is then combined with the original image, constructing a less burry image.

The benchmark is a pipeline comprising a set of stencil operations, making it amenable for exploiting overlapped tiling using our technique. The stages of the pipeline can be fused into one group, benefiting from the tighter overlapped tile shape generated by our technique. The manually written schedule of Halide performs similar to the optimized version of PolyMage, outperforming the code generated from its automatic scheduling algorithm. We outline the performance comparison in Figure 9.7.

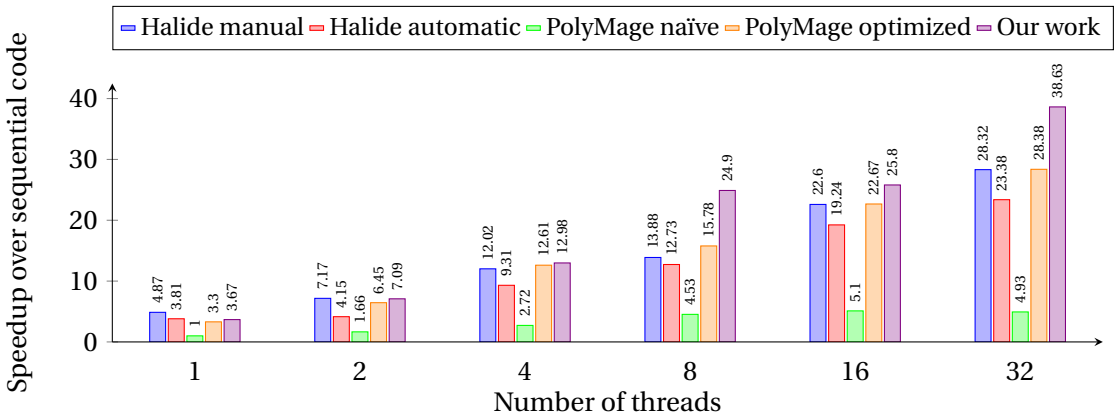


Figure 9.7 – Performance of Unsharp Mask on CPU

9.3 Iterated Stencils

To validate the general applicability of our technique, we also conduct experiments on three representative iterated stencils. The detailed information about the examples and tiling sizes we use in this subsection is list in Table 9.2, with the execution times of each sequential code shown in the last column. One can obtain the execution time of each CPU/GPU case by combining with Figure 9.8, 9.9, 9.10 and 9.12. We use right trapezoid tiling for experimenting because it has a better locality for iterated stencils than acute trapezoid tiling.

Table 9.2 – Problem sizes and tile sizes of the iterated stencils

	Problem sizes	Tile sizes			Baselinetime (s)
		standard	diamond	overlapped	
heat-1d	1000×160000	128×1024	2048^2	32×8192	6.99
heat-2d	1000×4000^2	16×64^2	64^3	4×64^2	7.17
heat-3d	100×150^3	$16^3\times256$	$16^3\times256$	$4\times16^2\times256$	1.21

We first run the sequential code of the stencils and record the execution time as a baseline reference. We compare the performance with state-of-the-art diamond tiling enabled by the Pluto compiler and parallelogram tiling enabled default by PPCG, showing the speedups of different techniques.

The *1/2/3d-heat* benchmarks we use in this subsection are iterated stencils solving the heat equations, iteratively updating data element using three-point and five-point stencils respectively. When selecting tile sizes, we follow the sizes chosen by diamond tiling [BBP17]. However, one may have to choose the sizes of parallelogram tiling carefully. The diamond tiling paper selects 1024×1024 for parallelogram tiling when comparing the performance, preventing the tiles from executing in wavefront parallelization because the tile size along time dimension is greater than the iteration time. We therefore choose to sacrifice locality to enable wavefront parallelization and ensure enough tiles on the wavefront.

One flaw of overlapped tiling is the redundant computation caused by shaded regions between neighboring tiles. One may thus have to select tile sizes for constructing a sharp overlapped tile, minimizing redundant computations as much as possible. We find 32×8192, $4\times64\times64$ and $4\times16^2\times256$ in practice for these stencils as the best tile sizes. The selection of diamond tile sizes follows its publication. We show the performance comparisons of these stencils in Figure Figure 9.8, 9.9 and 9.10, demonstrating overlapped tiling may achieve similar performance to diamond tiling by enabling inter-tile parallelization but without introducing complex rescheduling step and compilation overhead penalty.

Note that the slight performance gap between overlapped tiling and diamond tiling is due to the recomputation nature of overlapped tiling although an implementation of the latter can save compilation time. The purpose of our experiment by comparing with diamond tiling

is not to prove overlapped tiling can lead to a better performance, but we expect to instead validate the general applicability of our technique on iterated stencils. In addition, a fair comparison between such tiling techniques was also missing.

