Université Pierre et Marie Curie

École Doctorale Informatique, Télécommunications et Électronique

INRIA, École Normale Supérieure / Équipe PARKAS

A DECOUPLED APPROACH TO HIGH-LEVEL LOOP OPTIMIZATION

– TILE SHAPES, POLYHEDRAL BUILDING BLOCKS AND LOW-LEVEL COMPILERS –

Par Tobias Christian Grosser

Thése de doctorat en informatique

Codirigée par Albert Cohen et Sven Verdoolaege

Présentée et soutenue publiquement le 21/10/2014
Devant le jury composé de:

Rapporteurs:	Mary Hall	University of Utah
	Franz Franchetti	Carnegie Mellon University
Examinateurs:	Stef Graillat	Université Pierre et Marie Curie
	Jean-Luc Lamotte	Université Pierre et Marie Curie
	Franz Franchetti	Carnegie Mellon University
	Ponnuswamy Sadayappan	Ohio State University
	Albert Cohen	INRIA
	Sven Verdoolaege	École Normale Supérieure

Tobias Grosser: *A decoupled approach to high-level loop optimization,* Tile shapes, polyhedral building blocks and low-level compilers, November 19, 2014

RÉSUMÉ

Malgré des décennies de recherche sur l'optimisation de boucle aux haut niveau et leur intégration réussie dans les compilateurs C/C++ et FORTRAN, la plupart des systèmes de transformation de boucle ne traitent que partiellement les défis posé par la complexité croissante et la diversité du matériel d'aujourd'hui. L'exploitation de la connaissance dédiée a un domaine d'application pour obtenir le code optimal pour cibles complexes, tels que des accélérateurs ou des microprocessors multi-cœur, pose des problèmes pour les formalismes et outils d'optimisation de boucle existants. En conséquence, de nouveaux schémas d'optimisation qui exploitent la connaissance dédiée a un domaine sont développées indépendamment sans profiter de la technologie d'optimisation de boucle existante. Cela conduit à des possiblités d'optimisation raté et ainsi qu'à une faible portabilité de ces schémas d'optimisation entre des compilateurs différents. Un domaine pour lequel on voit la nécessité d'améliorer les optimisations est le calcul de pochoir itératifs, un probléme de calcul important qui est réguliérement optimisé par les compilateurs dédiées, mais pour lequel générer code efficace est difficile.

Dans ce travail, nous présentons des nouvelles stratégies pour l'optimisation dédiée qui permettent la génération de code GPU haute performance pour des calculs de pochoir. À la différence de la façon dont la plupart des compilateurs existants sont mis en œuvre, nous découplons la stratégie d'optimisation de haut niveau de l'optimisation de bas niveau et la spécialisation nécessaire pour obtenir la performance optimale. Comme schéma d'optimisation de haut niveau, nous présentons une nouvelle formulation de "split tiling", une technique qui permet la réutilisation de données dans la dimension du temps ainsi que le parallélisme équilibré à gros grain sans la nécessité de recourir à des calculs redondants. Avec le "split tiling", nous montrons comment intégrer une optimisation dédiée dans un traducteur générique source-à-source, C vers CUDA, une approche qui nous permet de réutiliser des optimisations existants non-dédiées. Nous présentons ensuite notre technique appelée "hybrid hexagonal / parallelogram tiling", un schéma qui nous permet de générer du code que cible directement les préoccupations spécifiques aux GPUs. Pour conclure notre travail sur le "loop tiling", nous étudions la rapport entre "diamond tiling" et "hexagonal tiling". À partir d'une analyse de "diamond tiling" détailée, qui comprend les exigences qu'elle pose sur la taille de tuile et les coefficients de front d'onde, nous fournissons une formulation unifiée de l'"hexagonal tiling" et du "diamond tiling" qui nous permet de réaliser un "hexagonal tiling" pour

des problèmes avec deux dimensions (un temps, un espace) dans le cadre d'un usage dans un optimiseur générique, comme "Pluto". Enfin, nous utilisons cette formulation pour évaluer l'"hexagonal tiling" et le "diamond tiling" en terme de rapport de calcul-à-communication et calcul-à-synchronisation.

Dans la deuxième partie de ce travail, nous discutons nos contributions aux composants de l'infrastructure les plus important, nos "building blocks", qui nous permettent de découpler notre optimisation de haut niveau tant des optimisations nécessaires dàns la génération de code que de l'infrastructure de compilation générique. Nous commençons par présenter le nouveau "polyhedral extractor" (`pet`), qui obtient une représentation polyédrique d'un morceau de code C. `pet` utilise l'arithmétique de Presburger en sa généralité pour élargir le fragment de code C supporté et porter une attention particulière à la modélisation de la sémantique des langages même en présence de dépassement de capacité des entiers. Dans une prochaine étape, nous présentons une nouvelle approche polyédrique pour la génération de code, qui étend cell-ci au-delà de la génération de flot de contrôle classique en permettant la génération de expressions fournies par l'utilisateur. Avec un mécanisme d'option détaillé, nous donnons à l'utilisateur un contrôle précis sur les décisions de notre générateur du code et nous ajoutons un support pour la spécialisation extensive par exemple, avec une nouvelle forme de déroulage de boucle polyédrique. Pour faciliter la mise en œuvre de transformations polyédriques, nous présentons une nouvelle représentation, les "schedule trees", qui représentent explicitement la structure d'arbre inhérent aux ordonnancements polyédriques.

La dernière partie de cet ouvrage présente nos contributions aux compilateurs du bas niveau. L'objectif principal de cette partie est notre travail sur la délinéarisation optimiste, une approche pour dériver une vue de tableau multi-dimensionnel pour des formes accès en polynômiales multivariées qui résultent souvent de code qui travaille avec des tableaux multi-dimensionnels avec taille paramétrique.

ABSTRACT

Despite decades of research on high-level loop optimizations and their successful integration in production C/C++/FORTRAN compilers, most compiler internal loop transformation systems only partially address the challenges posed by the increased complexity and diversity of today's hardware. Especially when exploiting domain specific knowledge to obtain optimal code for complex targets such as accelerators or many-cores processors, many existing loop optimization frameworks have difficulties exploiting this hardware. As a result, new domain specific optimization schemes are developed independently without taking advantage of existing loop optimization technology. This results both in missed optimization opportunities as well as low portability of these optimization schemes to different compilers. One area where we see the need for better optimizations are iterative stencil computations, an important computational problem that is regularly optimized by specialized, domain specific compilers, but where generating efficient code is difficult.

In this work we present new domain specific optimization strategies that enable the generation of high-performance GPU code for stencil computations. Different to how most existing domain specific compilers are implemented, we decouple the high-level optimization strategy from the low-level optimization and specialization necessary to yield optimal performance. As high-level optimization scheme we present a new formulation of split tiling, a tiling technique that ensures reuse along the time dimension as well as balanced coarse grained parallelism without the need for redundant computations. Using split tiling we show how to integrate a domain specific optimization into a general purpose C-to-CUDA translator, an approach that allows us to reuse existing non-domain specific optimizations. We then evolve split tiling into a hybrid hexagonal/parallelogram tiling scheme that allows us to generate code that even better addresses GPU specific concerns. To conclude our work on tiling schemes we investigate the relation between diamond and hexagonal tiling. Starting with a detailed analysis of diamond tiling including the requirements it poses on tile sizes and wavefront coefficients, we provide a unified formulation of hexagonal and diamond tiling which enables us to perform hexagonal tiling for two dimensional problems (one time, one space) in the context of a general purpose optimizer such as Pluto. Finally, we use this formulation to evaluate hexagonal and diamond tiling in terms of compute-to-communication and compute-to-synchronization ratios.

In the second part of this work, we discuss our contributions to important infrastructure components, our building blocks, that en-

able us to decouple our high-level optimizations from both the necessary code generation optimizations as well as the compiler infrastructure we apply the optimization to. We start with presenting a new polyhedral extractor that obtains a polyhedral representation from a piece of C code, widening the supported C code to exploit the full generality of Presburger arithmetic and taking special care of modeling language semantics even in the presence of defined integer wrapping. As a next step, we present a new polyhedral AST generation approach, which extends AST generation beyond classical control flow generation by allowing the generation of user provided mappings. Providing a fine-grained option mechanism, we give the user fine grained control about AST generator decisions and add extensive support for specialization e.g., with a new generalized form of polyhedral unrolling. To facilitate the implementation of polyhedral transformations, we present a new schedule representation, schedule trees, which proposes to make the inherent tree structure of schedules explicit to simplify the work with complex polyhedral schedules.

The last part of this work takes a look at our contributions to low-level compilers. The main focus in this part is our work on optimistic delinearization, an approach to derive a multi-dimensional array view for multi-variate polynomial expressions which commonly result from code that models data as multi-dimensional arrays of parametric size.

PUBLICATIONS

The following is a list of publications I wrote or contributed to while working towards my doctoral degree. Ideas, figures, listings and text of some of these works have been incorporated in this thesis.

The work published in [8, 3] was mostly performed within the context of my diploma degree and is listed due to the relevance for parts of this work. [4] has been submitted for review, but has not yet been published. A new technical report which we cite in the AST generation chapter still needs to be published.

[1] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. Hybrid Hexagonal/Classical Tiling for GPUs. In *International Symposium on Code Generation and Optimization (CGO)*, Orlando, FL, United States, 2014.

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

[3] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters (PPL)*, 22(04), 2012.

[4] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Transactions on Programming Languages and Systems (TOPLAS)*. (submitted for review).

[5] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P. Sadayappan. The relation between diamond tiling and hexagonal tiling. In *1st International Workshop on High-Performance Stencil Computations (HiStencils)*.

[6] Tobias Grosser, Sven Verdoolaege, Albert Cohen, and P. Sadayappan. The Promises of Hybrid Hexagonal/Classical Tiling for GPU. Rapport de recherche RR-8339, INRIA, July 2013.

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

[8] Tobias Grosser, Hongbin Zheng, Ragesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly –

polyhedral optimization in LLVM. In C. Alias and C. Bastoul, editors, *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Chamonix, France, 2011.

[9] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[10] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *2nd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Paris, France, January 2012.

[11] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule trees. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *4th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vienna, Austria, January 2014.

ACKNOWLEDGMENTS

Many colleagues, friends and family members supported me during the time of my doctoral studies, but also during the years leading towards them. Those influences have been important both in forming this work, but also in forming myself personally. I would like to express my deep gratefulness to all who have walked with me this path, the ones I will name personally, but especially the ones who I will inevitable forget to name.

First and foremost I would like to thank you, Albert, Sven and Saday, you who guided me through these three years of doctoral studies. Albert, I still remember the moment in a Canadian bar when you told me I should write you an email in case I would like to do my doctoral studies with you. From this moment, several years before I actually started, you have been immensely supportive, leaving me the freedom to choose the directions I want to follow, but still supporting me at any moment I was in need for guidance or help. Sven, working with you has been an experience. I must admit it was hard to read your first emails with your always very direct and open feedback, but I learned quickly how valuable your in-depth feedback is and now really enjoy working and discussing with you. Saday, thank you for your supervision and support both during my time in Ohio, but especially during the many phone calls and Paris visits throughout the last three years. You did not only help me to shape my research direction, but also gave helpful advices in so many other ways.

I would like to send a big thank you to Passau and the LooPo team, especially naming Martin Griebl, Christian Lengauer, Armin Größlinger and Andreas Simbürger as well as Dirk Beyer and Sven Apel. All of these supported me from the first years of my studies. Throughout my PhD I met and got to know many interesting researchers. I would like to especially thank Louis-Noël Pouchet, who I have been working with in Ohio, Uday Bondugula, who I visited in Bangalor and Ram (Ramanujam), who I also met very regularly at many different places. I would like to thank Aart Bik, Beate List and Stephen Hines for my great time at Google and their support in the context of my Google Doctoral Fellowship, Anton Lokmotov for the great time at ARM as well as Sebastian Pop, who I regularly interact with since my first internship at AMD. The Parkas team, including Mark, Francesco, Louis, Tim, Jean, Boubacar, Serge, Michael, Jun, Antoine, Feng, Guillaume, Riyadh, Adrien, Nhat, Robin, Cédric and Chandan for the many interesting discussions, the nice coffee breaks as well as joint travels excursions. Assia and Joelle as well as all the other administrative staff, who did a great job in supporting

my travels and my conference organization. I would also like to thank you, Arnaud, Sylvestre and Duncan, for the great time when organizing the LLVM conference and our regular Paris social. As well as the LLVM and Polly open source community with many very nice people, starting to name them would inevitably make me forgot someone important. Finally, there are my thesis reviewers Mary Hall and Franz Franchetti as well as my thesis committee with Stef Graillat and Jean-Luc Lamotte who I would like to express my thankfulness for their efforts and comments.

I also had the support of many great friends, including Yulia who helped with my registration, Amalia who helped with my English and my roommates Mathieu, Gælle, Caro and Kristin, who supported my crazy working hours and always prepared great food for me.

I also would like to thank my parents and siblings, Rolf, Waltraud, Tanja and Michael, who I neglected way too many times as well as my you Kasia, the woman on my side, who I met right at the beginning of my studies and who had an infinite amount of patience supporting me while working on this thesis.

CONTENTS

List of Figures xvii
List of Tables xviii
List of Listings xviii
List of Acronyms xix

i INTRODUCTION 1
1 INTRODUCTION 3
1.1 Outline 6

ii BACKGROUND 9
2 POLYHEDRAL COMPILATION 11
2.1 Mathematical Foundations 11
2.1.1 Integer sets 11
2.1.2 Integer maps 13
2.1.3 Named unions sets/named union maps . 15
2.1.4 Libraries for integer sets / maps 15
2.2 Model and Transform Imperative Programs 16
2.2.1 An illustrative example 16
2.2.2 What programs can be modeled? 19
2.2.3 The polyhedral representation 19
2.2.4 Transformations 20

iii TILINGS AND OPTIMIZATIONS FOR STENCILS . . . 25
3 STENCIL COMPUTATIONS 27
3.1 What are Stencil Computations? 27
3.2 Tiling of Stencil Computations 29
3.3 Related Work 32
3.4 Our Work on Stencil Computations 35
4 SPLIT TILING 37
4.1 Overview 37
4.2 Preprocessing 39
4.3 The Split Tiling Schedule 40
4.3.1 Core algorithm 40
4.3.2 Tile shape simplification 42
4.3.3 Multi-statement loop nests 42
4.4 CUDA Code Generation 44
4.4.1 Shared memory usage 45
4.4.2 Instruction level parallelism 46
4.4.3 Full/partial tile separation 46
4.5 Summary 46
5 HYBRID HEXAGONAL/PARALLELOGRAM TILING . . 49
5.1 Overview 49

5.2 The Hybrid Hexagonal/Parallelogram Schedule . 52
5.2.1 Hexagonal tiling 52
5.2.2 The parallelogram tile schedule 57
5.2.3 Intra-tile schedules 57
5.2.4 Hybrid tiling 58
5.2.5 Tile size selection 59
5.3 CUDA Code Generation 59
5.3.1 Generating CUDA code 59
5.3.2 Shared memory 60
5.3.3 Interleaving computations and copy-out . 61
5.3.4 Stencil specific code generation heuristics . 61
5.4 Summary 63
6 UNIFICATION WITH DIAMOND TILING 65
6.1 Diamond Tiling 66
6.1.1 The pluto optimizer 66
6.1.2 The diamond tiling extensions 68
6.1.3 Relation between tile sizes and wavefronts 68
6.1.4 Optimal tiles with default wavefront 73
6.2 Unified Diamond and Hexagonal Tiling 74
6.3 Tile Sizes that Maximize Compute/Communication 77
6.4 Summary 82
7 EXPERIMENTAL RESULTS 83
7.1 Split-tiling 83
7.2 Hybrid-Hexagonal 83
7.2.1 Comparison with state-of-the-art tools . . 83
7.2.2 Hybrid tiling and shared memory 87
7.3 Summary 89

IV POLYHEDRAL BUILDING BLOCKS 91
8 THE CONCEPT 93
9 POLYHEDRAL EXTRACTOR 97
9.1 Overview 99
9.2 Constructing a Polyhedral Representation 100
9.2.1 Access relations 101
9.2.2 Conditions 102
9.2.3 Loops 102
9.2.4 Schedule 103
9.3 Additional Features 103
9.3.1 CLooG specific features 104
9.3.2 Support for unsigned integers 104
9.4 Related Work 106
9.5 Limitations and Future Work 109
9.6 Summary 109
10 AST GENERATION 111
10.1 A new approach to AST generation 113
10.2 Input 117

10.3 Abstract Syntax Tree . . . 118
10.4 New AST Generation features . . . 118
10.4.1 Fine grained option mechanism . . . 119
10.4.2 Isolation . . . 121
10.4.3 Polyhedral unrolling . . . 122
10.4.4 Partial Unrolling . . . 125
10.4.5 Generating AST Expressions . . . 127
10.5 Experimental Results . . . 128
10.5.1 Existentially quantified variables . . . 128
10.5.2 Performance of AST generation strategies 132
10.5.3 Generation Time . . . 133
10.6 Related Work . . . 134
10.7 Summary . . . 135
11 SCHEDULE TREES . . . 137
11.1 Schedule Uses . . . 139
11.1.1 Original execution order . . . 139
11.1.2 Transformations . . . 140
11.1.3 AST generation . . . 141
11.2 Schedule Representations . . . 141
11.2.1 Properties . . . 142
11.2.2 Comparison . . . 143
11.3 Schedule Tree Representation . . . 147
11.3.1 Nodes . . . 147
11.3.2 Operations . . . 149
11.4 Hybrid hexagonal-parallelogram tiling . . . 151
11.5 Summary . . . 153

V LOW-LEVEL COMPILERS . . . 155
12 CONTRIBUTIONS TO LLVM / POLLY . . . 157
12.1 Compute out . . . 157
12.2 AST Generation . . . 157
12.3 GPolly - Automatic GPU offloading . . . 158
12.4 Representing parallelism . . . 159
13 OPTIMISTIC DELINEARIZATION . . . 161
13.1 Motivating example . . . 163
13.2 Problem statement . . . 164
13.3 Array views with single-parameter sizes . . . 166
13.3.1 Multiple array references . . . 169
13.3.2 Array sizes in subscript expressions . . . 170
13.3.3 Arrays of size $A[*][\beta_1 P_1][\beta_2 P_2]$ . . . 172
13.4 Arrays of size "parameter + constant" . . . 172
13.5 Implementation . . . 179
13.6 Experimental evaluation . . . 180
13.7 Related work . . . 181
13.8 Summary . . . 182

VI CONCLUSION 185
14 CONCLUSION 187
14.1 Personal contributions 187
14.2 Future work 188

VII APPENDIX 191
BIBLIOGRAPHY 193

LIST OF FIGURES

1 A two-dimensional integer set (dense) 12
2 A two-dimensional integer set (sparse) 13
3 A two-dimensional integer map 14
4 Iteration Space – Unmodified 17
5 Iteration Space – Tiled 18
6 Heat distribution after running the stencil in Listing 1 28
7 Data-flow of the stencil in Listing 1 28
8 Iteration space of 1D-space stencil 29
9 Rectangular tiling of 1D-space stencil 30
10 Iteration space of 1D-space stencil 31
11 Split tiling for the simple example (tile size 8×3). 38
12 Split tiled jacobi-2d kernel 43
13 Two statement kernel 44
14 1D Hexagonal tiling - Created from 1D Split tiling 50
15 Jacobi 2D stencil 51
16 Generated PTX (CUDA bytecode) 51
17 Opposite dependence cone 53
18 A hexagonal tile 53
19 Hexagonal tiling pattern 55
20 n-dimensional tile schedule (± 1 distances) 58
21 Symmetric dependences & square tiling (original/transformed) 69
22 Symmetric dependences & non-square tiling (original/transformed) 69
23 Asymmetric dependences & square tiling (original/transformed) 71
24 Multiple time steps. Square tiles reduce parallelism. (original/transformed) 71
25 Multiple time steps. Non-square tiles maximize parallelism. (orig./trans.) 72
26 Diamond tiling (original/transformed) 72
27 Hexagonal-tiling (original/transformed) 72
28 The stretching in the transformed space (unstretched/stretched) 76
29 1D hexagonal tiling ($T = 6, B = 4$) 78
30 The first two time steps of 1D hexagonal tiling ($T = 6, B = 4$) 79
31 Compute-to-read ratio - Hexagonal vs. diamond tiling 80
32 Compute-to-sync ratio - Hexagonal vs. diamond tiling 81
33 Split-tiling performance (desktop GPU) 84
34 Split tiling performance (mobile GPU) 84
35 Copy code from hybrid hexagonal/parallelogram tiling (a single loop) 114