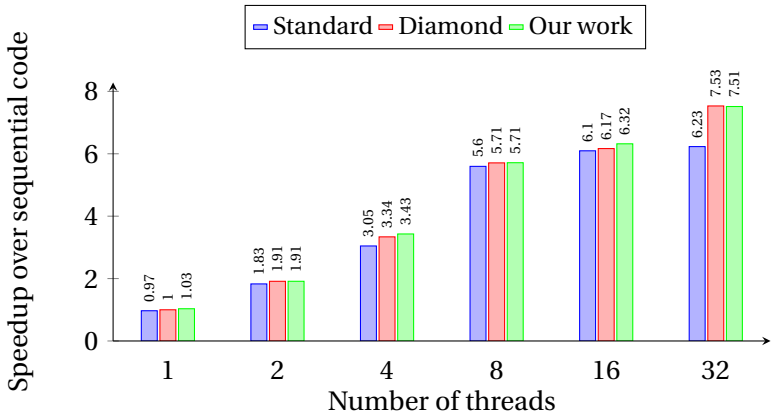


Figure 9.8 – Performance of heat-1d stencil on CPU

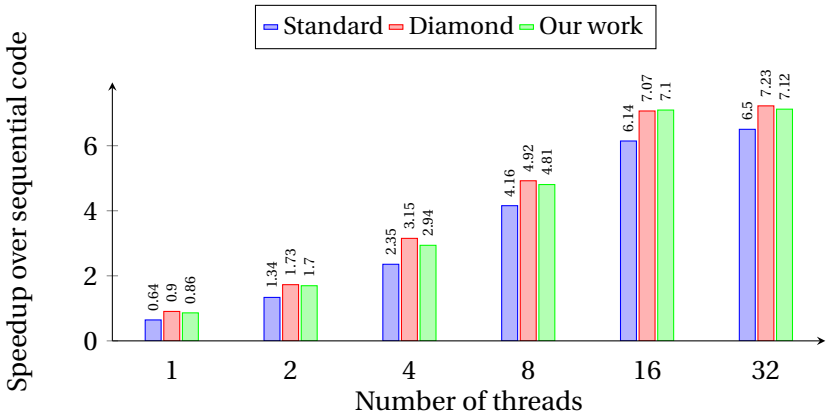


Figure 9.9 – Performance of heat-2d stencil on CPU

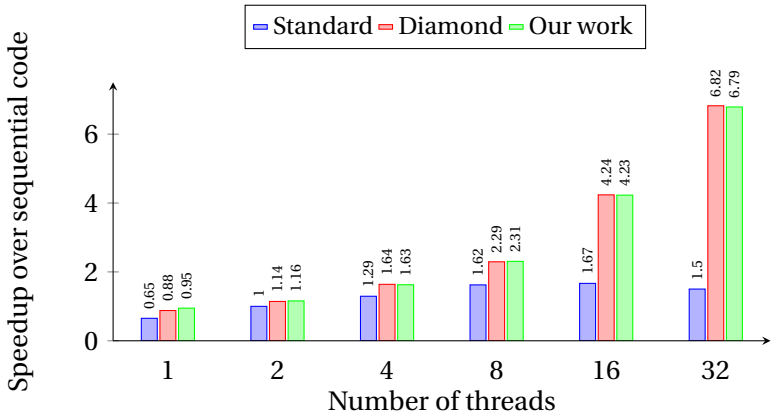


Figure 9.10 – Performance of heat-3d stencil on CPU

9.4 Performance on GPU Architectures

We also evaluate our technique on GPU architectures. The missing of the portability to GPU architectures is another flaw of the PolyMage framework, constraining its applicability to heterogeneous accelerators. We implement our technique in PPCG, allowing the generation of code for GPU architectures and thus extending the applicability of overlapped tiling on different architectures.

We use the generated CUDA code of PPCG by performing a simple fusion heuristic and rectangular tiling as the baseline for image processing pipelines. The Halide schedules may also be targeted to GPU devices, generating CUDA code by selecting approximate parameters. We therefore compare the performance with the manual schedule and the automatic scheduling algorithm of Halide. Figure 9.11 shows the performance comparison on GPU architectures.

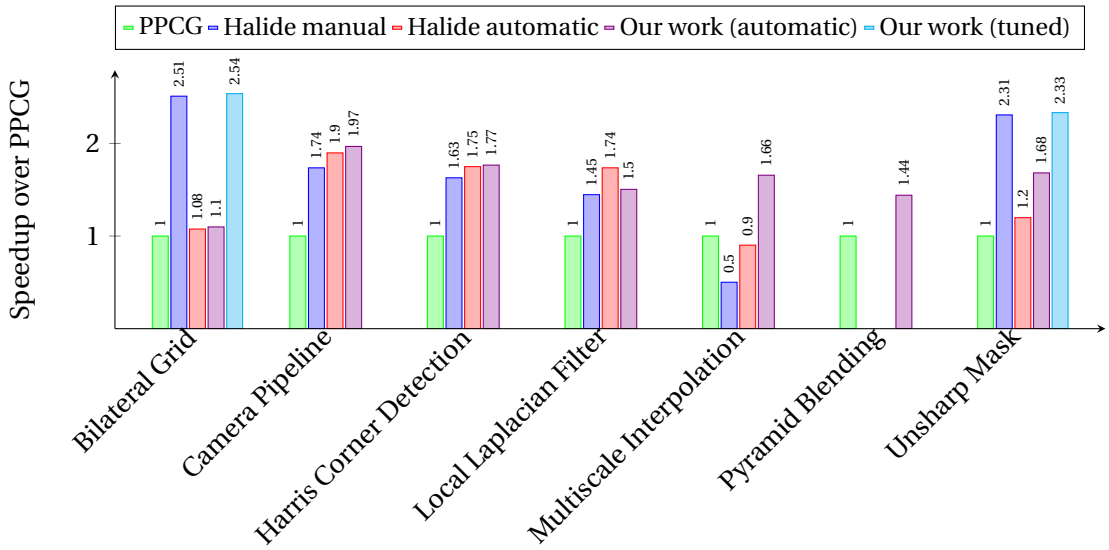


Figure 9.11 – Performance of the PolyMage benchmarks on GPU

Our technique obtains a steady performance improvement over the default setting of PPCG by enabling overlapped tiling on the pipelines. Our technique falls behind the manually written schedule of Halide for Bilateral Grid. The manually written schedule of Halide fuses the pipeline into two groups, one combining the histograms with one stencil and the other grouping all the remaining.

Our fusion heuristic is the same as the CPU case. The manual schedule of Unsharp Mask reduces load operations by unrolling inner loops where the same value is reloaded multiple times, leading to a better performance than automatic techniques. The Halide performance on Pyramid Blending is missing because the benchmark cannot be found from the repository.

Following the CPU case, we are allowed to fine-tune the generated code of our framework to eliminate the impact of manual fusion and unrolling optimization. The performance

such "tuned" version is also shown in Figure 9.11. Our technique can obtain a competitive performance with the manually scheduling method of Halide with such contemporary fusion and unrolling optimizations.

We also apply our technique on the iterated stencils when targeting on GPU architectures. We still use the sequential code of these stencils as a reference, and report the speedups of different tiling techniques in Figure 9.12. We first let PPCG perform a standard tiling on the stencils and compute the speedup over sequential code. We also compare the performance with the state-of-the-art hybrid hexagonal/classical tiling [GCH+14] that could be turned on by setting *--hybrid* option to PPCG, combining a hexagonal shape with classical tiling on time-space dimensions of stencils. We still use the same tiling size as the CPU case for our technique, and follows the tile sizes of diamond tiling for hexagonal/classical tiling.

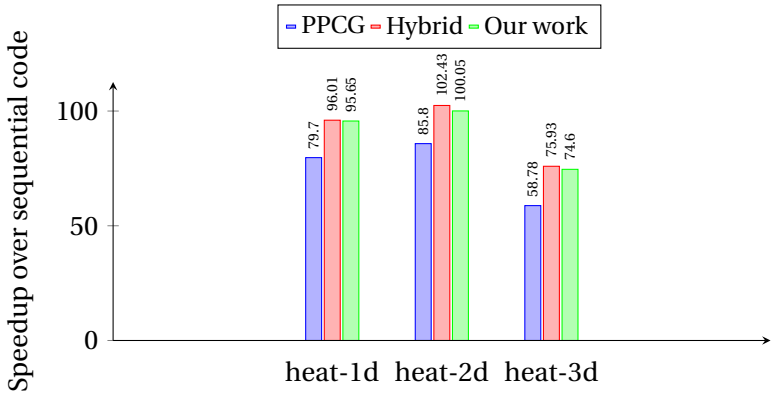


Figure 9.12 – Performance of the iterated stencils on GPU

Part IV

Conclusions

10 Conclusions and Perspectives

In this chapter, we conclude the thesis with an overview of our contributions and a prospect of future work.

10.1 Conclusions

The polyhedral model has become a powerful candidate for compilation tools achieving automatic parallelization and locality optimization. Its success is due to its impressive effectiveness on a variety of computational problems, and to its encouraging integration efforts into both research and industry compilers. Its applicability has also witnessed several extensions, resolving various challenges in domain specific areas and at different scales. However, polyhedral compilation was long criticized for its restrictions to affine applications and transformations. In this dissertation, we overcome some static control limitations of the polyhedral model from the following angles.

Handling Non-affine Applications. We first studied the parallelizing compilation and optimization of an important class of loop nests where counted loops have a dynamically computed, data-dependent upper bound. Such loops are amenable to a wider set of transformations than general `while` loops. To achieve this, we introduce a static upper bound and model control dependences on data-dependent predicates by revisiting a state-of-the-art framework to parallelize arbitrary `while` loops. We specialize this framework to facilitate its integration in schedule-tree-based affine scheduling and code generation algorithms, covering all scenarios from a single dynamic counted loop to nested parallelism across bands mapped to GPUs with fixed-size data-parallel grids.

Our method relies on systems of affine inequalities, as implemented in state-of-the-art polyhedral libraries. It takes a C program with PENCIL functions as input, covering a wide range of non-static control application encompassing the well studied class of sparse matrix computations. The experimental evaluation using the PPCG source-to-source compiler on representa-

tive irregular computations, from dynamic programming, computer vision and finite element methods to sparse matrix linear algebra, validated the general applicability of the method and its benefits over black-box approximations of the control flow.

Automating Non-affine Transformations. As the second contribution of the dissertation, we revisited overlapped tiling in a polyhedral compilation framework for optimizing image processing pipelines and iterated stencils. These classes of computations exhibit abundant data parallelism but require locality optimizations for improving performance. Our technique allows for tighter overlapped tile shapes than the state of the art, further improving the performance of such pipelines on both general-purpose multicores and heterogeneous accelerators by integrating with transformations including alignment and scaling of stages in the pipeline, loop fusion, scratchpad allocation, hybrid tiling, etc. Our technique can generate both acute and right trapezoid tile shapes, and has been implemented in the PPCG source-to-source compiler running on a general-purpose C input. We validated the general applicability of the approach and its benefits over a state-of-the-art framework.