36 Copy code from hybrid hexagonal/parallelogram tiling (unrolled) 116
37 Example Program 117
38 Interleaved schedule without code generation options 119
39 Interleaved schedule with code generation options 121
40 Modulo conditions (examples not supported by CLooG) 129
41 Existentially quantified variables (examples not supported by CLooG/codegen+) 130
42 Example schedule tree representation 138
43 Band forest representation of the schedule in Figure 42 146
44 Fuse bands B_1 and B_2 150
45 Order the active statement instances at B according to filters F_1 and F_2 150
46 Input pattern for hybrid tiling 151
47 Output pattern for hybrid tiling 152
48 Linearized expression for multi-dimensional array of constant size 161
49 Linearized expression for multi-dimensional array of parametric size 162
50 Subarrays accesses for different parameter values 168

LIST OF TABLES

1 Performance on NVIDIA GTX 470: GStencils/second & Speedup 85
2 Performance on NVS 5200: GStencils/second & Speedup 85
3 Characteristics of Stencils 86
4 Optimization steps: GFLOPS & Speedup 88
5 Performance counters (units of 10^9 events) 89
6 Features of different polyhedral extractors 108
7 AST generation strategy based performance (GFLOPS) 132
8 Comparison of some generic schedule representations 144

LIST OF LISTINGS

1 Implementation of a 5-point, 2D-space, jacobi-style heat stencil 27
2 Split tiled code 39
3 A trivial program 100

4 Part of CLooG output for thomasset test case 104
5 Unsigned operation in loop bound 105
6 Invalid fusion of program in Listing 5 105
7 Loop with unsigned iterator 105
8 Trivial unrolling example 123
9 Unrolling in the presence of strides 123
10 Unrolling in case of bound, non-constant number of iterations 124
11 Unrolling with two lower bounds 125
12 Partial unrolling - original loop nest 125
13 Partial unrolling - tiled 126
14 Partial unrolling - tiled + unrolled 126
15 Partial unrolling - tiled + unrolled + isolated core computation 127
16 A single loop marked parallel using LLVM metadata 159
17 Nested loops marked parallel using LLVM metadata 159
18 A gemm kernel written in C99 using variable length arrays 162
19 A gemm kernel written using manually implemented multi-dimensional arrays. 164
20 Array dimensions used in subscripts 169
21 A gemm kernel written in C++ using boost::ublas 180
22 A gemm kernel written in Julia 180

LIST OF ACRONYMS

AST Abstract syntax tree

CLooG Chunky loop generator

clang The LLVM C/C++ compiler

CPU Central processing unit

CUDA NVIDIA's parallel computing platform

DSL Domain specific language

DRAM Dynamic random-access memory

FPGA Field Programmable Gate Array

gcc gnu compiler collection

GPU Graphics processing unit

GPGPU General-purpose computing on graphics processing units

ILP Integer linear programming

IP Intellectual property

IR Intermediate representation

isl Integer set library

LLVM LLVM compiler infrastructure

LLVM-IR LLVM intermediate representation

OpenCL Open compute language

OpenMP Open multi-processing

PPCG Polyhedral parallel code generator

PTX Parallel thread execution (NVIDIA IR)

SCC Strongly connected component

SCEV Scalar evolution

SCoP Static control part/program

SIMD Single instruction multiple data

SPU Stream processing unit

Part I

INTRODUCTION

1 INTRODUCTION

The steadily growing complexity of problems solved in scientific and high performance computing has caused a continuous hunger for compute power. The same growth can be seen with mobile devices, which have been turned into personal super computers driven by the increasing demands of gaming and image processing. Looking back twenty years, the most powerful supercomputer as published on the TOP500 supercomputer list [52] of July 1994 had a theoretical peak performance of 236 GFLOPS (double precision) with a power consumption of almost 500.000 Watts. Today, NVIDIA claims a peak performance of 365 GFLOPS (single precision) with 5 Watt power consumption for their recently announced mobile platform Tegra K1 [45]. Even though those numbers are not directly comparable, they show clearly that large compute capabilities have reached mobile. And, with 54,902 TFLOPS peak performance at the top of today's Top500 list [52], supercomputers did not stand still either.

A major factor for this enormous progress in compute power and energy efficiency is, besides others, the increasingly *specialized* and *heterogeneous* hardware. Both mobile devices and super computer nodes rely today on multi or many cores, short vector instructions, various levels of caches and often dedicated accelerators. For programs to benefit, they need to be optimized to effectively exploit the available hardware.

In the world of supercomputers it was common to optimize important programs manually. Today manual optimization is, even on supercomputers, increasingly complemented with automatic program generation and search space exploration. In the mobile market the sheer number of different hardware platforms makes manual optimization impractical. This becomes evident just by looking at the mobile GPU market alone, where there are over ten entirely different hardware designs a program needs to be tuned for. IP suppliers such as ARM, DMP, Imagination Technologies, or Vivante and vertically integrated suppliers such as AMD, Intel, Nvidia, and Qualcomm, all provide their own designs, in addition to which vendors such as Samsung and Broadcom use their own internal designs [111]. On Android, the most widely used mobile platform, direct accelerator access is not even possible. Instead the RenderScript compute interface was designed with automatic performance optimization as a design goal.

> *"While testing and tuning a variety of devices is never bad, no amount of work allows them to tune for unreleased hardware*

> *they don't yet have. A more portable solution places the tuning burden on the runtime, providing greater average performance at the cost of peak performance."* [Jason Sams, RenderScript Tech Lead, 112]

We can conclude that advances in automatic optimization of compute programs are important.

Automatic compiler optimizations have a long history [14]. In the context of this work, the optimization of loop programs is of particular interest. Loop optimizations had already been investigated in the context of FORTRAN compilers over thirty years ago. Together with early work on data dependence analysis [22], as well as the introduction of the data dependence graph [85], loop transformations such as fusion and fission have been introduced to improve program performance e.g., by enabling better use of vector hardware. For the very same reason automatic loop interchange has been discussed [15] and, to reduce expensive memory movements, loop blocking [12, 139] was introduced. Even though the precision of data dependence computation has increased throughout the years and loop transformations have evolved notably, the basic concept of taking a loop nest and optimizing it step-by-step by applying a set of individual loop transformations remains important today, with many production C/C++/FORTRAN compilers relying on the use of these "classical loop optimizations". Important work on classical non-trivial loop transformations has been performed in several commercial compilers such as the KAP compiler or icc, but also in research compilers such as Parafrase [98], Polaris [34], PIPS [74], SUIF [136] or Cetus [50].

Even though classical loop transformations are well understood, finding and reasoning about the right sequence to apply them in is hard. To address this problem researchers have investigated the possibility of computing an entirely new loop structure [58, 57] directly from a set of data dependences without ever applying any loop transformations, or of deriving combinations of classical loop transformations using a single unifying loop transformation theory [137, 138]. As a result of this research the polyhedral model (Chapter 2) evolved, a generic and compiler independent way to reason about complex loop nests and their transformation. Within the last two decades this model has been heavily investigated and regularly serves as an instrument to address challenging optimization problems. In fact, it often enables transformations difficult or even impossible to describe with classical loop transformations. One such use case are recent C to GPU code translators [24, 135] which have shown that polyhedral techniques can even be used for complex program transformations such as the generation of code to exploit software managed caches. Another interesting use case is a recent paper discussing the combination of SIMD code generation and polyhedral loop optimization techniques [83], a nice example of how to parameterise target inde-

pendent loop transformations to obtain competitive platform specific code.

Even though polyhedral loop transformation frameworks have many benefits, at the moment they are, with the exception of IBM XL [36], mainly employed by research focused compilers [87, 35, 135, 40, 136]. Existing C/C++ open source compilers such as gcc or LLVM currently do not take advantage of polyhedral loop transformations in their default optimization sequence, despite the existence of loop transformation frameworks such as gcc/graphite [122] and LLVM/Polly [3]. As both compilers also do not incorporate an extensive set of classical loop transformations, programs [65] are required to incorporate program specific transformations in their source code which reduces performance portability and increases code bloat.

Domain or problem specific solutions are also hindered by the lack of generic loop transformation systems. Julia [31], a just-in-time compiled language specialized on scientific computing, does for example not leverage any loop optimization opportunities, even though their scientific compute kernels would strongly benefit. "Automatic" GPU code generation still requires explicit user annotated kernel code as proposed for example by openacc [64]. Anything more complicated is left to DSL compilers which hardcode specific transformations [70, 68]. Even though DSL compilers commonly obtain great performance, the ad hoc implementation of the required loop transformation causes redundant work and, most importantly, the benefits a generic and formal transformation framework would bring are missed. Overall, there is a strong need for making generic loop transformations more accessible.

Motivated by the missing reuse of loop optimization and the growing, but still lacking, adoption of polyhedral loop optimization techniques that could facilitate such reuse, we aim in this work to widen the scope in which polyhedral loop optimizations can be used effectively. As a driving force we develop a set of advanced domain specific loop optimizations that enable the efficient execution of stencil computations on GPUs. Stencil computations (Chapter 3) as discussed in this work are computations that iteratively recompute the elements of a (multi-)dimensional data space, just from the neighborhood of each individual element. Optimizing their execution is difficult and often requires sophisticated optimization strategies. However, in contrast to many existing works, we specifically avoid the implementation of a specialized optimizer. Instead, we aim for a solution that clearly separates domain and target specific optimizations from non-problem specific code generation strategies and techniques necessary to obtain peak performance. We do this by developing new problem specific tiling schemes, which are translated to highly performant code by a general purpose infrastructure that has only been parameterized to ensure the generation of optimal code. As a result

we hope to show that polyhedral loop optimization techniques can be successfully used even in highly performance critical codes without the need to implement specialized code generation strategies.

Following the concept of reusability, we aim to make loop transformations less compiler dependent to facilitate the transfer of optimizations between different generic and domain specific compilers. To reach this goal we do not only take special care throughout this work to highlight and address corner cases that may inhibit the use of our work in production, but we specifically present a set of polyhedral building blocks. Such building blocks do not only enable our new stencil optimizations, but they are designed for reusability in the context of different domain specific and general purpose compilers. Going one step further, we present new work in the context of the LLVM compiler infrastructure that prepares the path for later integration of our optimizations in a low-level static compiler.

1.1 OUTLINE

After some background information on integer sets and polyhedral compilation given in Chapter 2, the following topics will be discussed in this thesis.

Tilings and Optimizations for Stencils

Our work on developing new tiling schemes that allow the efficient execution of iterative stencil computations and similar compute patterns on GPUs.

- In Chapter 4 we discuss the implementation of a split tiling scheme that enables the time tiled execution of stencil computations on GPUs ensuring balanced coarse-grained parallelism without the need for redundant computations. We discuss the implementation of this tiling scheme in a general purpose polyhedral optimizer, including a set of optimizations that enable the use of software managed shared caches on GPUs.
- In Chapter 5 we discuss an extension and specialization of our previous work on split tiling. The result is a new tiling scheme combining hexagonal tiling on one dimension with parallelogram tiling on the remaining dimensions. This new tiling scheme allows us to better address important GPU specific concerns now being able to ensure the absence of thread divergence and the use of coalesced and aligned loads. As a result, we obtain a tiling scheme that can, even for 3D test cases, profitably use software managed shared caches on GPUs.

- In Chapter 6 we present a detailed theoretical analysis of diamond tiling, a tiling scheme closely related to our hexagonal tiling work. In the context of this analysis we present new insights on constraints that diamond tiling poses on tile-sizes and on wavefront coefficients. We complement these results with a set of conditions that ensure that the integer points contained in different tiles are placed at identical offsets. We then unify the two tiling schemes by providing a formulation of hexagonal tiling for 2 dimensional problems (1 time dimension, 1 space dimension) in the framework of diamond tiling. We complete this chapter with an analysis of tile sizes that yield optimal compute-to-communication and compute-to-synchronization ratios.

Polyhedral Building Blocks

A set of polyhedral building blocks that have been developed to facilitate and generalize the use of polyhedral schedule and data layout transformations and their use in imperative programs.

- In Chapter 9 we present a polyhedral extraction tool that, by combining a real C compiler with an advanced integer set library, pushes the limits of the amount of statically analyzable code that can be translated to a polyhedral representation. By presenting how to model integer wrapping with modulo constraints, we ensure correctness even in difficult areas of the C standard.

- In Chapter 10 we present a new AST generator with complete support for Presburger relations including support for piecewise schedules and their use to express index set splitting as a schedule-only transformation. Going beyond the generation of control flow, we support the generation of AST expressions from arbitrary user provided piecewise quasi affine expressions. New simplification methods for AST expressions allow the exploitation of local context information to generate fast modulo and division operations. With fine grained options we give the user precise control over AST generator decisions. Finally, we provide support for heavy specialization through polyhedral unrolling and user directed versioning.

- In Chapter 11 we provide a detailed analysis of existing approaches to describe polyhedral schedules as well as their uses within the polyhedral tool chain. We then derive a new approach to describe polyhedral schedules which makes the inherent tree structure of polyhedral schedules explicit and which we use to provide a more intuitive formulation of the schedule generated by hybrid-hexagonal tiling.

Low-level Compilers

- We will briefly discuss various projects that have taken place in the context of Polly, our high-level loop optimizer for LLVM. This includes the need for compute outs to bound compile time, GPolly a project to bring automatic GPU code generation to a low-level compiler, our experience with using the isl AST generator in Polly as well as work on how to annotate parallelism on LLVM-IR.

- In Chapter 13 we discuss a new approach to delinearize a multivariate polynomial expression to an access to an array shape of parametric size. In our work we show how optimistic delinearizations enables us to delinearize expressions where the delinearization can not be proven statically. We evaluate our work on code written with Julia, boost::ublas and C99 variable length arrays.

Part II

BACKGROUND

2 POLYHEDRAL COMPILATION

Polyhedral compilation uses a compact mathematical representation to precisely model the individual elements of a compute program. The use of a solid mathematical model enables detailed static analysis and exact transformations. Dynamic extensions [30] are available for programs that lack static information. The following chapter gives an introduction to the mathematical concepts and explains how they are used to model, analyze and transform compute programs.

2.1 MATHEMATICAL FOUNDATIONS

(Integer) polyhedra [89] or Presburger relations [106, 105] are the mathematical foundations of polyhedral compilation. In this section we give an introduction to the most important concepts. The notations [131, 130] presented are the ones proposed by isl [128], the integer set library we use for our work. isl itself is built around Presburger relations and uses many ideas originally introduced in the Omega project [81]. Another important concept used at the core of isl is parametric integer programming [55].

2.1.1 *Integer sets*

Integer sets as defined by isl are sets of integer tuples from $\mathbb{Z}^d$ described by Presburger formulas. $S = \{(i,j) \mid (a \leqslant i,j \wedge i+j < b) \vee (4 \leqslant i,j \wedge i \leqslant b \wedge j \leqslant 6)\}$ is an example of a two-dimensional integer set described in terms of two parameters a and b. Figure 1 illustrates S for $a = 1$ and $b = 8$. It shows a triangular red shape containing red points and a rectangular blue shape containing blue squares. Only the red points/blue squares located at the integer coordinates within the colored shapes are the actual elements of S. The colored shapes have been derived from the description of the integer set. They form a set of convex shapes that enclose the elements of S and help to visualize S. As there are generally different ways to represent an integer set, such visualizations are not unique.

Figure 2 illustrates a second set $S' = \{(i,j) \mid (a \leqslant i,j \wedge i+j < b \wedge (i) \bmod 2 = 0) \vee (4 \leqslant i,j \wedge i \leqslant b \wedge j \leqslant 6 \wedge i \bmod 2 = 0)\}$. It is similar to S with the only difference being that additional modulo constraints have been added that permit only even values on the i-dimension. Visualizing S' with a set of convex shapes that enclose its elements is hindered by the modulo constraints. Therefore, we

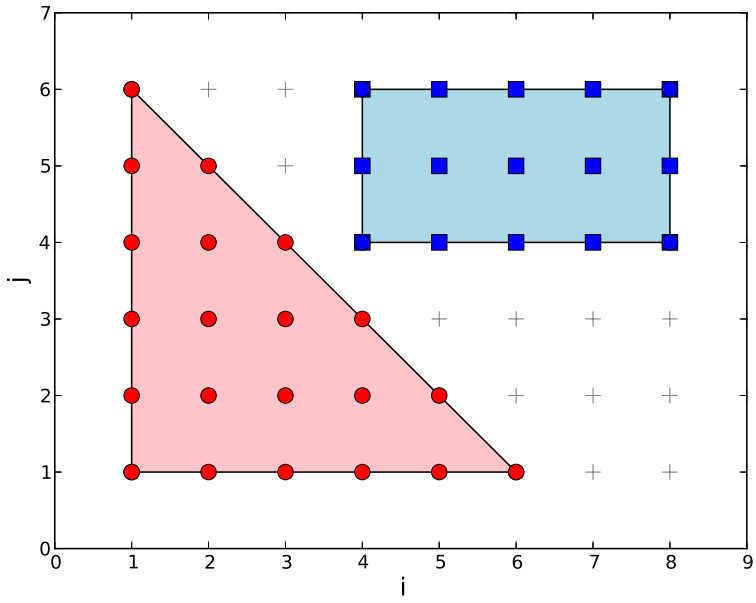


Figure 1: A two-dimensional integer set (dense)

illustrate S' by directly highlighting the points and squares that are part of the integer set.

In general, an integer set has the form

$$S = \{\vec{s} \in \mathbb{Z}^d \mid f(\vec{s}, \vec{p})\} \tag{1}$$

with $\vec{s}$ representing the integer tuples contained in the integer set, d the dimensionality of the set, $\vec{p} \in \mathbb{Z}^e$ a vector of e parameters and $f(\vec{s}, \vec{p})$ a Presburger formula that evaluates to true, *iff* $\vec{s}$ is element of S for given parameters $\vec{p}$.

A Presburger formula p is defined recursively as either a boolean constant ($\top, \bot$), the result of a boolean operation such as negation, conjunction, disjunction or implication ($\neg p$, $p_1 \wedge p_2$, $p_1 \vee p_2$, $p_1 \Rightarrow p_2$), a quantified expression ($\forall x : p$, $\exists x : p$) or a comparison between different quasi-affine expressions ($e_1 \oplus e_2, \oplus \in \{<, \leqslant, \geqslant, >\}$). A quasi-affine expression[1] e is defined as a plain integer constant (e.g., 10), a parameter, a set dimension or a previously introduced quantified variable. It can also be constructed recursively as the result of a unary negation of a quasi-affine expression ($-e$), a multiplication of an integer constant with a quasi-affine expression (e.g., $10e$), an addition/subtraction of two quasi-affine expressions ($e_1 \oplus e_2, \oplus \in \{+, -\}$), an integer division of a quasi-affine expression by a constant (e.g., $\lfloor e/10 \rfloor$) or the result of computing a quasi-affine expression modulo

1 We use the term quasi-affine [54, 55] to describe the potential use of existentially quantified variables which are, besides set and parameter dimensions, needed to internally model constructs such as integer divisions, the modulo operations or quantified expressions as affine inequalities.