10.2 Future Work

The thesis extends the application domain of polyhedral compilation to non-affine cases by combining the polyhedral model with a well-defined intermediate language, allowing for aggressive program transformations and automatic code generation strategies without resorting to speculative optimizations nor custom time-consuming rescheduling algorithms. There still exist a large number of remaining challenges along the non-affine extensions.

Dynamic Control Extensions to Deep Learning. In this dissertation, we studied the dynamic control issue in polyhedral compilation. The polyhedral model nowadays has been integrated with deep learning by automatically converting a high-level description of convolutional/recurrent networks into high-performance implementations amenable to the target processors and accelerators [VZT+18, BRR+18]. Such polyhedral-model-based software stacks are widely welcomed as a promising solution to optimizing custom operators that do not fit existing library calls. What is missing is the support of dynamic control extensions in such compiler stacks, restricting the applicable cases to dense tensor operations only.

The `taco` library [KKC+17] models sparse tensor algebra and allows for the automatic code generation of such operations. However, the library still remains being a prototype implementation for sparse tensors, leaving a variety of performance-crucial optimizations including tiling, autotuning, etc. that have already been implemented in or integrated with the polyhedral-model-based tools outside their frameworks. More importantly, the missing of code generation for heterogeneous platforms is still an opening issue. The parallelization and optimization of dynamic counted loops may inspire existing polyhedral tensor compiler stacks for supporting dynamic control cases, achieving an end-to-end compilation flow of

sparse tensor operations for both homogeneous and heterogeneous architectures.

Trade-off between Memory Locality and Redundant Computation. Tiling and fusion are two effective transformations for exploiting locality optimization. While a much larger space of different compositions of loop transformations could be exploited by the polyhedral model, the difficulty to reason about trade-offs between different criteria is also heavily exacerbated. In spite of the redundant computations, overlapped tiling is chosen to benefit from inter-tile parallelism while preserving locality in image processing pipelines due to its ability to allow for aggressive storage optimization. However, such redundant computations would be heavily exacerbated when fusing excessive stages in an image pipeline.

By constructing tighter trapezoid shapes, we successfully minimized the introduced recomputations in overlapped tiling. A fusion heuristic based on dynamic programming [JB18] was also proposed for optimizing the fusion strategy. Overlapped tiling would not be effective without storage reduction, while the latter in turn could be exploited by fusing as many stages as possible but result in more recomputations between tiles. However, we may still expect for a well-defined cost model to maximize locality and parallelism while minimizing redundant computations by integrating overlapped tiling and fusion heuristic.

Support to Dynamic High Performance Languages. In a similar way, the evolution of programming languages in the field of scientific computing also has to make a compromise with regard to ease of optimization, generality of coverage, difficulty in development, support of dynamism, etc. The scope of polyhedral compilation has been extensively widened by increasing the kinds of supported languages, from general-purpose languages to domain-specific languages with regard to the generality of coverage or from high-level descriptions to intermediate representations in respect of abstraction level of programming philosophy.

An interesting possible direction to further broaden the scope of polyhedral compilation is to model dynamic features of languages like C++ or the recently released Julia [BEKS17]. Particularly, the invention Julia integrates the merits of numerous programming languages, raising a variety of new challenges to optimizing system software. The dynamic control extensions made in this dissertation open the door to addressing new research problems in such dynamic languages by combining an intermediate language with the polyhedral model, providing a flexible compilation flow for lowering high-level, dynamic features into static controls amenable to polyhedral compilation and expanding the support to runtime dynamic checks.

Scalability of Polyhedral Compilation. The polyhedral compilation was long considered outside the domain of real-world applications due to its missing to handle dynamic control and non-affine programs. We provided a systematic way for handling dynamic, non-affine applications and generalizing non-affine transformations, evaluating the scalability of the idea

behind our techniques to a great number of real-world applications. Still, the performance of our generated code falls behind manually written libraries in some cases, leaving further room to optimize existing techniques. For example, we may further extend the framework to leverage the dynamic parallelism APIs provided by CUDA programming; we may also consider the compositions of overlapped tiling with other transformations like fusion and unrolling as mentioned before.

Finally, compilation time complexity also limits the scalability due to the underlying principles of polyhedral methods. Recently, some efforts attempted to relax the complexity of the underlying integer linear programming problem [ABC18] or leveraging statement clustering methods [MY15]. Nonetheless, such approaches rely heavily on exact dependence analyses which is still a very time-consuming step. A well-defined frontend capable of relaxing the time complexity of both dependence analysis and schedule transformation might be a promising solution, possibly in conjunction with a helper intermediate language for optimization purposes.

Bibliography

[ABC18] Aravind Acharya, Uday Bondhugula, and Albert Cohen. Polyhedral auto-transformation with no integer linear programming. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 529–542, New York, NY, USA, 2018. ACM.

[AI91] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '91, pages 39–50, New York, NY, USA, 1991. ACM.

[AKV+14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM.

[APH+14] Mathieu Aubry, Sylvain Paris, Samuel W. Hasinoff, Jan Kautz, and Frédo Durand. Fast local laplacian filters: Theory and applications. *ACM Trans. Graph.*, 33(5):167:1–167:14, September 2014.

[BA83] Peter J. Burt and Edward H. Adelson. A multiresolution spline with application to image mosaics. *ACM Trans. Graph.*, 2(4):217–236, October 1983.

[BAA+14] S Balay, S Abhyankar, M Adams, J Brown, P Brune, K Buschelman, V Eijkhout, W Gropp, D Kaushik, M Knepley, et al. Petsc users manual revision 3.5. *Argonne National Laboratory*, 2014.

[BAC16] Uday Bondhugula, Aravind Acharya, and Albert Cohen. The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests. *ACM Trans. Program. Lang. Syst.*, 38(3):12:1–12:32, April 2016.

[Ban93] Utpal Banerjee. *Unimodular Matrices*, pages 21–48. Springer US, Boston, MA, 1993.

[Bas04] Cedric Bastoul. Code generation in the polyhedral model is easier than you think. In *Proceedings of the 13th International Conference on Parallel Architectures and*

Compilation Techniques, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[BB13] Somashekaracharya G. Bhaskaracharya and Uday Bondhugula. Polyglot: A polyhedral loop transformation framework for a graphical dataflow language. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 123–143, Berlin, Heidelberg, 2013. Springer-Verlag.

[BBC+14] Uday Bondhugula, Vinayaka Bandishti, Albert Cohen, Guillain Potron, and Nicolas Vasilache. Tiling and optimizing time-iterated computations on periodic domains. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 39–50, New York, NY, USA, 2014. ACM.

[BBC+15] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema, Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, PACT '15, pages 138–149, Washington, DC, USA, 2015. IEEE Computer Society.

[BBC16] Somashekaracharya G. Bhaskaracharya, Uday Bondhugula, and Albert Cohen. Smo: An integrated approach to intra-array and inter-array storage optimization. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 526–538, New York, NY, USA, 2016. ACM.

[BBP17] Uday Bondhugula, Vinayaka Bandishti, and Irshad Pananilath. Diamond tiling: Tiling techniques to maximize parallelism for stencil computations. *IEEE Transactions on Parallel and Distributed Systems*, 28(5):1285–1298, October 2017.

[BCVT13] Riyadh Baghdadi, Albert Cohen, Sven Verdoolaege, and Konrad Trifunović. Improved loop tiling based on the removal of spurious false dependences. *ACM Trans. Archit. Code Optim.*, 9(4):52:1–52:26, January 2013.

[BEKS17] J. Bezanson, A. Edelman, S. Karpinski, and V. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017.

[BF03] Cédric Bastoul and Paul Feautrier. Improving data locality by chunking. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 320–334, 2003.

[BG09] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[BG11] Aydin Buluc and John R Gilbert. The combinatorial blas: Design, implementation, and applications. *Int. J. High Perform. Comput. Appl.*, 25(4):496–509, November 2011.

[BGDR10] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM.

[BGS94] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[BHZ08] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, June 2008.

[BKP+16] Wenlei Bao, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, and P. Sadayappan. Polycheck: Dynamic verification of iteration space transformations on affine programs. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 539–554, New York, NY, USA, 2016. ACM.

[Bon08] Uday Kumar Reddy Bondhugula. *Effective Automatic Parallelization and Locality Optimization Using the Polyhedral Model*. PhD thesis, Columbus, OH, USA, 2008. AAI3325799.

[Bon13] Uday Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 33:1–33:12, New York, NY, USA, 2013. ACM.

[BPB12] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[BPCB10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think.

In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 283–303, Berlin, Heidelberg, 2010. Springer-Verlag.

[BRR+18] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. Tiramisu: A code optimization framework for high performance systems. *CoRR*, abs/1804.10694, 2018.

[BRS07] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to fpgas. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 101–111, New York, NY, USA, 2007. ACM.

[CBF95] Jean-François Collard, Denis Barthou, and Paul Feautrier. Fuzzy array dataflow analysis. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 92–101, New York, NY, USA, 1995. ACM.

[CCH08] Chun Chen, Jacqueline Chame, and Mary Hall. Chill: A framework for composing high-level loop transformations. Technical report, 2008.

[Che] Chun Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12.

[cla] Clan: A polyhedral representation extraction tool for c-based high level languages. http://icps.u-strasbg.fr/~bastoul/development/clan/. Accessed: 2018-08-01.

[CMJ+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: end-to-end optimization stack for deep learning. *CoRR*, abs/1802.04799, 2018.

[Col94] J. . Collard. Space-time transformation of while-loops using speculative execution. In *Proceedings of IEEE Scalable High Performance Computing Conference*, pages 429–436, May 1994.

[Col95] Jean-François Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. Parallel Program.*, 23(2):191–219, April 1995.

[CPD07] Jiawen Chen, Sylvain Paris, and Frédo Durand. Real-time edge-aware image processing with the bilateral grid. In *ACM SIGGRAPH 2007 Papers*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.

[CSG+05] Albert Cohen, Marc Sigler, Sylvain Girbal, Olivier Temam, David Parello, and Nicolas Vasilache. Facilitating the search for compositions of program transformations. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 151–160, New York, NY, USA, 2005. ACM.

[DH11] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[DSO18] Eddie C. Davis, Michelle Mills Strout, and Catherine Olschanowsky. Transforming loop chains via macro dataflow graphs. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 265–277, New York, NY, USA, 2018. ACM.

[DSV05] A. Darte, R. Schreiber, and G. Villard. Lattice-based memory allocation. *IEEE Transactions on Computers*, 54(10):1242–1257, Oct 2005.

[EM90] H Eissfeller and S. M Muller. The triangle method for saving startup time in parallel computers. In *Distributed Memory Computing Conference, 1990., Proceedings of the Fifth*, pages 568–572, 1990.