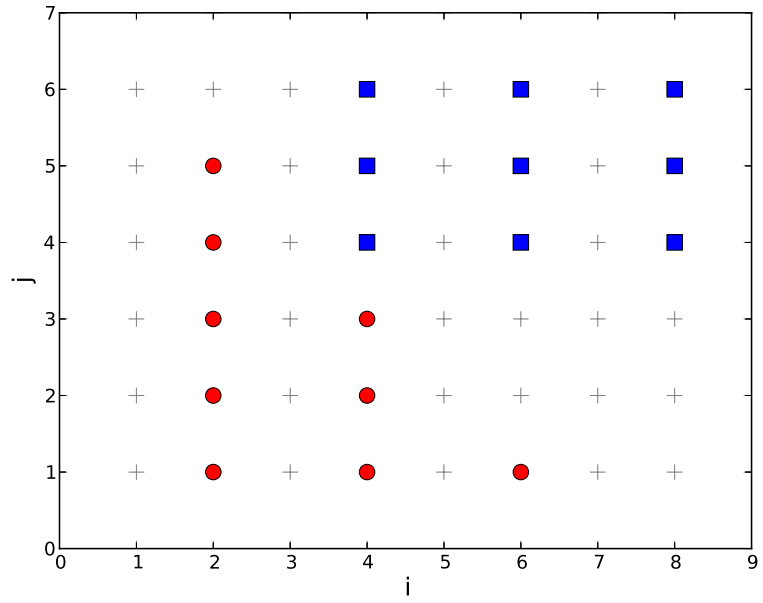


Figure 2: A two-dimensional integer set (sparse)

a constant (e.g., e mod 10). The set $\{\vec{s} \mid \top\}$ is called the universal set or the universe and is commonly abbreviated as $\{\vec{s}\}$.

There are also various properties and operations that can be computed on integer sets. We present the most important ones. Given a single set we can check if the set is empty, if it is the universal set or if a certain dimension always has a fixed value. We can also ask for a sample value contained in the set, for the minimal/maximal value of a certain dimension or the lexicographically smallest/largest element of a set. Two sets can be checked for equality, disjointness or the existence of a subset relation. It is also possible to derive new sets by computing the complement of a set, by projecting out dimensions from a set or by adding new dimensions to a set. A set can be approximated by computing various hulls (convex, affine, simple, polyhedral). Two sets can be combined by intersecting them, computing their union or the difference between them. The properties and operations presented generally follow known mathematical semantics and notations. In case uncommon notations are used they will be explained at the point of usage.

2.1.2 *Integer maps*

Integer maps are binary relations between integer sets. The first set in the relation is called the *domain* or the input set, the second set is the *range* or the output set. Integer maps are modeled as pairs of integer tuples from $\mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2}$.

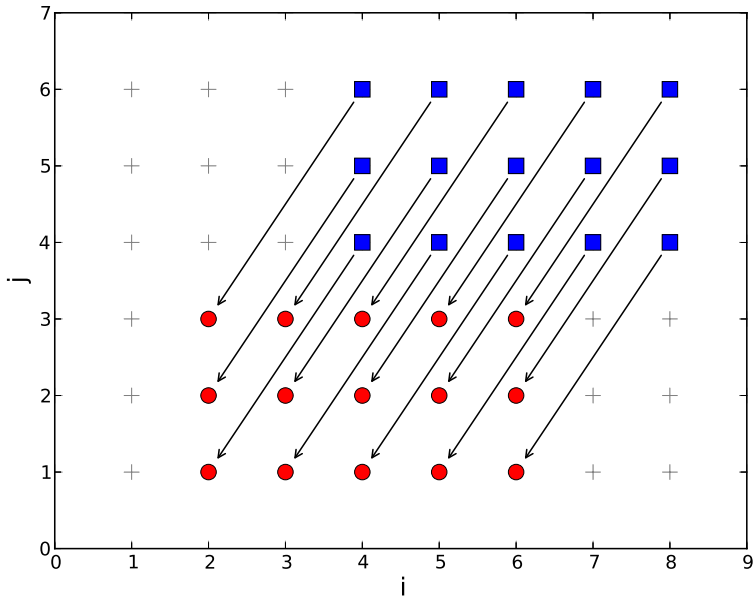


Figure 3: A two-dimensional integer map

Figure 3 illustrates the integer map $M = \{(i, j) \to (i-2, j-3)\}$ with the input values restricted to the elements contained in the blue rectangular of Figure 1. Each black arrow represents a relation between one input tuple and one output tuple. The input values (blue squares) shown are the very same values as illustrated in Figure 1. The output values (red circles) are the same values, but translated according to M.

The general form of an integer map is

$$M = \{\vec{i} \to \vec{o} \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid f(\vec{i}, \vec{o}, \vec{p})\} \tag{2}$$

where $\vec{i}$ describes the d_1-dimensional input tuples, $\vec{o}$ the d_2-dimensional output tuples, $\vec{p} \in \mathbb{Z}^e$ a vector of e parameters and $f(\vec{i}, \vec{o}, \vec{p})$ a Presburger formula that evaluates to true, *iff* $\vec{i}$ and $\vec{o}$ are related in M for given parameters $\vec{p}$. Integer maps can represent arbitrary relations and can, contrary to what the use of the "$\to$" notation suggests, relate multiple output values to a single input value. The Presburger formulas that describe integer maps follow the rules presented for integer sets with the only difference, that they now reference $\vec{i}$ and $\vec{o}$ instead of $\vec{s}$. The description of an integer map $\{\vec{i} \to (o_1, \ldots, o_{d_2}) \mid o_1 = f_1(\vec{i}, \vec{p}), \ldots, o_{d_2} = f_{d_2}(\vec{i}, \vec{p})\}$ is often syntactically shortened to $\{\vec{i} \to (f_1(\vec{i}, \vec{p}), \ldots, f_{d_2}(\vec{i}, \vec{p}))\}$, as shown in our example.

Various properties and operations are defined on integer maps. Most of them are similar to the ones on integer sets, but can often be applied to either the entire map or just its range or domain. One interesting operation is the "application" of a map to a set. In the previous example, we can *apply* M to the blue squares and the result

will be the set of red circles. Similarly, it is possible to apply maps to the ranges or domains of other maps. The result of this "chaining" of maps is a map that performs the combined mapping of the input maps. It is also possible to reverse maps to change the meaning of input and output dimensions, as well as to obtain the input set (domain) or the output set (range) of a map.

2.1.3 *Named unions sets/named union maps*

It is possible to name integer sets and maps according to where they are used or what they represent. Named integer sets are integer sets which contain named tuples. An example is the set {S(i,j)} which is an integer set that lives in a two-dimensional space with the name "S". We define named integer maps as integer maps between two named spaces. The map $\{S(i,j) \rightarrow A(i)\}$ is an example of a named integer map. It is also possible to define integer sets that contain tuples from different spaces, e.g. $\{S1(i,j); S2(i)\}$. Such sets are called (named) union sets. Named union maps are defined accordingly. Operations that can be performed on sets and maps can often be performed on union sets and maps. In case we use such operations with non-obvious semantics, the actual semantics will be explained at the point of use. We may omit the prefix 'named' or 'union' in cases where it is either obvious or not relevant for the discussion.

2.1.4 *Libraries for integer sets / maps*

There exist several libraries that can express and modify integer sets or related data structures. `isl` [128], as discussed earlier, is the integer library we use for our work. It directly supports the operations and notations given above. Aside from isl, there is the Omega library [81] which, having served as a source of inspiration for isl, supports similar notations and operations. However, omega does not support the concepts of named sets (maps) or union sets (maps). isl and omega differ in the algorithms and internal data structures used. While isl relies on integer division in its internal representation, omega uses intersections of polyhedra and lattices to represent integer sets and maps.

In certain (common) situations it is also possible to approximate integer sets by a set of rational polytopes that enclose the points in the integer set. `PolyLib` [88] and `PPL` [20] are libraries which have been originally developed to perform computations on such rational polytopes. In case computations can be performed on rational polytopes, their use may reduce the theoretical (and practical) computational complexity at the cost of a reduced precision. PolyLib later gained support for computations on so called $\mathbb{Z}$-polyhedra, polyhedra that model only the integer points they contain. PPL also supports inte-

ger lattices and allows the creation of objects that have the properties of both lattices and polyhedra, a feature that can possibly be used to model integer sets. While representing integer sets with the above libraries is possible, our experience has shown that doing so can be inconvenient. First, there is a certain risk for the library user to accidentally use rational computations for operations where integer computations are needed. This may result in incorrect results being obtained. Second, to model certain integer sets correctly additional helper dimensions (existentially quantified dimensions) need to be introduced, which the user of these classical libraries needs to keep track of manually. In contrast, native integer set libraries keep track of such dimensions automatically. As we make heavy use in our work of features that require existentially quantified dimensions (modulo constraints and integer divisions), we choose to use an integer library that supports them natively.

2.2 MODEL AND TRANSFORM IMPERATIVE PROGRAMS

Polyhedra or, in our case, integer sets and maps can be used to model "sufficiently regular" compute programs with the goal to reason about and precisely control higher-level properties without distraction from imperative or lower-level constructs. To do so the individual statement instances in a program (i.e., each dynamic execution of a statement inside a loop nest), their execution order as well as the individual array elements accessed are modeled, analyzed and transformed, whereas control flow constructs, loop induction variables, loop bounds or array subscripts are hidden and only regenerated when converting a transformed loop nest back to imperative constructs.

2.2.1 *An illustrative example*

We start with a simple piece of code that consists of a single compute statement S, which is surrounded by two loops, the i-loop and the j-loop. Data is loaded and stored to a two-dimensional array A.

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

To model this computation we construct four data structures. An *iteration space* I, a *schedule* S as well as a relation of *read-accesses* A_{read} and a relation of *must-write-accesses* A_{write}. I is an integer set that describes the set of statement instances that is executed. It gives no information about the execution order of these statement instances. S is an integer map that assigns to each statement instance a possibly multi-dimensional time. It allows us to define an execution order

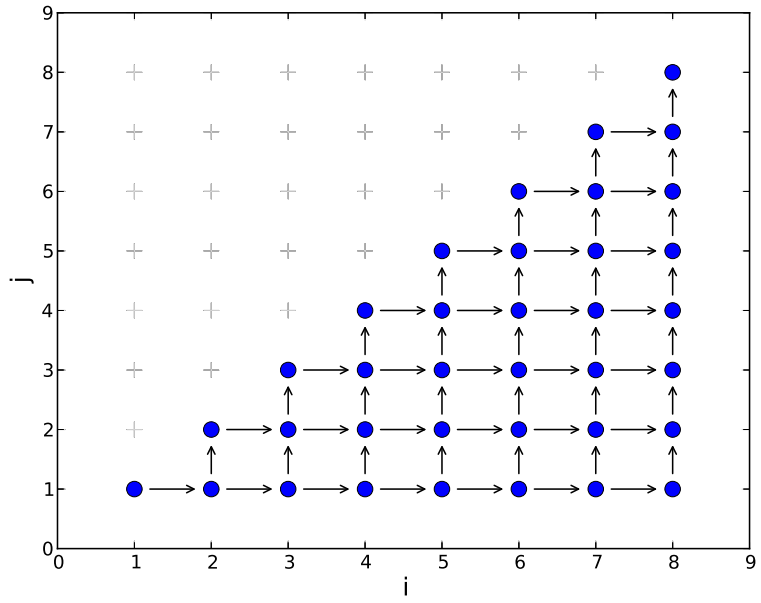


Figure 4: Iteration Space – Unmodified

by sorting the statement instances according to the lexicographic order of their time stamps. A_{read} and A_{write} are integer maps which define for each statement instance the memory locations that are accessed.

For our example, we get the following model:

$$\begin{aligned} I &= \{S(i,jj) \mid 1 \leqslant j \leqslant i \leqslant n\} \\ S &= \{S(i,j) \to (i,j)\} \\ A_{read} &= \{S(i,j) \to A(i-1,j); S(i,j) \to A(i,j-1)\} \\ A_{write} &= \{S(i,j) \to A(i,j)\} \end{aligned}$$

The illustration of I in Figure 4 represents each statement instance with a single dot. The arrows between such statement instances illustrate data flow dependences modeled by an integer map $D = \{S(i,j) \to S(i,j+1); S(i,j) \to S(i+1,j)\}$. Those dependences relate statement instances with the statement instances they depend on. Computing precise data-flow dependences [54, 104, 92] is one analysis that is significantly facilitated by the use of an integer set based representation.

We now try to improve data-locality by ensuring that statement instances that operate on the same data elements are executed in close time proximity. The data dependences force us to ensure that each statement instance is always mapped to a point in time that is later than the execution time of all statement instances it depends on. However, within these constraints we are free to modify the schedule. A common transformation to increase data locality is loop tiling. Loop

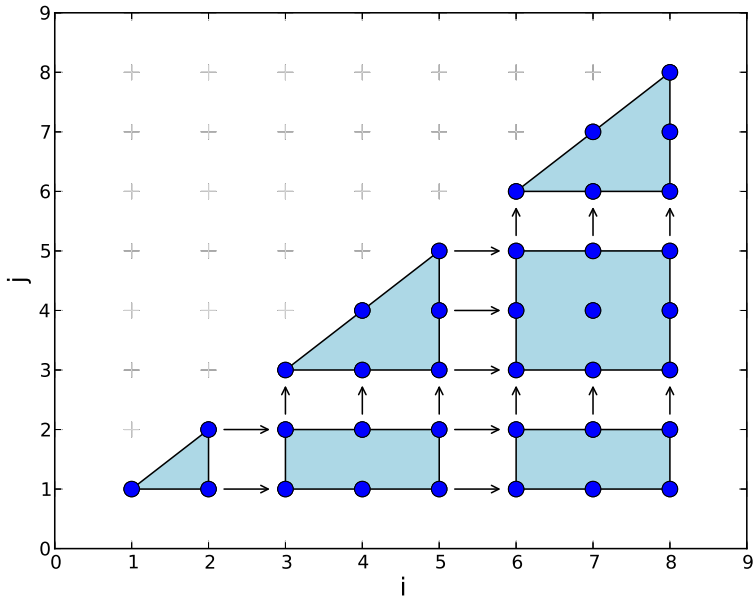


Figure 5: Iteration Space – Tiled

tiling is in this case both legal and effective. To implement loop tiling we define a new schedule

$$S' = \{S(i)(j) \to (\lfloor i/3 \rfloor)(\lfloor j/3 \rfloor)(i)(j)\}$$

which defines an execution order where the statement instances are always executed in blocks of size 3×3. The new execution order is illustrated in Figure 5 and is shown in two levels. At the higher level, the blue blocks are executed in lexicographic order. At the lower level, within the individual blue blocks, the statement instances are again executed in lexicographic order. As statement instances that are close by are placed in the same block, they are also executed close by in time.

Up to this point, the transformation has only be applied at the level of our abstract model. To materialize it, it is necessary to generate imperative code according to it. For this we use a polyhedral AST generator (discussed in Chapter 10) which translates our abstract schedules back into a set of imperative loops and conditions. For the example above the following AST could be generated.

```
for (i = 1; i <= n/3; i+=1)
  for (j = 1; j <= n/3; j+=1)
    for (ii = max(1,3 * i); ii <= min(n, 3*i+2); ii+=1)
      for (jj = max(ii, 3 * j); ii <= min(n, 3*j+2); jj+=1)
S:      A[ii][jj] = A[ii-1][jj-1];
```

2.2.2 *What programs can be modeled?*

A program that is "sufficiently regular" to be modeled, analyzed and transformed using integer sets and maps is traditionally called Static control part/program (SCoP) [56, 25]. What is considered a SCoP depends on what can be translated to an integer set based description and may look very different in the context of source-level C/C++ code [28][10], Graphical Dataflow Languages [33] or compiler IRs [122][3]. Ongoing research continuously widens the set of programs that can be modeled both by using approximation, but also by extending the underlying model [127, 30] and the generality of tools that extract this model. To give the reader an idea of how a SCoP may be defined, we present a set of common imperative constructs that can be translated into a polyhedral representation. For a precise and complete description of what state-of-the-art C/C++ code extractor can translate we refer to Chapter 9.

A SCoP is a program (region) that consists of a set of *statements* possibly enclosed by (not necessarily perfectly) *nested loops* and *conditional branches*. Within this region read-only scalar values are called *parameters*. The statements in the SCoP are side effect free, besides explicit *reads* and *writes* to multi-dimensional arrays or scalar values. Loops are regular loop bounds with a single lower and a single upper bound and increments by fixed, positive integers (`i+=10`). Both loop bounds and array accesses are (piecewise-quasi) affine expressions in terms of parameters and induction variables of outer loops.

2.2.3 *The polyhedral representation*

The representation we use to model SCoPs consists of the following components:

ITERATION SPACE/DOMAIN The set of statements instances that are part of a SCoP. It is modeled as a named union set, where each named component of the union set describes a statement, with individual instances of a statement being described by the elements contained in the corresponding named set.

ACCESS RELATIONS A set of read, write and may-write access relations relate statement instances to the data-locations they access. These access relations are modeled as named union maps.

DEPENDENCES A relation between statement instances that defines restrictions on the execution order, due to producer-consumer relationships or the shared usage of certain data locations. Data dependences are modeled as named union maps.

SCHEDULE An execution order which assigns each statement instance a multi-dimensional execution time. One statement instance is

executed before another statement instance, if its execution time is lexicographically smaller.

It is important to note that there is a strong separation between the statement instances themselves and the order in which they are executed. Program optimizations that do not change the set of statements that are executed, but only change the order they are executed in, consequently only affect the schedule.

We also want to note that there is a relation between the dependences and the schedule. A schedule is only valid if it is of a form such that all data-dependences go forward in time. This means, if we take the data dependences and apply the schedule to the related statement instances to translate them to the scheduling time they are executed at, the time at the source of the dependence must be lexicographically strictly smaller than the time at the target. Also, we know that given a set of dependences, we can compute a schedule and, vice versa, given an iteration space, access relations and a schedule the corresponding data dependences can be computed.

In contrast to many previous works we distinguish between (must) write and may write accesses. This distinction is important to understand if a certain write always overwrites a specific data-location and consequently makes previously written data inaccessible or if previously written data may possibly remain intact and accessible for later reads. This difference is important as in the former case data-dependences only need to be computed from the last write to subsequent reads whereas in the latter case data-dependences to all previous may writes up to the next must write access need to be computed. There is no need to track may read accesses separately.

2.2.4 *Transformations*

The use of integer sets or polyhedra to perform loop optimizations has a long tradition, both in the history of automatic parallelization, but also in the optimization of sequential code. Compared to classical loop transformations the use of integer sets as a mathematical model has several advantages. Besides the ability to use mathematical tools to reason about the possible transformations, they simplify the composition of transformations and they allow us to express transformations that are otherwise difficult or even impossible to express.

We can group loop transformations according to what program properties are modified:

- The order in which computations are performed
- The loop structure, but not the order of computation
- The data layout and the data locations accessed
- The computation (algorithmic changes)

The first three kinds of transformations are regularly modeled with integer sets, whereas modeling algorithmic changes is more difficult, but possible in some cases [140]. Integer sets are also used for accelerator programming and vectorization, but such transformations are mostly a combination of execution reordering and data layout transformations supplemented with the generation of specialized target instructions or library calls. We will use such GPU code generation techniques in Part iii. In this section we focus on basic computation reordering transformations, which will be used and extended later in this work.

Classical transformations that reorder computations are fusion, fission, reversal, interchange, strip-mining, and skewing. They all change the order statements instances are executed in and can consequently be modeled by schedule transformations. The same holds for combinations of such elementary transformations that yield tiling or unroll-and-jam. The subsequent examples illustrate these transformations as well as the corresponding iteration spaces $\mathcal{I}$, the original schedule $\mathcal{S}$, the transformation $\mathcal{T}$ and the transformed schedule $\mathcal{S}_T$.

Fusion

$$\begin{aligned}\mathcal{I} &= \{S1(i) : 1 \leqslant i < n;\\ &\qquad S2(i) : 1 \leqslant i < m\}\\ \mathcal{S} &= \{S1(i) \to (0,i);\\ &\qquad S2(i) \to (1,i)\}\\ \mathcal{T} &= \{(0,i) \to (i,0);\\ &\qquad (1,i) \to (i,1)\}\\ \mathcal{S}_T &= \{S1(i) \to (i,0);\\ &\qquad S2(i) \to (i,1)\}\end{aligned}$$

```
// Original loops
for (i=1; i<n; i+=1)
  S1(i);

for (i=1; i<m; i+=1)
  S2(i);

// Fused loops
for (i=1; i < min(n,m); i+=1) {
  S1(i);
  S2(i);
}
for (i=max(1,m); i<n; i+=1)
  S1(i);
for (i=max(1,n); i<m; i+=1)
  S2(i);
```

Fission

$$\begin{aligned}\mathcal{I} &= \{S1(i) : 1 \leqslant i < n;\\ &\qquad S2(i) : 1 \leqslant i < n\}\\ \mathcal{S} &= \{S1(i) \to (i,0);\\ &\qquad S2(i) \to (i,1)\}\\ \mathcal{T} &= \{(i,0) \to (0,i);\\ &\qquad (i,1) \to (1,i)\}\\ \mathcal{S}_T &= \{S1(i) \to (0,i);\\ &\qquad S2(i) \to (1,i)\}\end{aligned}$$

```
// Original loop
for (i=1; i<n; i+=1)
  S1(i);
  S2(i);

// Separated loops
for (i=1; i<n; i+=1)
  S1(i);

for (i=1; i<n; i+=1)
  S2(i);
```

Reversal

$$\mathcal{I} = \{S1(i) : 1 \leqslant i < n\}$$
$$\mathcal{S} = \{S1(i) \to (i)\}$$
$$\mathcal{T} = \{(i) \to (-i)\}$$
$$\mathcal{S}_T = \{S1(i) \to (-i)\}$$

```
// Original loop
for (i=1; i<n; i+=1)
  S1(i);

// Reversed loop
for (i=1-n; i<0; i+=1)
  S1(-i);
```

Interchange

$$\mathcal{I} = \{S1(i,j) : 1 \leqslant i < n; \wedge 1 \leqslant j < m\}$$
$$\mathcal{S} = \{S1(i,j) \to (i,j)\}$$
$$\mathcal{T} = \{(i,j) \to (j,i)\}$$
$$\mathcal{S}_T = \{S1(i,j) \to (i,j)\}$$

```
// Original loops
for (i=1; i<n; i+=1)
  for (j=1; j<m; j+=1)
    S1(i,j);

// Interchanged loops
for (i=1; i<m; i+=1)
  for (j=1; j<n; j+=1)
    S1(j,i);
```

Strip-Mining

$$\mathcal{I} = \{S1(i) : 1 \leqslant i < n\}$$
$$\mathcal{S} = \{S1(i) \to (i)\}$$
$$\mathcal{T} = \{(i) \to (4 * \lfloor i/4 \rfloor, i)\}$$
$$\mathcal{S}_T = \{S1(i) \to (4 * \lfloor i/4 \rfloor, i)\}$$

```
// Original loop
for (i=1; i<n; i+=1)
  S1(i);

// Strip mined loop
for (ti=0; ti<n; ti+=4)
  for (i = max(1,ti);
       i <= min(n-1,ti+3);
       i += 1)
    S1(i);
```

Skewing

$$\mathcal{I} = \{S1(i,j) : 1 \leqslant i < n \wedge 1 \leqslant i < m\}$$
$$\mathcal{S} = \{S1(i,j) \to (i,j)\}$$
$$\mathcal{T} = \{(i,j) \to (i, i+j)\}$$
$$\mathcal{S}_T = \{S1(i) \to (i, i+j)\}$$

```
// Original loops
for (i=1; i<n; i+=1)
  for (j=1; j<n; j+=1)
    S1(i,j);

// Skewed loops
for (i=1; i<n; i+=1)
  for (j=i+1; j<m+i; j+=1)
    S1(i, j-i);
```

Tiling

$$\mathcal{I} = \{S1(i,j) : 0 \leqslant i < 1024 \wedge 0 \leqslant i < 1024\}$$
$$\mathcal{S} = \{S1(i,j) \to (i,j)\}$$
$$\mathcal{T} = \{(i,j) \to (4 * \lfloor i/4 \rfloor, 4 * \lfloor j/4 \rfloor, i, j)\}$$
$$\mathcal{S}_T = \{S1(i) \to (4 * \lfloor i/4 \rfloor, 4 * \lfloor j/4 \rfloor, i, j)\}$$

```
// Original loops
for (i=0; i<1024; i+=1)
  for (j=0; j<1024; j+=1)
    S1(i,j);

// Tiled loops
for (ti=0; ti<1024; ti+=4)
  for (tj=0; tj<1024; tj+=4)
    for (i=tj; i<ti+4; i+=1)
      for (j=tj; j<tj+4; j+=1)
    S1(i, j-i);
```

Unroll and Jam

$$\mathcal{I} = \{S1(i,j) : 1 \leqslant i < 1024 \wedge 1 \leqslant i < 1024\}$$
$$\mathcal{S} = \{S1(i,j) \to (i,j)\}$$
$$\mathcal{T} = \{(i,j) \to (4 * \lfloor i/4 \rfloor, j, i)\}$$
$$\mathcal{S}_T = \{S1(i) \to (4 * \lfloor i/4 \rfloor, j, i)\}$$

```
// Original loops
for (i=0; i<1024; i+=1)
  for (j=0; j<1024; j+=1)
    S1(i,j);

// Unroll and jammed loops
for (ti=0; ti<1024; ti+=4)
  for (j=0; j<1024; j+=1)
    for (i=tj; i<ti+4; i+=1)
      S1(i, j-i);
```

We do not describe all transformations in detail, but invite the reader to compare the source code transformations with the transformation map $\mathcal{T}$ that describe them to get an idea of the concepts we use to model such transformations. While doing so a couple of interesting details may be spotted. In the fusion example we see that fusing two loops does not only work in the trivial case where the two loops have the same number of iterations, but also in more general cases where additional loops are needed to enumerate statement instances from one loop that may not have a matching instance in the other loop. Similar boundary condition handling can be seen in the strip-mining example where the relevant `min/max` expressions have been introduced. For tiling and unroll-and-jam we have chosen examples that do not require specific boundary condition handling. Another interesting observation is that strip-mining is modeled as a schedule only transformation. This was first presented by Kelly et al. [79], but in later literature strip-mining and consequently tiling and unroll-and-jam are often shown as transformations that require modifications of the iteration space [37, 27]. We prefer to model them as schedule only transformations as this has shown to simplify both the reasoning about such transformations as well as their implementation.

Another interesting group of schedule transformations are transformations where different schedules are assigned to different instances of the same statement. We call such schedules piecewise schedules. They result for example from index set splitting [63], but also from tiling schemes as presented in Chapter 4 or Chapter 5. To describe such schedules, we use the previously introduced integer maps with just slightly more complex Presburger formulas. Assuming we have a statement $\{S(i) \mid 0 \leqslant i < 2n\}$ where we want to apply a reverse schedule $S_1 = \{S(i) \to (0, n-i-1)\}$ to the statement instances $\{S(i) \mid 0 \leqslant i < n\}$ and an identity schedule $S_2 = S(i) \to (1, i)\}$ to the remaining instances $\{S(i) \mid n \leqslant i < 2n\}$, we can form a combined schedule $\{S(i) \to (0, n-i-1) \mid 0 \leqslant i < n\} \cup \{S(i) \to (1, i) \mid n \leqslant i < 2n\})$. By generating code for this schedule, we obtain the following result:

```
for (i = 0; i < n; i += 1)
  S1(n - i - 1);
for (i = n; i < 2 * n; i += 1)
  S1(c1);
```

In previous works index set splitting was often modeled by introducing (virtual) statement copies. We advocate for the usage of partial schedules as it eliminates the need for statement copies as another concept.

There is also a set of classical loop transformations that change the loop structure, but not the order in which statement instances are executed. Such transformations are loop unrolling, loop peeling or loop unswitching. All of these do not require any schedule transformations, but are different ways of specializing code to reduce control overhead. In Chapter 10 we discuss our work on translating a schedule back to imperative code (AST generation) and present different AST generation strategies that correspond to a set of classical execution order preserving loop transformations.

Part III

TILINGS AND OPTIMIZATIONS FOR STENCILS

3 STENCIL COMPUTATIONS

The first part of this thesis is dedicated to the generation of optimized GPU code for so-called stencil computations. We start in Chapter 3 with an overview, which begins with an introduction to stencil computations in Section 3.1, continues with a discussion of why tiling of stencil computations is beneficial in Section 3.2 as well as as a discussion of related work in Section 3.3, and concludes in Section 3.4 with a brief section about our work on stencil computations.

In the following chapters we present two approaches to generate optimized GPU code for stencil computations. We start with split tiling in Chapter 4 and subsequently present hybrid hexagonal/parallelogram tiling in Chapter 5. In Chapter 6 we analyse the properties of our new hexagonal tiling strategy and compare it with work on diamond tiling. We finish in Chapter 7 with a set of experiments to evaluate our work.

3.1 WHAT ARE STENCIL COMPUTATIONS?

Stencil computations are computations that iteratively update data in a multi-dimensional grid by recomputing each data point from a set of neighboring points. Computations of this form are common in many scientific applications e.g., computational electromagnetism [120], weather prediction [71], cellular automata [90] or solvers for differential equations [117].

```
L1: for (t = 0; t < T; t++)
L2:   for (i = 1; i < N-1; i++)
L3:     for (j = 1; j < N-1; j++) {

S1:       A[(t+1)%2][i][j] = 0.2 * (  A[t%2][i  ][j  ]
                                    + A[t%2][i  ][j-1]
                                    + A[t%2][i  ][j+1]
                                    + A[t%2][i-1][j  ]
                                    + A[t%2][i+1][j  ]);
        }
```

Listing 1: Implementation of a 5-point, 2D-space, jacobi-style heat stencil

Listing 1 shows a possible implementation of a 5-point heat stencil on a two dimensional data-grid. The outermost loop L1 is the time loop which enumerates the different time steps, whereas the loops L2 and L3 scan at each time step the elements of the data-grid. Each

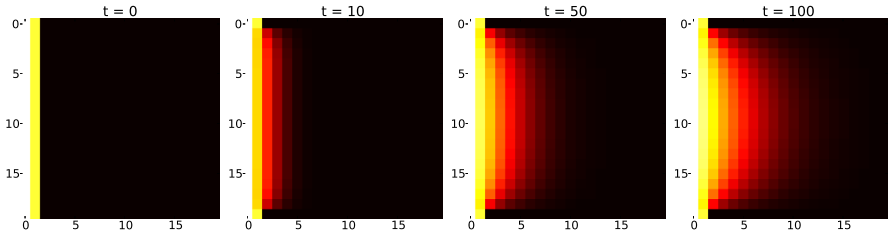


Figure 6: Heat distribution after running the stencil in Listing 1

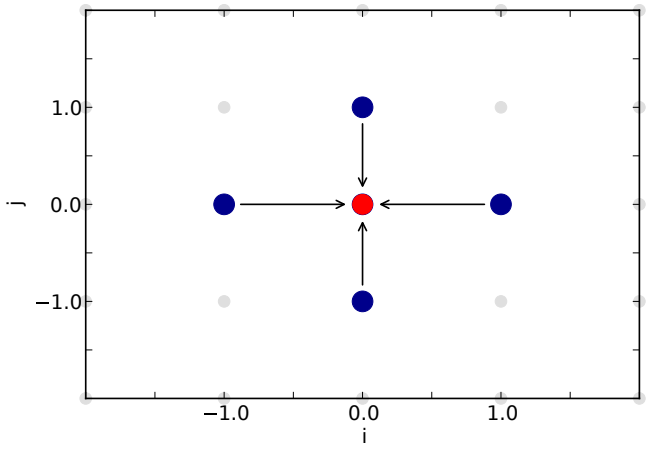


Figure 7: Data-flow of the stencil in Listing 1

data point is then recomputed by statement S1 using data from the point itself as well as its neighbors in the north, east, south and west. The pattern in which data is retrieved from the neighbors is called the stencil. The stencil of our example is shown in Figure 7. To get a better idea of the computation performed we also illustrate the result of the heat computation at different points in time (Figure 6).

An interesting observation to make is that even though the data space itself is two dimensional the array on which the computation is performed has three dimensions. The additional dimension is used as a temporary array that ensures that newly computed data values do not overwrite values still needed to compute other data points at the same time step.[1] There are also stencil computations such as Gauss-Seidel which do not need a temporary array, but the cost of the reduced data footprint is a loss of parallelism. For so-called Jacobi style stencils, the use of a temporary array enables all space loops to be executed fully in parallel.

1 Instead of using an additional dimension and modulos, existing implementations often use a second array and switch the pointers to the arrays to alternate. We have chosen our implementation over pointer switching, as pointer switching complicates further analysis and optimizations. Another option taken e.g. in Polybench [101] is to copy after each time step the full temporary array back to the original array. We do not use this approach as it introduces unnecessary copy overhead.

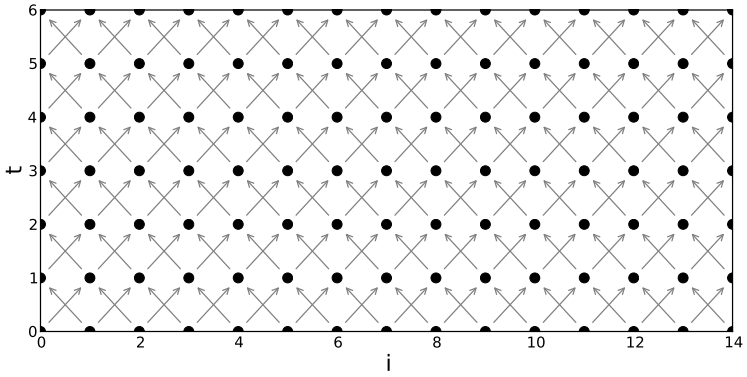


Figure 8: Iteration space of 1D-space stencil

To understand the available parallelism in Jacobi style stencils, we take a look at an even simpler 1D stencil:

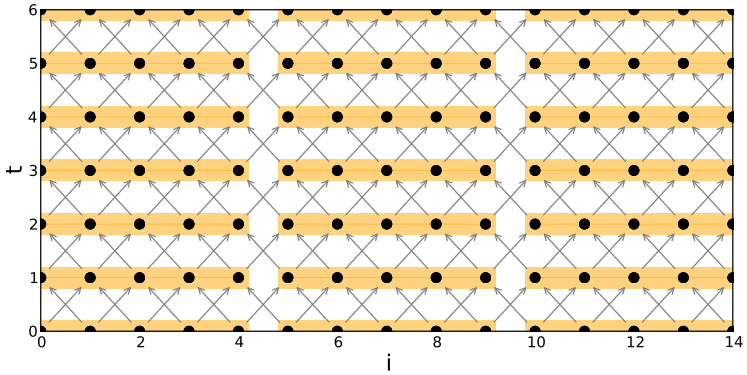
```
for (t = 0; t < T; t++)
  for (i = 1; i < N-1; i++)
    A[(t+1)%2][i] = A[t%2][i-1] + A[t%2][i+1];
```

When analysing its iteration space, illustrated in Figure 8, we observe that there are no data-dependences (arrows) between the individual computations (points) executed within a single time step t, but all dependences cross at least one time step. This means that within one time step the individual computations can be executed fully in parallel – a large amount of fine grained parallelism is available.

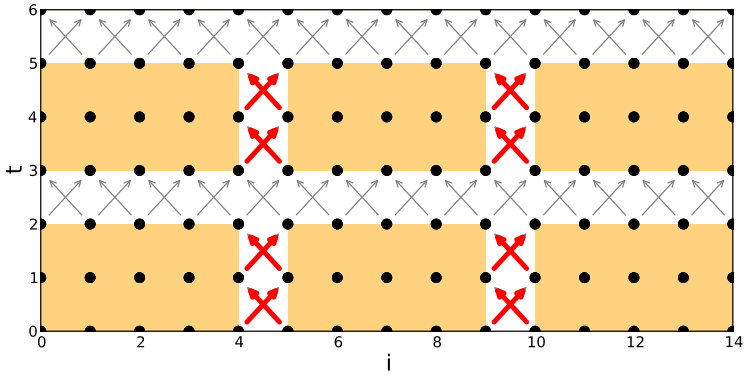
3.2 TILING OF STENCIL COMPUTATIONS

Even though stencil computations provide a large amount of fine-grained parallelism, executing them efficiently is non-trivial. Most stencils perform by default only a small amount of work compared to the data processed, which means the data transfer commonly limits the achievable performance. To execute such stencils efficiently, it is important to increase the computation to data transfer ratio. An effective way to do so is to form larger blocks of work, so called tiles. For stencil computations, forming tiles is almost generally beneficial, but individual tiling strategies impact the efficiency of the computation and the amount of parallelism exposed differently.

The simplest tile shapes we can choose are rectangular tile shapes. Figure 9 illustrates two kinds of rectangular tile shapes. Figure 9a shows rectangular tiles that only contain a single time step, but which combine computations on different elements in the data space. Such

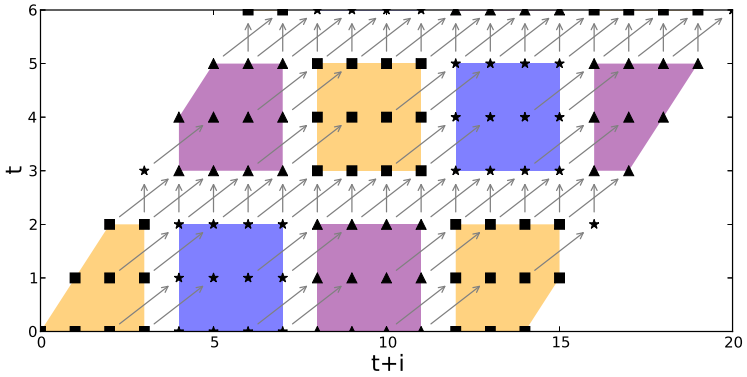


(a) Space tiling

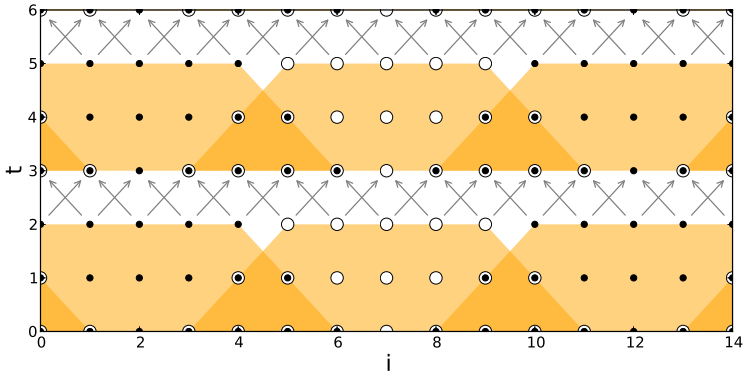


(b) Time tiling

Figure 9: Rectangular tiling of 1D-space stencil



(a) skewing + rectangular tiling



(b) overlapped tiling

Figure 10: Iteration space of 1D-space stencil

a tiling is called space tiling. Space tiling can be used to coarsen the computation and to enable a certain amount of reuse due to neighboring computations possibly sharing certain input values. However, the maximal amount of reuse space tiling allows is limited. Higher levels of reuse can be obtained when using time tiling, a tiling that combines computations of different time steps in individual tiles. Figure 9b shows a rectangular tiling that crosses several time steps. We notice immediately that such a tiling introduces circular dependences between neighboring tiles. This does not only mean that we lose all parallelism, but, even worse, there is not any valid execution order of these tiles.