[Fea88] Paul Feautrier. Parametric integer programming. *RAIRO-Operations Research*, 22(3):243–268, 1988.

[Fea91] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb 1991.

[Fea92a] Paul Feautrier. Some efficient solutions to the affine scheduling problem. i. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–347, Oct 1992.

[Fea92b] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, Dec 1992.

[FL11] Paul Feautrier and Christian Lengauer. *Polyhedron Model*, pages 1581–1592. Springer US, Boston, MA, 2011.

[GC95] Martin Griebl and Jean-François Collard. Generation of synchronous code for automatic parallelization of while loops. In Seif Haridi, Khayri Ali, and Peter Magnusson, editors, *EURO-PAR '95 Parallel Processing*, pages 313–326, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[GCH+14] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[GCK+13] 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 *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, GPGPU-6, pages 24–31, New York, NY, USA, 2013. ACM.

[GFL00] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index set splitting. *Int. J. Parallel Program.*, 28(6):607–631, December 2000.

[GGL98] Max Geigl, Martin Griebl, and Christian Lengauer. A scheme for detecting the termination of a parallel loop nest. *Proc. GI/ITG FG PARS*, 98, 1998.

[GGL99] Max Geigl, Martin Griebl, and Christian Lengauer. Termination detection in parallel loop nests with while loops. *Parallel Computing*, 25(12):1489 – 1510, 1999.

[GGL12] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[GL94] Martin Griebl and Christian Lengauer. On scanning space-time mapped while loops. In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI*, pages 677–688, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.

[Gro14] Tobias Grosser. *A decoupled approach to high-level loop optimization: tile shapes, polyhedral building blocks and low-level compilers*. PhD thesis, Université Pierre et Marie Curie-Paris VI, 2014.

[GV15] Pieter Ghysels and Wim Vanroose. Modeling the performance of geometric multigrid stencils on multicore computer architectures. *SIAM Journal on Scientific Computing*, 37(2):C194–C216, 2015.

[GVB+06] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, June 2006.

[GVC15] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12:1–12:50, July 2015.

[GVCS14] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P Sadayappan. The relation between diamond tiling and hexagonal tiling. *Parallel Processing Letters*, 24(03):1441002, 2014.

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

[HS88] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[HVF+13] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 13–24, New York, NY, USA, 2013. ACM.

[ISO99] ISO. The ansi c standard (c99). *Technical Report WG14 N1124, ISO/IEC*, 1999.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 319–329, New York, NY, USA, 1988. ACM.

[JB18] Abhinav Jangda and Uday Bondhugula. An effective fusion and tile size model for optimizing image processing pipelines. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 261–275, New York, NY, USA, 2018. ACM.

[JCD+14] Alexandra Jimborean, Philippe Clauss, Jean-François Dollinger, Vincent Loechner, and Juan Manuel Martinez Caamaño. Dynamic and speculative polyhedral parallelization using compiler-generated skeletons. *Int. J. Parallel Program.*, 42(4):529–545, August 2014.

[JGTC14] J. C. Juega, J. I. Gomez, C. Tenllado, and F. Catthoor. Adaptive mapping and parameter selection scheme to improve automatic code generation for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 251:251–251:261, New York, NY, USA, 2014. ACM.

[KA02] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[KBB+07] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective automatic parallelization of stencil computations. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 235–244, New York, NY, USA, 2007. ACM.

[KKC+17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[KMP+96] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The omega calculator and library, version 1.1. 0. *College Park, MD*, 20742:18, 1996.

[KMW67] Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *J. ACM*, 14(3):563–590, July 1967.

[KP95] W. Kelly and W. Pugh. A unifying framework for iteration reordering transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 1, pages 153–162 vol.1, April 1995.

[KPP+15] Martin Kong, Antoniu Pop, Louis-Noël Pouchet, R. Govindarajan, Albert Cohen, and P. Sadayappan. Compiler/runtime framework for dynamic dataflow parallelization of tiled programs. *ACM Trans. Archit. Code Optim.*, 11(4):61:1–61:30, January 2015.

[KPR95] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. In *Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, Feb 1995.

[KRR+07] DaeGon Kim, Lakshminarayanan Renganarayanan, Dave Rostron, Sanjay Rajopadhye, and Michelle Mills Strout. Multi-level tiling: M for the price of one. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 51:1–51:12, New York, NY, USA, 2007. ACM.

[KVS+13] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan. When polyhedral transformations meet simd code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 127–138, New York, NY, USA, 2013. ACM.

[LBG+12] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[LL97] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[LLS06] B Meister R Lethin, Allen Leung, and E Schweitz. R-stream: A parametric high level compiler. In *High Performance Embedded Computing Workshop*, 2006.

[Loe99] Vincent Loechner. Polylib: A library for manipulating parameterized polyhedra, 1999.

[LW97] Vincent Loechner and Doran K. Wilde. Parameterized polyhedra and their vertices. *Int. J. Parallel Program.*, 25(6):525–549, December 1997.

[MAL93] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 2–15, New York, NY, USA, 1993. ACM.

[Mas94] Vadim Maslov. Lazy array data-flow dependence analysis. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 311–325, New York, NY, USA, 1994. ACM.

[MAS+16] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016.

[MCG04] John Mellor-Crummey and John Garvin. Optimizing sparse matrix-vector product computations using unroll and jam. *Int. J. High Perform. Comput. Appl.*, 18(2):225–236, May 2004.