One way to construct a valid time tiling is to skew the iteration space (in this case $\{(t,i) \rightarrow (t,i+t)\}$) before applying rectangular tiling. In Figure 10a we see that all circular dependences disappear. In fact, it is even possible to schedule neighboring blocks of the same color as a wavefront where all blocks within one wavefront can be executed in parallel and only blocks that belong to different wavefronts need to be scheduled sequentially. The combination of skewing and rectangular tiling is a good starting point, but not always optimal. One drawback is the unbalanced coarse-grained parallelism

which causes reduced parallelism at the beginning and the end of the computation. In our illustration this is clearly visible, as the first two wavefronts (orange squares and blue stars) only contain a single block, then two wavefronts (purple triangles, orange squares) contain two blocks and we finish with two wavefronts (blue stars, purple triangles) each consisting of a single block. For this very small iteration space, we are most of the time in the phase of reduced parallelism. One may think for realistic problem sizes this may not be the case, but for higher dimensional stencils and cache restricted environments this imbalance can be problematic.

We say that tiling schemes ensure concurrent start, if they ensure that the wavefront of tiles that are executed in parallel is parallel to the data space hyperplane. Tiling schemes with concurrent start ensure balanced coarse-grained parallelism. Overlapped tiling [84, 70] is one tiling scheme that ensures concurrent start. It can be seen as a form of rectangular tiling, but with additional computations added to resolve the circular dependences. As can be seen in Figure 10b, overlapped tiling has the same amount of coarse-grain parallelism throughout the computation (in our illustration 3 parallel tiles). However, the cost of this balanced parallelism is both redundant computations and the need for additional cache memory to store the results of the redundant computations. For environments where computations are cheap, but cache memory is expensive (e.g., GPUs), the latter might be even more problematic.

3.3 RELATED WORK

There is a lot of interesting work on optimizing stencil computations. To give the reader the necessary background on this related work, we will discuss it before presenting our own work. As we focus in our work on the development of tiling schemes that enable efficient GPU code generation, we will begin the discussion in this section with the most relevant related work and then slowly widen the focus of our discussion. At the end of this section we will provide related work on the theoretical analysis of tiling schemes, which is relevant to our work in Chapter 6.

A very interesting approach for generating optimized stencil codes for GPUs is presented by Holewinski et al. [70]. In their work they discuss a DSL compiler that automatically generates CUDA code for stencil computations. Using overlapped tiling, as discussed in Section 3.2, they ensure data reuse along the time dimension. Overlapped tiling itself has been introduced earlier by Krishnamoorthy et al. [84] who evaluate its benefits on a CPU cluster system and a hierarchical form of overlapped tiling is discussed by Zhou et al. [141] in the context of OpenCL code generation for CPUs. None of these earlier approaches discusses the automatic generation of CUDA code. Another time tiling

scheme, called 3.5D tiling, was introduced by Nguyen et al. [96] as data blocking scheme for both CPU and GPU devices. 3.5D tiling also uses overlapping regions to ensure the legality of time tiling. To our understanding their work does not present 3.5D tiling as part of an end-to-end compiler workflow. Meng and Skadron [94] also look into a form of overlapped tiling, even though they do not explicitly mention this very term. Instead they use the term ghost zones to describe the areas of redundant computations. Such ghost zones are also their main focus of research as they aim to model the performance of CUDA code in function of ghost zone size. They do not discuss the automatic generation of GPU code. Finally, the last piece of work we want to mention in the context of time tiling on GPUs is the work of Di and Xue [51] who model and automatically select tile sizes of GPU kernels. In their work they use a combination of skewing and rectangular tiling similar to the approach described at the beginning of Section 3.2.

Time tiling as a way to improve data reuse of stencil computations has also been used in the context of CPUs. Tang et al. [121] propose the Pochoir stencil compiler which uses a DSL embedded in C++ to produce high-performance code for stencil computations using time tiling obtained with cache-oblivious parallelograms. In their work they target C++ and Cilk to obtain thread-level parallelism on CPUs, but do not discuss GPU code generation or related optimizations. Strzodka et al. [118, 119] use time skewing with cache-size oblivious parallelograms to obtain fast CPU code. One major selling point of their tiling scheme is its cache size independence. Cache size independence is an interesting property, but the additional synchronization between tiles, as required by their tiling scheme, may hinder its exploitation on GPUs where inter thread block synchronization primitives are commonly not available. There has also been work from Henretty et al. who propose a DSL-based approach [68] for generating high-performance code for multi-core vector-SIMD architectures. In their work hybrid prismatic tile shapes ensure time tiling. Another interesting approach currently only evaluated on CPUs is diamond tiling as presented by Bandishti et al. [21]. Diamond tiling is a time tiling technique that uses diamond shaped tiles to avoid redundant computations while ensuring balanced coarse grained parallelism. However, to obtain this balanced coarse grained parallelism it imposes certain restrictions on the choice of tile sizes. An interesting point of diamond tiling is its integration into a general purpose polyhedral optimizer called Pluto [35]. We discuss this technique in detail in Chapter 6.

There is also interesting work that focuses less on the generation of optimal tile shapes, but on other aspects relevant for generating good stencil code. With the PATUS stencil compiler Christen et al. [41] propose a system which, given the description of a parameterizable code generation and parallelization strategy as well as the specification of

a specific stencil, generates efficient CPU or GPU code using autotuning. Even though PATUS was conceptually designed to support time tiling, the original implementation missed some essential parts to do so [42, 6.2] and to our understanding these parts have not yet been added. Han et al. [66] are developing with PADS pattern-based optimization of stencil codes on CPUs and GPUs using a proposed extension to OpenMP. Similar to PATUS, code generation strategies can be provided by the user. To our understanding also no time tiling is used. Datta et al. [48, 49] develop an optimization and auto-tuning framework for stencil computations, targeting multi-core systems, NVIDIA GPUs, and Cell SPUs. Similarly to the previous systems, no time-tiling is considered. For completeness, we also want to point out the existence of hand-tuning efforts. Micikevicius et al. [95] discussed the manual optimization of a 3-D finite difference computation stencil. He uses a space tiling scheme to obtain efficient CUDA code.

Besides domain specific approaches, there also exists a set of general purpose optimizers that fully automatically generate parallel GPU code for a larger range of programs. Such optimizers can often also generate code for stencil computations. Baskaran et al. [23] present with C-to-CUDA the first end-to-end automatic source-to-source compiler that translates C code to CUDA. It uses Pluto's scheduling algorithm and supports the use of software managed caches. Using similar techniques, Reservoir Labs' R-Stream [87, 125] is another polyhedral compiler that targets GPUs. PPCG [135], an evolution of C-to-CUDA, generates parallel CPU and GPU code using rectangualar tiling. It relies on affine transformations to extract parallelism and improve locality, using a variant of the Pluto algorithm. Par4All [18] is an open source parallelizing compiler developed by Silkan targeting multiple architectures. The compiler is not based on the polyhedral model, but uses abstract interpretation for array regions, performing powerful inter-procedural analysis on the input code.

We also want to briefly list a set of publications that discuss the theoretical analysis of tile shapes. As previously mentioned, such work is relevant for the last chapter of this part of the thesis. Already very early there was work by Schreiber and Dongarra [113] on how to derive optimal tile sizes for loop blocking. For rectangular tile shapes they compute tile sizes that optimize the ratio between the performed computations and the required data accesses using a simple machine-independent cost model. Another interesting piece of related work is the positivity based tile size selection framework of Renganarayana and Rajopadhye [109]. In their work they show that many cost models and heuristics in compilers can be described by posynominals which again can be used to define functions for which the optimal value can be computed efficiently. This work is of high interest as it applies across a large range of cost models. However, it does not provide insights on how to derive new cost models as needed for

example to derive conclusions about new and possibly complex tile shapes. Orozco et al. [97] use the dependency graph to find the optimal tile shape for stencil computations, and conclude that the optimal tile shape in terms of compute-to-communication ratio is a diamond. Even though this finding is interesting by itself, it does not provide insights about the optimal tile shape when considering communication reducing techniques (R_{REDUCED} in Chapter 6) or when looking at the ratio of computation to synchronization.

3.4 OUR WORK ON STENCIL COMPUTATIONS

Even though there has already been significant work on stencil computations there has been less work on generating GPU code fully automatically, and even less work when considering techniques that exploit time tiling.

In the following chapters we present two stencil specific tiling strategies which both exploit time tiling and provide concurrent start, without the need for redundant computations or a reduced freedom in the selection of tile-sizes. Split-tiling as presented in Chapter 4 is a technique that provides coarse-grained parallelism on all dimensions, whereas our hybrid hexagonal/parallelogram tiling scheme (Chapter 5) sacrifices coarse-grained parallelism on some space dimensions to even better address GPU specific code generation concerns.

All our tiling techniques are described as polyhedral schedule transformations that are applied by a generic GPU code generator (PPCG). The integration into a general purpose optimizer is challenging in the first place, but has several benefits. First, we can reuse existing code generation infrastructure and non-stencil specific optimizations. Second, the user can rely on the same GPU code generator for different types of codes and just benefit from higher quality code on stencil computations. Finally, the abstract description of the enabling tiling schemes makes the later integration into other DSL compilers or even compilers for languages as C/C++, Fortran or Julia [31] possible.

We complete our work on stencils with an analysis of diamond tiling, a tiling scheme that can be seen as a special case of hexagonal tiling. We discuss the constraints diamond tiling imposes on tile sizes and wavefront coefficients and we formulate additional constraints that ensure the uniform placement of integer points across tiles. Uniform integer point placement can help to reduce control flow statements and is by construction ensured in our first formulation of hexagonal tiling. We then provide a second formulation of hexagonal tiling, this time in the context of the Pluto framework, which can describe both diamond and hexagonally tiled code. Using his formulation we analyze the ratio of communication to computation and communication to synchronization obtained with both diagonal and hexagonal tiling.

4

SPLIT TILING

One tiling scheme that allows time tiling of stencil computations without introducing unbalanced inter-tile parallelism or redundant computations is split tiling [84]. Split tiling uses index set splitting [63] to partition the iteration space into phases such that within a single phase all tiles can be executed in parallel. Due to carefully choosing the tile shapes of each phase and the use of possibly different shapes in different phases, split tiling does not require redundant computations.

In this chapter we propose a generic algorithm to calculate index-set splitting for an arbitrary number of dimensions and without the need to construct any larger integer linear program. Using this algorithm we enhance PPCG, a general purpose polyhedral GPGPU code generator, such that it can use split tiling to generate optimized GPU code for stencil computations.

4.1 OVERVIEW

The general idea of split tiling is to divide the iteration space in a sequence of bands that are orthogonal to the sequential dimension and which themselves need to be executed in sequence. Within each band the iterations are split into phases that partition the band such that all tiles within a single phase are independent and can consequently be executed in parallel. Figure 11 illustrates split tiling for the example shown in Figure 8. We can see horizontal bands of height three which are orthogonal to the time dimension and for which a sequential execution order is enforced by the data dependences. Within each band we can see two kinds of tiles, orange upright trapezoids as well as blue inverse triangles. All tiles of the same kind and within the same band can be executed in parallel, but dependences enforce an ordering of the individual phases. In this case, the trapezoidal tiles need to be executed before the triangular tiles. Looking more closely we see that due to its carefully chosen tile shape, a certain trapezoidal tile only depends on tiles from an earlier band, but never on tiles in the same band. Similarly the triangular tiles in a single band never depend on each other, but only on trapezoid tiles or tiles from an earlier band. The overall result is a time tiling that does not require any redundant computation, but still provides balanced coarse-grained parallelism.

To obtain a schedule that implements split tiling, the iteration space is divided into subspaces and for each subspace a different schedule

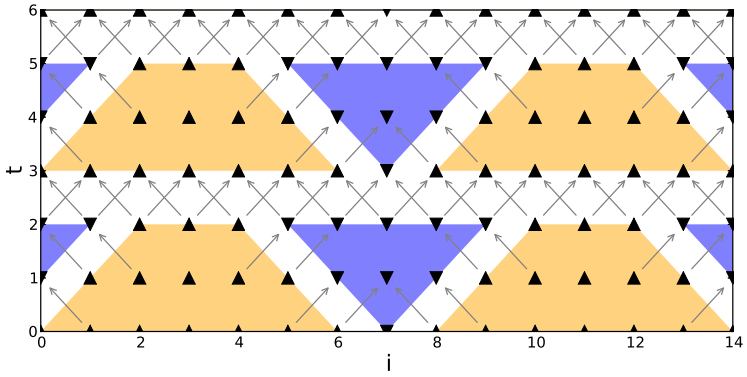


Figure 11: Split tiling for the simple example (tile size 8×3).

is computed. For the example in Figure 11 we define the subspaces T_O, the orange tile space, and T_B, the blue tile space. The schedule for T_O is S_O and the schedule for T_B is S_B. The overall schedule S is a piecewise integer map where S_O is used for all elements in T_O and S_B is used for all elements in T_B. For each subspace we can now define a schedule that executes the convex subsets (our tiles) in parallel. S_O and S_B define such schedules. For our example we define T_O, T_B, S_B and S_B as follows (tile size 64×32):

$$
\begin{aligned}
T_O = \{&A(t,i) \mid (\exists i', t' : t' = t \bmod 32 \wedge i' = i \bmod 64 \\
&\wedge i' - t' \leqslant 0 \wedge i' + t' \leqslant 64 - 2)\} \\
T_B = \{&A(t,i) \mid (\exists i', t' : t' = t \bmod 32 \wedge i' = i \bmod 64 \\
&\wedge i' - t' < 64 \wedge i' + t' > 64 - 2)\} \\
S_O = \{&A(t,i) \to (t', 0, i', t, i) \mid t' = \lfloor t/32 \rfloor * 32 \\
&\wedge i' = \lfloor i/64 \rfloor * 64\} \\
S_B = \{&A(t,i) \to (t', 1, i', t, i) \mid t' = \lfloor t/32 \rfloor * 32 \\
&\wedge i' = \lfloor (i-31)/64 \rfloor * 64\}
\end{aligned}
$$

To generate split tiled CPU code it is sufficient to provide this schedule to a polyhedral code generator such as CLooG [26] or the new isl AST generator (Chapter 10). The resulting code is shown in Listing 2. GPU code generation is more involved, but the high-level picture is rather simple. We use normal CPU host code to enumerate the different bands. We then execute each phase of a band individually on the GPU by mapping each tile to a thread block, creating explicit sequential GPU code to enumerate the individual time steps of a tile and we map statements instances that work on different elements of the data space to independent GPU threads. On top of this we add synchronization that ensures that all elements in a tile that are part of one time step are computed before computing the elements of the subsequent time steps. Further optimizations are applied to take advantage

of shared memory and the on-GPU software managed caches, as well as to introduce instruction level parallelism.

```
for (c1=0;c1<=M-1;c1+=32) {
  lb = 0;
  ub = min(N-2,c1+N-3);
#pragma omp parallel for shared(c1,ub,lb) private(c3,t,i)
  for (c3 = lb; c3 <= ub; c3+=64)
    for (t = max(1,c1);
         t <= min(min(M-1,c1+31),c1-c3+N-2); t++)
      for (i = max(1,-c1+c3+t);
           i <= min(N-2,c1+c3-t+62); i++)
        A[t][i] = A[t-1][i-1] + A[t-1][i+1];

  lb = max(-64,-64*floord(-c1+M-3,64)-64);
  ub = min(N-34,-c1+M+N-66);
#pragma omp parallel for shared(c1,ub,lb) private(c3,t,i)
  for (c3 = lb; c3 <= ub; c3+=64)
    for (t = max(max(max(1,c1),c1-c3-62),c1+c3-N+65);
         t <= min(M-1,c1+31); t++)
      for (i = max(1,c1+c3-i+63);
           i <= min(N-2,-c1+c3+t+63); j++)
        A[t][i] = A[t-1][i-1] + A[t-1][i+1];
}
```

Listing 2: Split tiled code

4.2 PREPROCESSING

As a first step, we extract a polyhedral description from our input C program using `pet` (Chapter 9), compute dependences using `isl` [128] and transform the polyhedral description into some canonical form that later simplifies the construction of the schedule. The C input can contain modulos, non-unit stride loops and piecewise affine expressions, the latter are useful for example to model boundary conditions. There is also no limit on the number of arrays in the kernel. Focusing on the algorithmic domain of stencil computations, we assume that the input program consists of an outer loop containing $k \geqslant 1$ perfect nests of loops such that none of the loops in these nests carry any dependences. That is, all dependences are either carried by the outer loop or connect instances from different loop nests. If these conditions are met, then we construct a schedule of the form $\{L_i(t, s_0, \ldots, s_n) \to (k \cdot t + i, s_0, \ldots, s_n)\}$, where i satisfying $0 \leqslant i < k$ reflects the order in which the loop nests appear inside the outer loop. If the loop nests have different nesting depths, then they are currently manually aligned. In the constructed schedule, all dependences are

carried by the outer dimension $k \cdot t + i$, meaning that the remaining dimensions s_i are fully parallel.

More generally, we could use a general purpose optimizer such as Pluto [35] to construct such an initial schedule (i.e., one with a single outer sequential dimension followed by only parallel dimension). This would allow us to consider more general inputs, but is left for future work.

4.3 THE SPLIT TILING SCHEDULE

We present our new algorithm in steps, starting with the main idea applied to a single statement stencil, then generalizing it to multi-statement kernels, and refining the method with necessary optimizations.

4.3.1 *Core algorithm*

Given an initial schedule as described in Section 4.2, we partition the iteration space of the schedule domain by placing equally distanced hyperplanes orthogonal to the axis of the time dimension. The different partitions form bands of fixed height (the height of the tiles). As the time dimension increases from band to band and as all dependences are carried by the time dimension, the bands can and must be executed sequentially. To obtain parallelism, we split the iterations within a single band into tiles of different colors, such that dependences may enforce an execution order between the different colors, but that within a single color all tiles can be executed in parallel.

To partition the band into different colors, we derive a tile shape for each color such that the full band can be covered with these shapes. The tile shape of the first color C_0 is constructed by choosing an arbitrary point X. X will be the apex of a pyramid that contains all iterations within the band that are needed to satisfy the (transitive) dependences of X. To construct this pyramid, we calculate the dependence distance vectors of the program and attach all of them to X. Together they form the upper part of a pyramid. We now extend the dependence vectors backward until their length along the time dimension matches the tile height we are aiming for. The convex hull of the extended dependence vectors forms a pyramid. This pyramid is the minimal set of points that we consider as the shape of the first color. In some cases it is preferable to have a shape that is wider along certain space dimensions. We can form such wider shapes by "stretching" the initial pyramid along these space dimensions. Stretching along a dimension means to position two copies of the original shape, such that the positions of the copies only differ in the dimension along which we stretch them. The stretched shape is now the convex hull of the two shapes.

In addition to the first color, we derive one color for each space dimension in the input. The shape of a color C_x (where x corresponds to some space dimension) is derived by stretching the pyramid of C_0 along the x-dimension and by subsequently subtracting the shapes of all previously calculated colors.

In the case of more than one space dimension, additional colors are needed. Besides the initial color C_0 and the colors for individual dimensions, we introduce a color for each combination of dimensions. This means, for a 3D input, the colors C_{xy}, C_{xz}, C_{yz} as well as C_{xyz} are introduced. Their tile shapes are derived by stretching the initial pyramid along the set of dimensions they are named after. This can be compared to calculating the different faces of a cube, where the pyramid itself forms the shape of a vertex, the pyramids stretched along a single dimension form the differently oriented edges, the pyramids stretched along two dimensions form the facets and the pyramid stretched along all three dimensions forms the cube itself. Stretching the pyramid along more than one dimension (e.g., along x-y-z) is done recursively. We select one dimension (e.g., y) and calculate the union of the tile shapes that correspond to the colors of the remaining dimensions (here C_{xy}, C_x, C_z, C_0). This union is then replicated along the selected dimension, the convex hull of the entire construct is calculated, and finally the previous colors as well as their replicated copies are subtracted.

The split tiling schedule is constructed by tiling the original iteration space with the previously calculated tile shapes, such that the sequential execution of the different bands as well as of the different colors is ensured. Tiles of the same color and within the same band are mapped to parallel dimensions. The iterations within the tiles are executed according to the original schedule. As the index set splitting is calculated without considering the bounds of the iteration space, there is no constraint on the shape of the iteration space. Only as the very last step do we constrain the schedule to the actual iteration space.

The left part of Figure 12 shows a split tiling of the Jacobi 2D kernel. The pyramid that forms the first color was placed in the center of the iteration space. At time step one (the upper left illustration) the number of elements in the first color is still large. When going down to time steps two and three we are moving up the pyramid such that the number of elements executed becomes smaller. At time step three, color one consists only of a single point, the summit of the pyramid. The shape of color two forms the connection between two vertical neighbors of color one. The shape is non-convex and resembles a simple butterfly. Color three now forms the horizontal connection between two neighboring shapes of color one. Color four, the last color constructed, fills the space enclosed by the previously calculated colors.

4.3.2 *Tile shape simplification*

The previously introduced split tiling algorithm starts from a single pyramid that exactly covers the dependence vectors. Depending on the dependence vectors, such a minimal pyramid may not always be parallel to the axes of the iteration space. In case it is not, such as in Figure 12, subsequent tile shapes may have a non-convex form. Such non-convex tile shapes are undesirable, not only because they increase the required amount of communication between the different tiles, but they also introduce more complex control flow structures. To avoid such complex control flow structures, we normally widen the original pyramid to create a rectangular base. This can avoid the construction of non-convex tiles. Figure 12 illustrates the tile shapes resulting from the widening of the original pyramid to a rectangular base. As can been seen, this leads to greatly simplified (and convex) shapes.

4.3.3 *Multi-statement loop nests*

Up to this point, our split tiling algorithm is only defined for kernels typical for single-statement stencils. In this section, we extend it to stencil computations that apply more than one statement in each iteration of the time loop. We do this by matching for a specific iteration space pattern that we have observed to commonly appear in multi-statement stencils (e.g. the Polybench [101] jacobi stencils). We point out to the reader that our approach is an optimistic heuristic, which could possibly be generalized to work on a wider range of patterns. An interesting approach in this area is also proposed by [68] who models multi-statement stencils in his DSL compiler.

To detect multi statement stencils we look for a SCoP that consists of an outer sequential dimension (time), an additional sequential dimension with a known non-parametric number of iterations (the lexicographic position of the different statements), as well as a set of fully parallel dimensions (space dimensions). Figure 13 shows a simple two-statement kernel that matches this pattern. Its iteration domain is $\{S(t,i) \mid 0 \leqslant t < T \wedge 0 \leqslant i < N; P(t,i) : 0 \leqslant t < T \wedge 0 \leqslant i < N\}$ and its execution order is defined by the following schedule $\{S(t,i) \to (t,0,i); P(t,i) \to (t,1,i)\}$. The following dependences exist: $\{S(t,i-2) \to P(t,i); P(t-1,i-1) \to S(t,i); S(t-1,i) \to S(t,i); P(t-1,i) \to S(t-1,i)\}$. Mapped into the scheduling space, this yields $\{(t,0,i-2) \to (t,1,i); (t-1,1,i-1) \to (t,0,i); (t-1,0,i) \to (t,0,i); (t-1,1,i) \to (t,1,i)\}$. By analyzing the dependences, we see that the two outermost dimensions both carry loop dependences. This means our split tiling algorithm is not directly applicable.

By applying a simple pre-transformation we can canonicalize the code such that it is again possible to use the previously presented

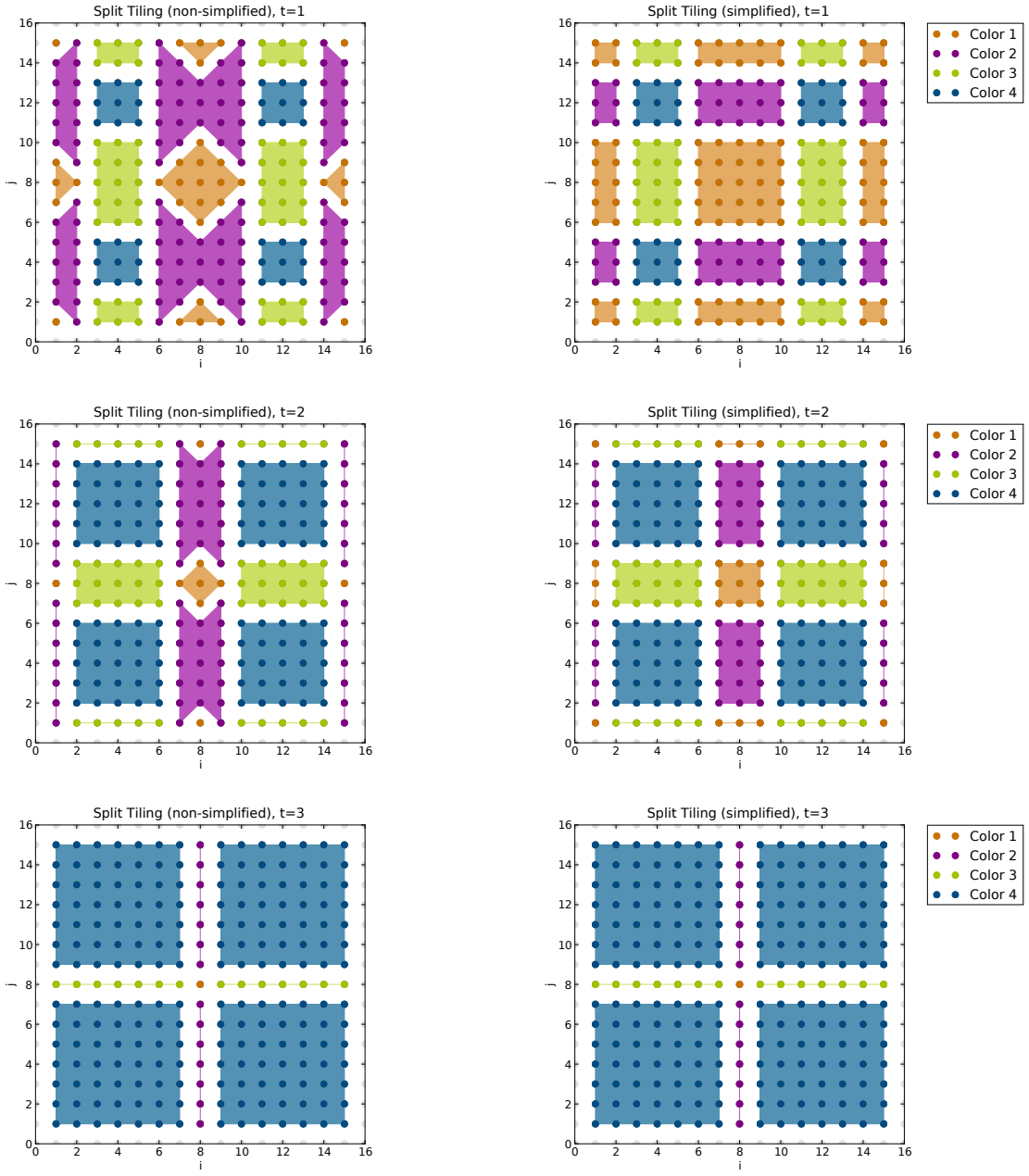


Figure 12: Split tiled jacobi-2d kernel

split tiling algorithm. We detect that only the outermost time dimension can have parametric size related to the number of time steps executed. The size of the second sequential dimension is independent of the number of executed time steps. As the second dimension represents the lexical order of the statements in the source code, its size is bound by the number of statements in the source code. As the integer value of this bound is available at compile time, we can fold the two time dimensions into a single one. For a two statement kernel this transformation can be described by the following mapping $\{(t, 1, i) \to (2t + 1, i) \mid 0 \leqslant l \leqslant 1\}$. Applying this mapping on

the original schedule gives us $\{S(t,i) \to (2t,i); P(t,i) \to (2t+1,i)\}$ as well as the dependences $\{(2t,i-2) \to (2t+1,i); (2t-1,i-1) \to (2t,i); (t-2,i) \to (t,i)\}$. After this transformation, all dependences are again carried by the outermost dimensions and the inner parallel dimensions remain unchanged. Now, the previously presented split tiling algorithm can be applied.

```
for (t = 0; t < T; t++) {
  for (i = 0; i < N; i++)
S:        A[1][i] += A[0][i+1]

  for (i = 0; i < N; i++)
P:        A[0][i] += A[1][i+2]
}
```

Figure 13: Two statement kernel

4.4 CUDA CODE GENERATION

To generate CUDA code, we extended the polyhedral GPU code generator PPCG [135]. PPCG is a state-of-the-art GPU code generator that translates static control programs in C to CUDA enabled programs. For certain classes of computations (e.g., linear algebra kernels), this produces very efficient code that reaches the performance of highly-tuned libraries. For stencil computations, PPCG performs a basic mapping where for each time step, a new kernel instance is spawned and each kernel applies a single stencil to just a couple of data points. This mapping exposes a high level of parallelism, but at each time step all data is read from and written to global memory. This means the global memory bandwidth becomes the limiting factor of the stencil computation.

By adding support for split tiling, we enabled PPCG to produce time-tiled CUDA code for stencil like computations. Such code executes several iterations of the time loop within each kernel and keeps intermediate results in shared memory. This significantly lowers the pressure on the global memory bandwidth and consequently allows a higher computational throughput.

The split tiling support for PPCG was developed by enhancing and parameterizing the polyhedral optimization infrastructure that was already available in PPCG. We specifically avoided the development of a new domain-specific code generator, but aimed instead at enhancing an existing GPU optimization framework. From a user's point of view, this provides a smoother experience as the same framework can be used for a wide range of kernels. The only difference is that it is now possible to obtain improved code for stencil computations. From the developer's point of view, the use of a uniform optimization

framework has several benefits. Developing on top of an existing infrastructure speeds up the development of the CUDA code generator. It also enabled us to develop generic features and optimizations that can be beneficial for PPCG itself, but that show immediate benefits for split tiling if parameterized accordingly. The uniform framework makes it again very easy to specify and communicate the necessary parameters to the relevant transformations.

When generating split-tiled CUDA code we start from the C code of the program. This code is read by the polyhedral extraction tool *pet* (Chapter 9) which is available from within PPCG. Based on the extracted polyhedral description, we check if the program is suitable for split tiling. If this is the case, we derive a split-tiled schedule according to the generic algorithm described above. This new schedule is now provided to the generic PPCG transformation infrastructure where it replaces the PPCG internal schedule optimizer as well as the PPCG internal tiling. Instead, we parameterize PPCG with information about the schedule we provide. This information includes the number of dimensions of the entire schedule, the number of outer loops that should be executed on the host side, the first parallel loop that should be mapped to the GPU, the dimensionality of the tiles, the number of parallel dimensions in the tiles as well as information about the number of dimensions that should be considered when keeping values in shared memory.

PPCG uses this information to map the split tiles to the GPU. The mapping itself is rather straightforward. The tile loop of the time dimension is generally kept in the host code where it loops over a sequence of kernel calls. Each kernel call executes a set of thread blocks which in turn execute the parallel tiles as available at a certain time point using a one-to-one mapping from tiles to thread blocks. Within a tile, the parallel loops that enumerate the space dimensions are mapped to individual threads in a way that ensures coalesced memory accesses. The non-parallel loop for the time dimension is executed sequentially in each kernel. `__synchthreads` calls are introduced to ensure that each time step is finished before the next one is started.

4.4.1 *Shared memory usage*

The most important optimization for split tiling is the use of shared memory. The standard code that PPCG generates for stencils only uses shared memory to take advantage of spacial reuse within a single calculation. Such spacial reuse rarely happens for stencils and the additional synchronization overhead often outweighs the benefits of shared memory usage. However, with split tiling, we can now take advantage of reuse along the time dimension. This means all calculations within a single tile can be performed in shared memory. The

only accesses to global memory are transfers from global memory to shared memory before executing the code of a tile and transfers from shared memory back to global memory after the execution of a tile has finished. As each combination of kernel and work group only execute one tile at a time, there is no reuse of data between different tiles. The actual generation of the code that transfers data to and from shared memory is not specific to split tiling, but we just reuse the shared memory code generation that is already part of PPCG [135, Section 7 - Memory Allocation]. To do so we only need to provide PPCG with information about where we want to exploit reuse, in our case within individual tiles. In PPCG terms this information can be transmitted by giving the schedule dimensions in which shared memory should be exploited.

4.4.2 *Instruction level parallelism*

On CUDA architectures several kinds of parallelism are available. Parallelism due to the execution of parallel threads is the most obvious one. However, even when generating code without split tiling, PPCG maps by default several data points to a single thread. Mapping several data points to a single thread ensures that there is a certain number of instructions between two subsequent `__syncthreads` calls. Exposing this instruction level parallelism is beneficial as it helps to hide memory access latency. Our split tiling implementation uses loop unrolling to increase the amount of available instruction level parallelism.

4.4.3 *Full/partial tile separation*

One way to avoid overhead due to the evaluation of boundary conditions is to use full/partial tile separation [19, 61, 82]. The idea here is to generate specialized code for full tiles as well as for tiles that intersect with the iteration space boundary (i.e., partial tiles). Due to the absence of checks for the iteration space boundaries, the code of the full tiles evaluates a lot less conditionals. This is beneficial not only due to the reduced number of evaluated conditions, but also as it opens up new possibilities for loop-invariant code motion. Our split tiling compiler automatically performs full/partial tile separation (Chapter 10).

4.5 SUMMARY

In this chapter we presented a formulation of split tiling as a polyhedral schedule transformation using index set splitting to assign appropriate schedules to the different tile shapes. Our split tiled schedule is constructed directly from the dependence vectors without the need

to formulate and solve ILP problems to optimize tile shapes. It is formulated for an arbitrary number of space dimensions, the extracted parallelism is not limited to a single dimension, and no fixed relation between time tile height and the tile width along space dimensions is imposed.

Our split tiling algorithm has been implemented as a prototype extension to PPCG, which allows us to automatically generate split tiled CUDA. The CUDA code is generated such that it uses software managed shared memory and exploits instruction level parallelism. By using full/partial tile separation we avoid overhead of boundary condition checking in the core computation.

The techniques in this chapter have been collaboratively developed and an earlier version of the text in this chapter has been published in [2], with the experimental results of this paper being presented in Chapter 7. To enable the integration of our tiling scheme into PPCG, Sven Verdoolaege implemented some of the necessary changes in PPCG.

5 HYBRID HEXAGONAL/PARALLELOGRAM TILING

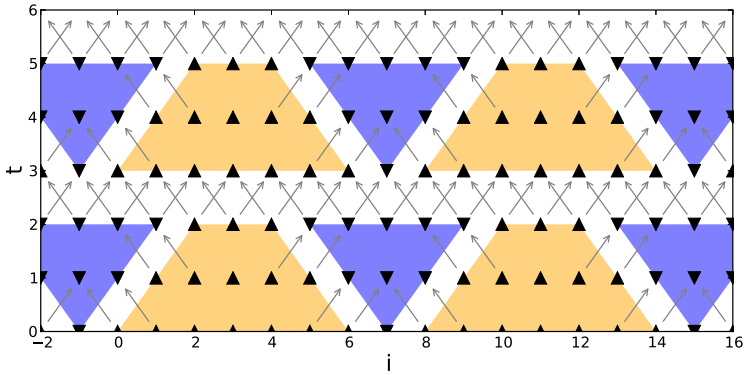
In this chapter[1] we develop a new tiling scheme called hybrid hexagonal/parallelogram tiling. Hybrid hexagonal/parallelogram tiling is a combination of hexagonal tiling on one space dimension and classical parallelogram tiling on the remaining space dimensions. We develop this tiling scheme with the goal to create a specialized tiling scheme that addresses the large number of GPU specific concerns necessary to reach optimal performance. One observation we can make is that by only extracting coarse-grained parallelism from one dimension, we can improve over split tiling. Figure 14a shows again split tiling of a one dimensional stencil code. We can optimize this code by mirroring the tile shapes in each second row such that neighboring tiles of the same color can be merged to form larger, hexagonal tiles (Figure 14b). Due to this merge, the data movement on the band boundaries can be fully omitted and the amount of computation executed in a single tile is doubled without increasing the data the tile works on, both purely beneficial transformations.

To obtain high performance GPU code many different concerns need to be addressed. Doing so requires both the development of a specialized tiling scheme as well as the actual generation of highly efficient program code. Both of these concerns are closely related. In the best case, a good tiling scheme does not only improve reuse or reduce the cache footprint, but also enables the generation of efficient code. In the following sections we show how complementing hexagonal tiling with parallelogram tiling enables us to refine the GPU code generation scheme introduced with split tiling to address a wide range of GPU specific concerns, many of them essential to maximize performance. Our work aims to decouple domain-specific scheduling transformations from general purpose low level optimizations by using a general purpose AST generator which is parameterized to exploit the domain specific tiling strategies. Thanks to this approach we can generate highly optimized and specialized code without the need to implement any specialized code generation strategies.

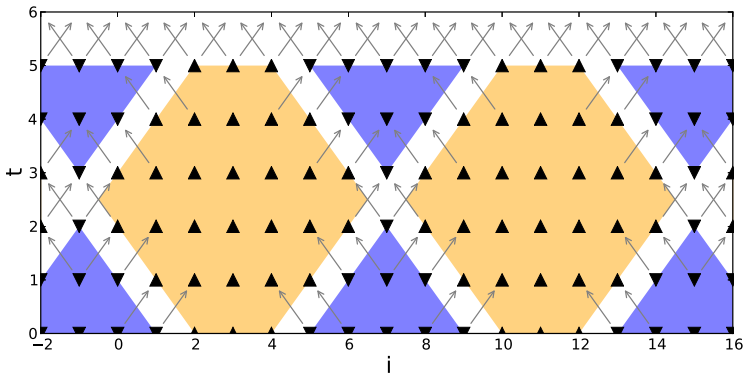
5.1 OVERVIEW

An effective tiling scheme for GPUs must address a number of constraints. It must carefully specialize unrolled inner loops to avoid divergent control flow among threads, it must minimize cumbersome address computations, effectively exploit register reuse, access shared

1 The text of the following chapter is a modified version of [1].



(a) split tiling



(b) merged tiles

Figure 14: 1D Hexagonal tiling - Created from 1D Split tiling

memory instead of global memory as often as possible while avoiding bank conflicts, and achieve coalesced transfers for essential global memory accesses.

Figure 15 shows a 2D Jacobi stencil in source form, and Figure 16 shows the core of the PTX code, as generated by our tool and extracted from the CUDA compiler. This highly tuned block is free of control flow, performs only 3 shared memory loads and 1 store for 5 compute instructions, no global memory access, and 2 out of the 5 values in flight are being reused in registers across sequential time steps.

Generating such optimized core loops and thread code is a significant challenge, especially for higher-dimensional stencils. We address this challenge by developing a sophisticated tiling scheme, paired with an advanced code generation strategy.