[MHLK17] Tareq M. Malas, Georg Hager, Hatem Ltaief, and David E. Keyes. Multidimensional intratile parallelization for memory-starved stencil computations. *ACM Trans. Parallel Comput.*, 4(3):12:1–12:32, December 2017.

[MLY14] Sanyam Mehta, Pei-Hung Lin, and Pen-Chung Yew. Revisiting loop fusion in the polyhedral framework. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 233–246, New York, NY, USA, 2014. ACM.

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM.

[MY15] Sanyam Mehta and Pen-Chung Yew. Improving compiler scalability: Optimizing large programs at small price. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 143–152, New York, NY, USA, 2015. ACM.

[OG09] Daniel A. Orozco and Guang R. Gao. Mapping the fdtd application to many-core chip architectures. In *International Conference on Parallel Processing*, pages 309–316, 2009.

[PAVB15] Irshad Pananilath, Aravind Acharya, Vinay Vasista, and Uday Bondhugula. An optimizing code generator for a class of lattice-boltzmann computations. *ACM Trans. Archit. Code Optim.*, 12(2):14:1–14:23, May 2015.

[PHK15] Sylvain Paris, Samuel W. Hasinoff, and Jan Kautz. Local laplacian filters: Edge-aware image processing with a laplacian pyramid. *Commun. ACM*, 58(3):81–91, February 2015.

[PKT+09] Sylvain Paris, Pierre Kornprobst, Jack Tumblin, Frédo Durand, et al. Bilateral filtering: Theory and applications. *Foundations and Trends® in Computer Graphics and Vision*, 4(1):1–73, 2009.

[Pug91] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[PW92] William Pugh and David Wonnacott. Eliminating false data dependences using the omega test. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 140–151, New York, NY, USA, 1992. ACM.

[PW94a] William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 546–566, London, UK, UK, 1994. Springer-Verlag.

[PW94b] William Pugh and David Wonnacott. Nonlinear array dependence analysis. Technical report, College Park, MD, USA, 1994.

[PW94c] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278, July 1994.

[PZSC13] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '13, pages 29–38, New York, NY, USA, 2013. ACM.

[QRW00] Fabien Quilleré, Sanjay Rajopadhye, and Doran Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, Oct 2000.

[RB14] Chandan Reddy and Uday Bondhugula. Effective automatic computation placement and data allocation for parallelization of regular programs. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[RDE+15] Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Distributed memory code generation for mixed irregular/regular computations. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, pages 65–75, New York, NY, USA, 2015. ACM.

[RKAP+12] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[RKC16] Chandan Reddy, Michael Kruse, and Albert Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on gpu. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 87–97, New York, NY, USA, 2016. ACM.

[RP95] L. Rauchwerger and D. Padua. Parallelizing while loops for multiprocessor systems. In *Proceedings of 9th International Parallel Processing Symposium*, pages 347–356, April 1995.

[SCF03] Michelle Mills Strout, Larry Carter, and Jeanne Ferrante. Compile-time composition of run-time data and iteration reorderings. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 91–102, New York, NY, USA, 2003. ACM.

[SCFS98] Michelle Mills Strout, Larry Carter, Jeanne Ferrante, and Beth Simon. Schedule-independent storage mapping for loops. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 24–33, New York, NY, USA, 1998. ACM.

[SGM+15] Sunil Shrestha, Guang R. Gao, Joseph Manzano, Andres Marquez, and John Feo. Locality aware concurrent start for stencil applications. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 157–166, Washington, DC, USA, 2015. IEEE Computer Society.

[SGO13] Michelle Mills Strout, Geri Georg, and Catherine Olschanowsky. Set and relation manipulation for the sparse polyhedral framework. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, pages 61–75, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[SHS17] Jun Shirako, Akihiro Hayashi, and Vivek Sarkar. Optimized two-level parallelization for gpu accelerators using the polyhedral model. In *Proceedings of the 26th International Conference on Compiler Construction*, CC 2017, pages 22–33, New York, NY, USA, 2017. ACM.

[SLC+16] Michelle Mills Strout, Alan LaMielle, Larry Carter, Jeanne Ferrante, Barbara Kreaseck, and Catherine Olschanowsky. An approach for code generation in the sparse polyhedral framework. *Parallel Comput.*, 53(C):32–57, April 2016.

[SPR17] Diogo N. Sampaio, Louis-Noël Pouchet, and Fabrice Rastello. Simplification and runtime resolution of data dependence constraints for loop transformations. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 10:1–10:11, New York, NY, USA, 2017. ACM.

[SRC15] Aravind Sukumaran-Rajam and Philippe Clauss. The polyhedral model of nonlinear loops. *ACM Trans. Archit. Code Optim.*, 12(4):48:1–48:27, December 2015.

[SSPS11] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache accurate time skewing in iterative stencil computations. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 571–581, Washington, DC, USA, 2011. IEEE Computer Society.

[SSPS16] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Polyhedral optimizations for a data-flow graph language. In *Revised Selected Papers of the 28th International Workshop on Languages and Compilers for Parallel Computing - Volume 9519*, LCPC 2015, pages 57–72, Berlin, Heidelberg, 2016. Springer-Verlag.

[TCE+10] Konrad Trifunovic, Albert Cohen, David Edelsohn, Feng Li, Tobias Grosser, Harsha Jagasia, Razya Ladelsky, Sebastian Pop, Jan Sjödin, and Ramakrishna Upadrasta. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, January 2010.