We choose a hybrid tiling scheme that combines hexagonal tiling on the outer dimension with classical parallelogram tiling on all remaining ones. Like most tiling schemes, our approach enables *reuse along the time dimension* while ensuring *balanced parallelism*, but hybrid tiling also addresses issues that make other approaches difficult to use on GPUs. In contrast to overlapped tiling [70], we perform *no redundant computations* and more importantly we avoid reserving shared

```
for (t=0; t < T; t++)
 for (i=1; i < N-1; i++)
  for (j=1; j < N-1; j++)
   A[(t+1)%2][i][j] = 0.2f * (A[t%2][i][j] +
        A[t%2][i+1][j] + A[t%2][i-1][j] +
        A[t%2][i][j+1] + A[t%2][i][j-1]);
```

Figure 15: Jacobi 2D stencil

```
ld.shared.f32   %f361, [%rd10+8200];
add.f32         %f362, %f353, %f361;
add.f32         %f363, %f362, %f345;
ld.shared.f32   %f364, [%rd10+7656];
add.f32         %f365, %f363, %f364;
ld.shared.f32   %f366, [%rd10+7648];
add.f32         %f367, %f365, %f366;
mul.f32         %f368, %f367, 0f3E4CCCCD;
st.shared.f32   [%rd10+1624], %f368;
```

Figure 16: Generated PTX (CUDA bytecode)

memory space for data used only in redundant computations. This is important to ensure a *high compute-to-memory ratio* for each tile. Our hexagonal tiling approach is closely related to diamond tiling [21], but has two important differences. First, diamond tiles always have a narrow peak, whereas the peak of hexagonal tiles is adjustable in width. For stencil codes, adjusting the width translates into a *wider range of tile size choices* and into the possiblity to take adavantage of *fine-grained parallelism even on the hexagonally tiled dimension* - the latter being useful in case there is insufficient parallelism available on the classically tiled dimensions. The second difference is that for diamond tiling, even though all tiles may have identical shapes, the actual number of integer points may vary between different tiles (see Chapter 6 for details). This difference may induce control flow divergence, when the diamond peaks sometimes fall on an integer point and sometimes do not. Our hexagonal tiling ensures an *identical number of computations within each full tile*.

Since hexagonal tiling along all spatial dimensions is not required to achieve an adequate degree of coarse-grained parallelism across thread blocks, we combine hexagonal tiling on an outer spatial dimension with classical tiling along the other dimensions, thereby binding the data footprint of tiles to enable *all temporary values to be kept in shared memory*. Also due to the use of classical tiling we can ensure that the width along the classical tiled dimension remains constant. By setting the tile width to a multiple of the warp size we can always ensure *full warp execution*, *stride one accesses* and *avoidance of bank con-*

flicts. Also, as tiles are now always offset by a multiple of the warp size, we can position them to always ensure *cache-line aligned loads*.

With our advanced code generation strategy we also exploit the fact that along the classical tiled dimension, tiles are executed in sequence. This enables them to be executed in the same kernel thread and thereby to *exploit reuse between successive tiles*. This is by itself already beneficial, but the real benefit is that the set of values that need to be loaded per tile is now a multiple of the tile width, which when chosen to be a multiple of the warp size will ensure that we *always load full cache lines*. Finally, we want *no conditional execution* and *no thread divergence* in the core computation. To ensure this we parameterize our code generation to create specialized code for full and for boundary tiles separately and we extensively unroll the innermost loops.

5.2 THE HYBRID HEXAGONAL/PARALLELOGRAM SCHEDULE

To calculate a hybrid hexagonal/parallelogram schedule that can be mapped nicely to the CUDA execution model we take several steps. First, the input program (Section 4.2) is analyzed statically and translated into a polyhedral representation. This representation is then canonicalized for stencil computations. Next, from this abstract information we derive a hexagonal schedule as well as a set of classically tiled schedules. Finally, the individual schedules are then combined into a hybrid hexagonal/classical execution schedule that materializes the ordering of iterations in a hybrid hexagonal/classical tiling. In addition, we explain how the calculated description of our tile shapes can be used to select good tile sizes.

5.2.1 *Hexagonal tiling*

We build hexagonal tiles starting from a two dimensional schedule space $P = (t, s_0)$ and a set of dependences $D \subseteq P \times P$. We first describe the restrictions on the input problem, then we construct the hexagonal tile shape and derive from it a hybrid tiling schedule. Finally, we show that the algorithm computes a correct tiled iteration space and that it allows parallel execution of the inner tile dimension.

5.2.1.1 *Constraints on input*

We require that the lexicographic order of the iterations in P is a valid schedule and that all dependences in D are such that t, the outer dimension of the index space, carries all dependences. As a result, the inner dimension s_0 is fully parallel. Finally, we assume that the dependence distances in the s_0-direction are bound by a fixed constant times the dependence distance in the t-direction, both from

above and below. Essentially, this assumption corresponds to the fact that we are dealing with a stencil computation.

5.2.1.2 *Hexagonal tile shapes*

To derive the tile shape of our hexagonal tiling we calculate two valid tiling hyperplanes from our dependences and use those hyperplanes to construct a tile shape for a given height h and width w_0. We illustrate the process on a slightly contrived example that computes

```
A[t][i] = f(A[t-2][i-2], A[t-1][i+2]);
```

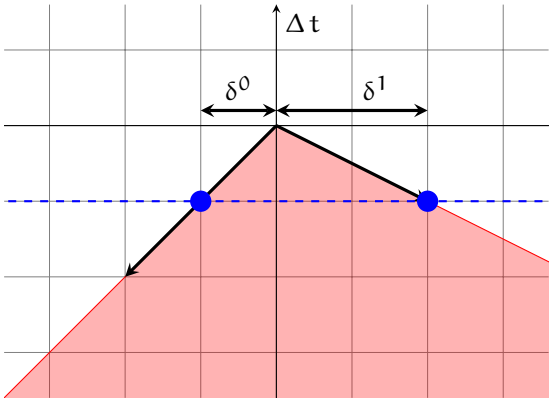


Figure 17: Opposite dependence cone

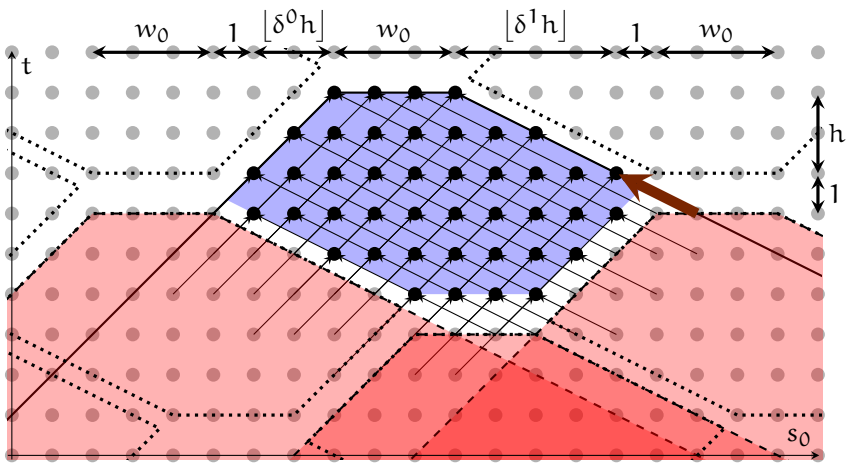


Figure 18: A hexagonal tile

We derive the tiling hyperplanes from the given dependences. We first compute the set of dependence distance vectors. In the example, we have $\{(1,-2);(2,2)\}$, meaning that the statement instances that directly depend on a given statement instance are executed in the original schedule at an offset $(\Delta t, \Delta s_0) = (1,-2)$ or $(2,2)$. Conversely, the opposites of these distance vectors are the offsets of state-

ment instances on which the current statement instance directly depends. The cone generated by these opposite distance vectors is an over-approximation of the set of offsets of statement instances on which the current statement instance depends directly or indirectly. This cone (for the example) is shown as the red area in Figure 17. As we required the input to have strictly positive dependence distances in the first dimension, the cone lies entirely in the negative Δt half-space. Furthermore, because of our requirement of bound distances in the s_0-direction, we can compute constants δ^0 and δ^1 such that $\Delta s_0 \leqslant \delta^0 \Delta t$ (or, equivalently, $-\Delta s_0 \geqslant \delta^0(-\Delta t)$) and $\Delta s_0 \geqslant -\delta^1 \Delta t$. These constants can be computed through the solution of an LP-problem. Figure 17 shows the points $(-1, -\delta^0)$ and $(-1, \delta^1)$ in blue and the cone generated by these two points in red.

The basic idea is now that a tile will compute one or more s_0-instances at a given time step t together with all the instances on which it depends, except those that have already been computed by previous tiles. We therefore take $w_0 + 1$ instances at a given time step and construct a truncated cone that contains all the instances on which these selected instances depend by taking the union of the opposite dependence cones (the red cone from Figure 17) shifted to each of these instances. Figure 18 shows three such truncated cones in red, bound by dashed lines. The blue tile shape is the result of subtracting these three truncated cones from the truncated cone bound by solid lines. The offsets of the truncated cone have been carefully selected such that the entire space can be tiled using a single shape. In particular, the truncated cone on the left has offset $(-h-1, -w_0 - 1 - \lfloor \delta^0 h \rfloor)$, the cone on the right has offset $(-h-1, w_0 + 1 + \lfloor \delta^1 h \rfloor)$ and the cone on the bottom has offset $(-2h-2, \lfloor \delta^1 h \rfloor - \lfloor \delta^0 h \rfloor)$. The tiling is shown in dotted lines. In the figure, $w_0 = 3$ and $h = 2$. If there are multiple statements in the kernel, then choosing h such that $h+1$ is a multiple of the number of statements ensures that each tile starts with the same statement. To ensure that the result of the subtraction is a convex shape, the width w_0 has to be large enough. This is illustrated by the large brown dependence vector in Figure 18. If w_0 were equal to 1, then the result of the subtraction would contain an extra component to the right of the right truncated cone. Such extra components can be avoided by imposing

$$w_0 \geqslant \max\left(\delta^0 + \left\{\delta^0 h\right\}, \delta^1 + \left\{\delta^1 h\right\}\right) - 1, \tag{3}$$

with $\{x\}$ the fractional part of x, i.e., $\{x\} = x - \lfloor x \rfloor$. In the example, we have $w_0 \geqslant 1$. The correctness of (3) will be shown in Section 5.2.1.3.

5.2.1.3 *Scheduling hexagonal tiles*

The schedule of our hexagonal tiling maps the two iteration space dimensions $[t, s_0]$ into a three dimensional tile space $[T, p, S_0]$. The schedule alternates between two phases, 0 and 1. In particular, within

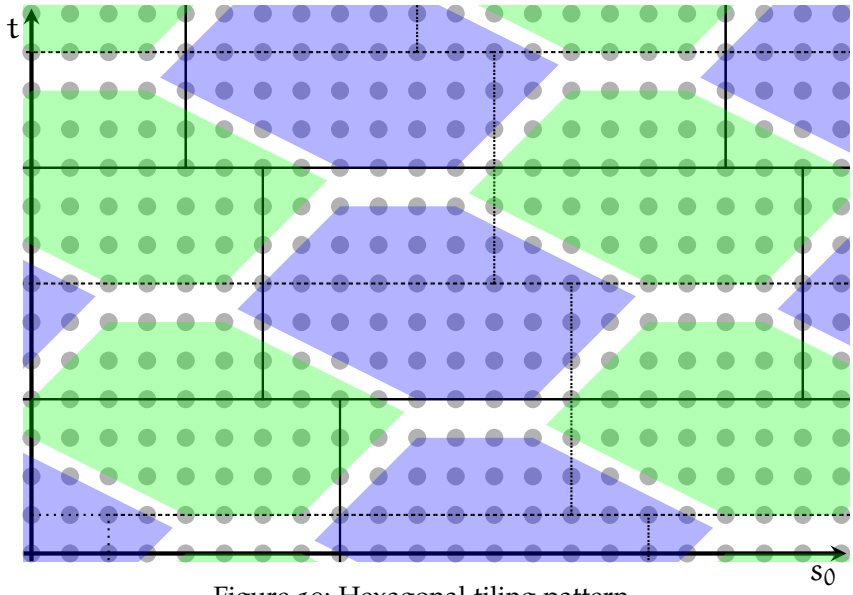


Figure 19: Hexagonal tiling pattern

each time tile T, the schedule first executes the blue tiles of Figure 19 (phase 0) and then the green tiles (phase 1). The tiles that belong to the same time tile and the same phase are indexed by S_0 and can be executed in parallel. In Figure 19 such tiles form a horizontal wavefront of identically colored tiles. For phase 0, we have

$$T = \lfloor (t + h + 1)/(2h + 2) \rfloor \tag{4}$$

$$S_0 = \left\lfloor \frac{s_0 + \lfloor \delta^1 h \rfloor + w_0 + 1 + T\left(\lfloor \delta^1 h \rfloor - \lfloor \delta^0 h \rfloor\right)}{2w_0 + 2 + \lfloor \delta^0 h \rfloor + \lfloor \delta^1 h \rfloor} \right\rfloor \tag{5}$$

while for phase 1, we have

$$T = \lfloor t/(2h + 2) \rfloor \tag{6}$$

$$S_0 = \left\lfloor \frac{s_0 + T\left(\lfloor \delta^1 h \rfloor - \lfloor \delta^0 h \rfloor\right)}{2w_0 + 2 + \lfloor \delta^0 h \rfloor + \lfloor \delta^1 h \rfloor} \right\rfloor . \tag{7}$$

The difference in the numerators of the expressions for T ensures that the blue tiles belong to the same T-tile as the green tiles that have the same and greater t coordinates. Within this T-tile, the blue tiles are then executed before the green tiles. The other offsets are required to make all the tiles line up.

The (T, S_0)-coordinates refer to the boxes in Figure 19, the solid boxes for phase 0 and the dotted boxes for phase 1. To ensure that each (t, s_0) is only executed once, we only execute parts of these overlapping boxes. In particular, we execute the blue tile in each solid box and the green tile in each dotted box. To describe the hexagons,

we use local coordinates (a, b) within each box. For example, for the green tiles, we have

$$a = t \bmod (2h+2)$$
$$b = s_0 + T\left(\left\lfloor \delta^1 h \right\rfloor - \left\lfloor \delta^0 h \right\rfloor\right) \bmod \left(2w_0 + 2 + \left\lfloor \delta^0 h \right\rfloor + \left\lfloor \delta^1 h \right\rfloor\right).$$

Using these local coordinates, the constraint of the top of the hexagons can be derived directly from the constraints of the opposite dependence cone. In particular, we have

$$\delta^0 a - b \leqslant (2h+1)\delta^0 - \left\lfloor \delta^0 h \right\rfloor \tag{8}$$
$$a \leqslant 2h+1 \tag{9}$$
$$\delta^1 a + b \leqslant (2h+1)\delta^1 + \left\lfloor \delta^0 h \right\rfloor + w_0. \tag{10}$$

The remaining constraints are obtained from subtracting the earlier truncated cones. Let (a', b') be the local coordinates in the box at offset $(-h-1, -w_0 - 1 - \left\lfloor \delta^0 h \right\rfloor)$, i.e., $a' = a + h + 1$ and $b' = b + w_0 + 1 + \left\lfloor \delta^0 h \right\rfloor$. When subtracting the truncated cone associated to this box, we need to add the negation of the constraint

$$\delta^1 a' + b' \leqslant (2h+1)\delta^1 + \left\lfloor \delta^0 h \right\rfloor + w_0, \tag{11}$$

i.e., $\delta^1 a + b \leqslant h\delta^1 - 1$. Let d^1 be the denominator of δ^1. The negation of this constraint can then be written as

$$\delta^1 a + b \geqslant h\delta^1 - \frac{d^1 - 1}{d^1}. \tag{12}$$

In principle, we now also need to consider other pieces of the difference that satisfy (11), but that do not satisfy one of the other two constraints. Because of the vertical position of the truncated cone we are subtracting it is impossible for there to be any integer points that lie in the original truncated cone, satisfy (11) and do not satisfy $a' \leqslant 2h+1$. To verify that there can be no points in the current truncated cone that do not satisfy the constraint

$$\delta^0 a' - b' \leqslant (2h+1)\delta^0 - \left\lfloor \delta^0 h \right\rfloor, \tag{13}$$

we again rewrite the constraint in terms of the current local coordinates and obtain

$$\delta^0 a - b \leqslant (2h+1)\delta^0 - \left\lfloor \delta^0 h \right\rfloor + w_0 + 1 + \left\lfloor \delta^0 h \right\rfloor - \delta^0(h+1).$$

Due to our choice of w_0 in (3), we have $w_0 - \delta^0 - \left\{\delta^0 h\right\} + 1 \geqslant 0$, meaning that (13) is implied by the corresponding constraint on the original truncated cone.

The truncated cone at offset $(-h-1, w_0 + 1 + \left\lfloor \delta^1 h \right\rfloor)$ similarly yields the constraint

$$\delta^0 a - b \geqslant \delta^0 h - \left\lfloor \delta^0 h \right\rfloor - w_0 - \left\lfloor \delta^1 h \right\rfloor - \frac{d^0 - 1}{d^0}, \tag{14}$$

with d^0 the denominator of δ^0. Finally, the box at offset $(-2h-2, \left\lfloor \delta^1 h \right\rfloor - \left\lfloor \delta^0 h \right\rfloor)$ yields the constraint

$$a \geqslant 0. \tag{15}$$

5.2.2 *The parallelogram tile schedule*

In the remaining spatial dimensions, we apply a more traditional form of tiling. This means that we lose parallelism along these dimensions, but it allows the reduction of the working set within each tile. Each spatial dimension s_i with $i \in [1, n]$ is strip-mined separately. Just like hexagonal tiling (see Figure 17), one computes the projection of the dependence cone onto the time dimension and the given spatial dimension s_i. Yet in this case, we only need to consider dependences on statement instances with higher values for the spatial dimension. This means that we only need to compute δ_i^1 and that therefore the dependence distance in the spatial dimension only needs to be bound in terms of the distance in the time dimension from below. The resulting tile shape is a parallelogram with sides that are parallel to the corresponding side of the opposite dependence cone. Since this tiling needs to be combined with the hexagonal tiling, the height of these tiles is equal to $2h + 2$. The width can be independently chosen as w_i. In sum, the corresponding tile dimension is given by

$$S_i = \left\lfloor (s_i + \delta_i^1 u)/w_i \right\rfloor, \tag{16}$$

where u is a normalized version of t that ensures that the starting positions of the tiles in the spatial direction are the same for all time tiles and for both phases. That is, we set

$$u = (t + h + 1) \bmod (2h + 2) \qquad \text{for phase 0 and} \tag{17}$$

$$u = t \bmod (2h + 2) \qquad \text{for phase 1.} \tag{18}$$

The above normalization is beneficial in two ways. Firstly, the generated code is simpler because the offset is a constant instead of an expression that needs to be (re)calculated at each time tile step. Secondly, constant offsets make it easier to align the load instructions that fetch data from global to local memory. This is because the location and alignment of the load instructions directly depends on the position of the individual tiles.