[TNC+09] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. Polyhedral-model guided loop-nest auto-vectorization. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 327–337, Washington, DC, USA, 2009. IEEE Computer Society.

[UC13] Ramakrishna Upadrasta and Albert Cohen. Sub-polyhedral scheduling using (unit-)two-variable-per-inequality polyhedra. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 483–496, New York, NY, USA, 2013. ACM.

[VBC06] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In Alan Mycroft and Andreas Zeller, editors, *Compiler Construction*, pages 185–201, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[VBCG06] Nicolas Vasilache, Cedric Bastoul, Albert Cohen, and Sylvain Girbal. Violated dependence analysis. In *Proceedings of the 20th Annual International Conference on Supercomputing*, ICS '06, pages 335–344, New York, NY, USA, 2006. ACM.

[VCJC+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013.

[VDY05] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[Ver10] Sven Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer-Verlag.

[VG12] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, January 2012.

[VHS15] Anand Venkat, Mary Hall, and Michelle Strout. Loop and data transformations for sparse matrix code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 521–532, New York, NY, USA, 2015. ACM.

[VMP+16] Anand Venkat, Mahdi Soltan Mohammadi, Jongsoo Park, Hongbo Rong, Rajkishore Barik, Michelle Mills Strout, and Mary Hall. Automating wavefront parallelization for sparse matrix computations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pages 41:1–41:12, Piscataway, NJ, USA, 2016. IEEE Press.

[VSHS14] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 185:185–185:194, New York, NY, USA, 2014. ACM.

[VZT+18] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[WL91] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2(4):452–471, October 1991.

[WLC14] Yuxin Wang, Peng Li, and Jason Cong. Theory and algorithm for generalized memory partitioning in high-level synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 199–208, New York, NY, USA, 2014. ACM.

[YGK+13] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In Hironori Kasahara and Keiji Kimura, editors, *Languages and Compilers for Parallel Computing*, pages 17–31, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[ZGG+12] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM.

[ZHC17] Tian Zhao, Xiaobing Huang, and Yu Cao. Deepdsl: A compilation-based domain-specific language for deep learning. *CoRR*, abs/1701.02284, 2017.

[ZLC+13] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. Improving polyhedral code generation for high-level synthesis. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '13, pages 15:1–15:10, Piscataway, NJ, USA, 2013. IEEE Press.

Résumé

De nos jours, l'optimisation des compilateurs est de plus en plus mise à l'épreuve par la diversité des langages de programmation et l'hétérogénéité des architectures. Le modèle polyédrique est un puissant cadre mathématique permettant aux programmes d'exploiter la parallélisation automatique et l'optimisation de la localité, jouant un rôle important dans le domaine de l'optimisation des compilateurs. Une limite de longue date du modèle réside dans sa restriction aux programmes affines à contrôle statique, ce qui a entraîné une demande émergente de prise en charge d'extensions non affines. Cela est particulièrement aigu dans le contexte d'architectures hétérogènes où une variété de noyaux de calcul doivent être analysés et transformés pour répondre aux contraintes des accélérateurs matériels et pour gérer les transferts de données à travers des espaces mémoire. Nous explorons plusieurs extensions non affines du modèle polyhédral, dans le contexte d'un langage intermédiaire bien défini combinant des éléments affines et syntaxiques. D'un côté, nous expliquons comment les transformations et la génération de code pour des boucles avec des limites de boucle dynamiques non dépendantes des données et dynamiques sont intégrées dans un cadre polyédrique, élargissant ainsi le domaine applicable de la compilation polyédrique dans le domaine des applications non affines. D'autre part, nous décrivons l'intégration du pavage en recouvrement pour les calculs de pochoir dans un cadre polyhédral général, en automatisant les transformations non affines dans la compilation polyhédrique. Nous évaluons nos techniques sur des architectures de CPU et de CPU, en validant l'efficacité des optimisations en effectuant une comparaison approfondie des performances avec des frameworks et des librairies écrites à la pointe de la technologie.

Mots Clés

Programmation parallèle, compilation polyédrique, parallélisation automatique

Abstract

Nowadays, optimizing compilers are increasingly challenged by the diversity of programming languages and heterogeneity of architectures. The polyhedral model is a powerful mathematical framework for programs to exploit automatic parallelization and locality optimization, playing an important role in the field of optimizing compilers. A long standing limitation of the model has been its restriction to static control affine programs, resulting in an emergent demand for the support of non-affine extensions. This is particularly acute in the context of heterogeneous architectures where a variety of computation kernels need to be analyzed and transformed to match the constraints of hardware accelerators and to manage data transfers across memory spaces. We explore multiple non-affine extensions of the polyhedral model, in the context of a well-defined intermediate language combining affine and syntactic elements. On the one hand, we explain how transformations and code generation for loops with non-affine, data-dependent and dynamic loop bounds are integrated into a polyhedral framework, extending the applicable domain of polyhedral compilation in the realm of non-affine applications. On the other hand, we describe the integration of overlapped tiling for stencil computations into a general polyhedral framework, automating non-affine transformations in polyhedral compilation. We evaluate our techniques on both CPU and CPU architectures, validating the effectiveness of the optimizations by conducting an in-depth performance comparison with state-of-the-art frameworks and manually-written libraries.

Keywords

Parallel programming, polyhedral compilation, automatic parallelization