5.2.3 *Intra-tile schedules*

We also specify non-trivial intra-tile schedules $t', s_0', \ldots, s_n'$. It is desirable to minimize the intra-tile coordinates of the schedule, ideally starting from zero, to ensure an efficient thread to iteration mapping. To achieve this, we derive the intra-tile schedules from the tile schedule by replacing the outermost integer division by the corresponding remainder. For the classically tiled dimension this yields

$$s_i' = (s_i + \delta_i^1 u) \bmod w_i, \tag{19}$$

5.2.4 *Hybrid tiling*

The final hybrid tiling is a combination of the hexagonal tiling of Section 5.2.1 and the classical tiling of Section 5.2.2 as well as the intra-tile schedules of Section 5.2.3. This tiling is of the form

$$\{(t, s_0, \ldots, s_n) \to (T, p, S_0, \ldots, S_n, t', s'_0, \ldots, s'_n)\}$$

with tile dimensions defined by (4), $p = 0$, (5) (for S_0), (16) (for S_i with $i \geqslant 1$) and (17) for phase 0 and by (6), $p = 1$, (7), (16) and (18) for phase 1. Each phase is only applied to the subset of the domain that satisfies the conditions (8), (10), (12) and (14) in the local coordinates of the rectangular tile defined by (T, p, S_0). The constraints (9) and (15) are automatically satisfied for all points in the rectangular tile. As an example, Figure 20 shows the phase-0 part of a hybrid tiling where all δs are equal to 1.

$$\begin{aligned}
&\Big\{(t, s_0, s_1, \ldots, s_n) \to (T, 0, S_0, S_1, \ldots S_n, t', s'_0, s'_1, \ldots, s'_n) \mid \\
&\quad \exists a, b : a = (t + h + 1) \bmod (2h + 2) \wedge \\
&\qquad\quad b = (s_0 + h + 1 + w_0) \bmod (2h + 2 + 2w_0) \wedge \\
&\qquad\quad a - b \leqslant h + 1 \wedge a + b \leqslant 3h + 1 + w_0 \wedge \\
&\qquad\quad a + b \geqslant h \wedge a - b \geqslant -w_0 - h \wedge \\
&\qquad\quad T = \lfloor (t + h + 1)/(2h + 2) \rfloor \wedge \\
&\qquad\quad S_0 = \lfloor (s_0 + h + 1 + w_0)/(2h + 2 + 2w_0) \rfloor \wedge \\
&\Big(\bigwedge_{k:1 \leqslant k \leqslant n} S_k = \lfloor (s_k + ((t + h + 1) \bmod (2h + 2)))/w_k \rfloor \Big) \wedge \\
&\qquad\quad t' = (t + h + 1) \bmod (2h + 2) \wedge \\
&\qquad\quad s'_0 = (s_0 + h + 1 + w_0) \bmod (2h + 2 + 2w_0) \wedge \\
&\Big(\bigwedge_{k:1 \leqslant k \leqslant n} s'_k = (s_k + ((t + h + 1) \bmod (2h + 2))) \bmod w_k \Big) \Big\}
\end{aligned}$$

Figure 20: n-dimensional tile schedule (± 1 distances)

The schedule is parameterized with the values h, $w_0, \ldots, w_n$. The parameter h allows to adjust the distance between two subsequent tiles on the time dimension, and the different values w_i define the distance between subsequent tiles along the space dimensions s_i. For dimensions s_i with $i \geqslant 1$ the parameter w_i gives the exact width along this dimension, whereas for the dimension s_0 the value of parameter w_0 only gives the minimal width. The maximal tile width along this dimension may increase depending on the current time step.

It should be noted that there is no need to map the spatial dimensions in the order to $s_0, \ldots, s_n$ in which the spatial loops are nested in the input code. Instead, any spatial dimension can be chosen as the one that is hexagonally tiled. However, to ensure our assumptions

about aligned and coalesced memory accesses hold, it is necessary that the innermost dimension is the dimension that yields stride one access. This is a property that inputs normally already have and that we currently rely on.

5.2.5 *Tile size selection*

In order to determine appropriate values for the tile size parameters h and w_i, we use a simple model based on the load-to-compute ratio. In particular, we take a generic tile (not at the border) and compute the number of iterations in the tile and the number of loads performed by the tile. Since the set of iterations and the set of loads can be described using quasi-affine constraints, these numbers can be computed exactly as a function of the tile size parameters. For the experiments in this work, we use manually derived functions, but tools to count points in integer polyhedra [132] can automate this. For a 3D stencil with $\delta^0 = \delta^1 = 1$, the number of iterations in a tile is $2(1 + 2h + h^2 + w_0(h + 1))w_1w_2$, while the number of loads depends on the type of stencil and on various optimization choices described in Section 5.3. We then evaluate these formulas for all values of the tile size parameters that yield a memory tile size within a specified bound and select those parameters that yield the smallest load-to-compute ratio.

5.3 CUDA CODE GENERATION

To generate GPU code, we use the same infrastructure as already described in Section 4.4. We continue to use the generic CUDA code generator of PPCG and get the previously described benefits of a generic infrastructure. On top of the already known optimizations, we add additional optimizations that ensure that the code that is generated exploits the optimization opportunities exposed by our hybrid-hexagonal schedule. To avoid the implementation of domain specific transformations we use a decoupled approach where we only use general purpose options to help the AST generator to generate code that fits our hybrid-hexagonal schedule.

5.3.1 *Generating CUDA code*

Our tool uses the previously generated hybrid schedule to create CUDA code by mapping the schedule's output dimensions $(T, p, S_0, S_1, \ldots, t, s_0, s_1)$ to nested loops in the generated code. The T dimension is mapped to the host code, where it takes the form of a `for` loop repeatedly iterating over two CUDA kernels — one kernel for $p = 0$ and the other one for $p = 1$. For each kernel call, the dimension S_0 is mapped to a one dimensional grid of thread blocks that are executed in parallel.

In case dimension S_0 has more elements than there are thread blocks supported by CUDA, the individual thread blocks execute multiple elements of S_0.

The remaining dimensions $(A_1, \ldots, S_n, t, s_0, \ldots, s_n)$ are code generated within each kernel. The dimensions $(S_1, \ldots, S_n, t)$ are code generated as sequential loops. As the dimensions $(s_0, \ldots, s_n)$ are fully parallel they can be mapped to different CUDA thread dimensions. In case there are more parallel dimensions than there are CUDA thread dimensions, the outer dimensions will be enumerated sequentially. To ensure all iterations of a dimension are executed even though there may be more iterations than threads in a thread block, additional iterations are assigned to threads in a cyclic way: iteration i is mapped to thread $i \bmod T_i$ with T_i being the number of threads used for dimension i. The sequential execution of subsequent time steps is ensured by generating a synchronization call at the end of each iteration of the sequential loops.

5.3.2 *Shared memory*

For hybrid-hexagonal tiled code the use of explicitly managed shared memory can be more efficient than a hardware managed cache. PPCG provides the following cache management strategy. Instead of performing all computations on global memory, PPCG allocates shared memory of the size of the smallest rectangular box that is large enough to accommodate the data accessed within a single tile. Now instead of just performing the computation of each tile, PPCG generates code that loads all data from global to shared memory, executes the computation on shared memory, and finally writes the modified elements back to global memory. To avoid thread divergence in the load phase, PPCG can over approximate the shape of the values to load with the rectangular box used to define the shared memory allocation.

5.3.2.1 *Inter-tile reuse*

Reducing the number of loads from global memory by reusing values already available in shared memory is another beneficial optimization. Due to the sequential execution of tiles enforced by the classical schedule at the inner dimension, it is possible to access values loaded by previous tiles without introducing any memory conflicts. For our tiling scheme some of the values used in one tile have already been made available by the preceding tile, either because they are used by the preceding tile itself or because the preceding tile over-approximates the values it loads from global memory. To make such values available to our current tile, we can directly move them from the shared memory location assigned in the preceding tile to the shared memory location where the current tile expects those values to be.

Another option would be to enforce a static mapping, where a single global location is always mapped to the same shared memory location. While this would eliminate the internal shared memory copy, accesses to statically mapped shared memory may induce more complex access patterns and can also cause bank conflicts, which both hurt the overall performance.

5.3.2.2 *Aligned loads*

It is important to ensure that loads from global memory to shared memory are aligned to cache line boundaries. The location of the data that is loaded from global memory directly depends on the position of the tiles in space, specifically, the offsets of the tiles along the different space dimensions. When calculating the schedule we ensured that all these offsets are independent of the time dimension T. Assuming the size of the innermost data space dimension is a multiple of the minimal alignment, we select a tile width along the innermost dimension that is also a multiple of the minimal alignment. This ensures that as soon as the first load from an array is perfectly aligned, the subsequent loads are also perfectly aligned. We allow the tiles in the schedule to be translated by manually specifying the translation offset. By specifying the right offset it is possible to fully align the initial (and therefore all) global memory loads from a specific array. In case of multiple arrays, it may not always be possible to align the loads from all arrays.

5.3.3 *Interleaving computations and copy-out*

When developing our hybrid-hexagonal tiling we have seen that the separate copy-out phase makes the shared memory usage inefficient due to a possibly complex to describe set of values that needs to be copied out, but also due to the absence of overlap between the compute and the copy phase. We consequently extended the generic code generator to optionally write out values right at the time at which they are calculated. The unnecessary stores that may possibly be introduced are not overly costly, as for stencils the number of stores is low compared to the number of reads. Also, because our hybrid schedule ensures no thread divergence in the compute phase, executing the copy out next to the computation avoids all thread divergence.

5.3.4 *Stencil specific code generation heuristics*

During the final translation from the polyhedral program representation back to an abstract syntax tree (AST), domain specific knowledge can be used to adapt the code generation heuristics. The same schedule can be written out as an AST in many different ways, resulting

in code that is functionality equivalent but that may have different performance behavior. The `isl` AST generator (Chapter 10) offers a flexible mechanism for allowing the user to choose between different ways of generating code across different parts of the schedule. We exploit this flexibility to implement specialized code generation heuristics for hybrid tiling.

5.3.4.1 *Specialized code for the core computation*

To generate optimal code for the core part of the computation we parameterize the code generation strategy such that specialized code is generated for full tiles and generic code for the remaining partial tiles.

When generating our schedule we have been especially careful to ensure that the number of integer points contained in a tile is the same for all tiles in the program and that the offsets used to derive the iterations that belong to a tile are constant within a single phase of our tiling scheme. We also made sure that within a core tile, there is no need for conditional execution that would cause thread divergence. To ensure that the simplicity of the core tiles is maintained and not lost by the need to handle rarely executed boundary cases we pass a description of the full tiles to `isl`'s AST generator, instructing it to generate code for these full tiles and the remaining partial tiles separately.

5.3.4.2 *Unrolling for hybrid tiled stencils*

Unrolling is often beneficial, but it is especially profitable in conjunction with our hybrid approach. As stated in the previous section, we construct a hybrid schedule such that the core computation is free of any thread divergence. In fact it does not require conditional control flow. However, due to the limited amount of shared memory and the large number of parallel threads, the number of iterations that need to be executed within a single thread is relatively low. Hence, we can unroll the point loops within the tile to create straightline code. This also contributes to exposing instruction level parallelism. Furthermore, depending on the tiling parameters chosen, we unroll neighboring points next to each other such that they can use a single load to get values that are within the neighborhood of both points.

Note that unrolling is not performed at the AST level, but on the constraint representation of the kernel. Constraint-based unrolling ensures that all conditions can be specialized or eliminated in the unrolled code, simplifying them according to the context in which an instruction is unrolled [124].

5.4 SUMMARY

In this chapter we introduced a hybrid hexagonal/parallelogram tiling scheme for stencils which addresses a large number of GPU specific concerns relevant for the generation of high-performance parallel GPU code. With the use of hexagonal tile shapes, an evolution of split-tiling for one dimension, we ensured both balanced, coarse-grained parallelism and reuse along the time dimension while maintaining flexible tile sizes. By combining it with parallelogram tiling we obtained a schedule that enables the generation of highly optimized code where the core computation has coalesced accesses to global memory, aligned loads and is free of thread divergence.

To actually generate high performance code we based our work again on PPCG, which allows us to automatically generate highly specialized GPU code.

The techniques in this chapter have been collaboratively developed and an earlier version of the text in this chapter has been published in [1], with the experimental results of this paper being presented in Chapter 7. To enable the integration of our tiling scheme into PPCG, Sven Verdoolaege implemented some of the necessary changes in PPCG.

6 UNIFICATION WITH DIAMOND TILING

In the previous chapters we discussed split and hybrid tiling as tiling strategies that enable time tiling without requiring redundant computations. An alternative to these techniques is diamond tiling [21]. Even though these different tiling schemes are closely related the exact relation is by far not obvious and several properties of the different tiling schemes are underspecified. This becomes very clear by comparing diamond tiling and hexagonal tiling at a very high level.

Similar to the previously presented approaches, diamond tiling enables concurrent start without requiring redundant computations, this time by using n-dimensional parallelotopes[1] as tile shapes. In contrast to previous approaches, these tile shapes are not directly derived from the set of data dependences. Instead, diamond tiling was presented as an extension to a general purpose scheduling optimizer which uses an adaptable cost function to determine the optimal tile shapes [21]. This design choice means that the computation of tile schedules now requires the solution of ILP problems. Even though this may be an expensive operation, it allows the use of a possibly more complex cost model and the integration into a general purpose optimizer.

In contrast to diamond tiling, with hexagonal tiles it is possible to adjust the time-tile height and the tile width along the spatial dimension independently. Hexagonal tiling also permits the creation of tiles with a flat top and it ensures that tiles not only have the same rational shape, but also identical integer point placements. Diamond tiles do not have these properties. Furthermore, certain properties of diamond tiling have not been discussed in the original publication. Even though the diamond tiling paper generally explains how to derive tiling hyperplanes that enable concurrent start, a tile schedule that includes both the tile sizes as well as the parallel wavefront coefficients necessary to obtain concurrent start was not presented.

We believe there is a clear need for a more precise analysis and comparisons of these tiling techniques. To address this need we provide in this chapter an analysis of diamond tiling to understand the properties previously discussed. Using this information we develop a new tiling strategy for two dimensional problems (one time dimension, one space dimension) that combines the positive features of diamond tiling and hexagonal tiling. Using this new tiling scheme we analyze

1 A general term for what is known in 2D as parallelogram and in 3D as parallelepiped.

the effect of tile shape and tile size choice on properties such as the compute-to-communication and compute-to-synchronization ratio.

This chapter[2] is structured as follows. Section 6.1 revisits diamond tiling, providing insights on tile size and wavefront coefficient constraints, and discussing constraints and important properties of diamond tiles. We then introduce the unified hexagonal tiling scheme in Section 6.2 which includes a full formulation for two-dimensional tiling. Section 6.3 studies tile sizes that maximize the compute-to-communication ratio and compares the synchronizations induced by diamond and hexagonal tile shapes. We summarize this chapter in Section 6.4.

6.1 DIAMOND TILING

The main contribution of diamond tiling [21] is the combination of affine transformations and a form of rectangular tiling that enables concurrent start. It is particularly effective on stencil computations. The idea of concurrent start is to ensure that the wavefront of tiles that are executed in parallel is aligned to a concurrent start hyperplane (normally an iteration space boundary) such that the number of tiles that are executed in parallel remains constant throughout the entire computation. This ensures that already at the beginning of the computation a sufficient amount of parallelism is available. Even though the name "diamond" suggests that the tile shapes are rhombi or rhombohedra (a.k.a. diamonds) and Figure 12 in Bandishti et al. [21] also uses edges of identical length, the tile shapes formed by diamond tiling are not restricted to diamonds, but can be more general parallelograms (parallelotopes in higher dimensions) as can be seen in Figure 23. However, some restrictions to the tile shape and sizes must be enforced to ensure that concurrent start is possible.

6.1.1 *The pluto optimizer*

Diamond tiling was presented and implemented as an extension to Pluto [35], a general-purpose optimizer for data locality and parallelism. In contrast to other approaches that directly tile the iteration space (e.g., Chapter 4, Chapter 5), the original Pluto tiling as well as diamond tiling are implemented as a two phase process. As a first step a program transformation is calculated that exposes sequences of loops (bands) that are tileable with rectangular tiles. In the second step a rectangular tiling is performed on these bands. Combined, this yields tiles with a possibly not rectangular, but parallelotope tile shape. There are several benefits of separating these two concerns. First, when calculating the parallel bands Pluto can and does perform other optimizations, e.g., data locality optimizations such as loop fu-

2 The text of the following chapter is a modified version of [7, 5].

sion. Second, tiling of the transformed program makes the tile shapes independent of the tiling hyperplanes, which makes the tiling easier to describe and analyze.

Pluto calculates program transformations on a polyhedral representation. In this representation the set of executed program statements (the iteration space) is modeled with a multi-dimensional integer set where each element represents an individual statement iteration. The execution order of elements of the iteration space is described by the schedule, an integer map that assigns a possibly multi-dimensional relative execution time to each element of the iteration space. Program transformations are performed by modifying the schedule. For a single statement and a k-dimensional execution time such a schedule has the form $S = \{\vec{x} \to (\vec{h}_0 \cdot \vec{x}, \ldots, \vec{h}_k \cdot \vec{x})\}$, where $\vec{x}$ is an element of the iteration space, $\vec{h}_i$, for $i \in \{0, \ldots, k\}$, are tiling hyperplanes represented by their normal vectors and $\vec{h}_i \cdot \vec{x}$ denotes the sum of the per element products of $\vec{h}_i$ and $\vec{x}$. The result of Pluto's first step are exactly these tiling hyperplanes, selected such that the distance between two statements that depend on each other is not only lexicographically nonnegative (needed for validity of the schedule), but also nonnegative at each individual dimension. As input, the algorithm takes an overapproximation of the pairs of statement instances that depend on each other, described using affine constraints. For the exact algorithm on how to select such hyperplanes, we refer to [35]. For the present discussion, it is sufficient to understand that the all-nonnegative dependence vectors make rectangular tiling valid.

We present the Pluto rectangular tiling as a schedule only transformation which we believe is easier to understand than the actual Pluto transformation which modifies the iteration space as well. Conceptually, there should be no difference. Given a schedule S and a set of tile sizes $s_i, i \in \{0, \ldots, k\}$ a rectangularly tiled schedule of S consists of two partial schedules. The first one, S_t, is placed at the outer level and enumerates the tiles itself. This is called the tile schedule. The second one, S_p, is placed at the inner level and enumerates the points within each tile and is called the point schedule. We define $S_t = \{(x_0, \ldots, x_k) \to (\lfloor(\vec{h}_0 \cdot \vec{x})/s_0\rfloor, \ldots, \lfloor(\vec{h}_k \cdot \vec{x})/s_k\rfloor)\}$ and $S_p = S$. This tiled schedule may already expose parallelism, but exploiting it may involve a skewed wavefront schedule at the outermost tile dimension. Then, such a wavefront schedule carries itself all dependences and ensures that the inner loops can be executed in parallel. This yields $S'_t = \{(x_0, \ldots, x_k) \to (\lambda_0\lfloor(\vec{h}_0 \cdot \vec{x})/s_0\rfloor + \cdots + \lambda_k\lfloor(\vec{h}_k \cdot \vec{x})/s_k\rfloor, \lfloor(\vec{h}_1 \cdot \vec{x})/s_1\rfloor, \ldots, \lfloor(\vec{h}_k \cdot \vec{x})/s_k\rfloor)\}$ with $\lambda_i \in \mathbb{Z}_{\geqslant 0}, i \in \{0, \ldots, k\}$. The λ_i coefficients control the construction of different wavefronts. We call $\lambda_0 = \cdots = \lambda_k = 1$ the default wavefront coefficients. The hyperplanes computed by the original Pluto algorithm allow the formation of such a wavefront schedule, but those hyperplanes may not

allow the formation of a wavefront schedule in the direction of a given concurrent start face (represented by its normal vector $\vec{f}$).

6.1.2 *The diamond tiling extensions*

Diamond tiling [21] extends the Pluto algorithm in a way that ensures that for the tiling hyperplanes computed there are always wavefront coefficients that yield concurrent start. From the original publication [21] we know that "a transformation enables tilewise concurrent start along a face $\vec{f}$ if and only if the tile schedule is in the same direction as the face and carries all inter-tile dependences". It also shows that "concurrent start along a face $\vec{f}$ can be exposed by a set of hyperplanes if and only if $\vec{f}$ lies strictly inside the cone formed by the hyperplanes, i.e., if and only if $\vec{f}$ is a strict conic combination of all the hyperplanes". For a concurrent start hyperplane $\vec{f}$, it finds tiling hyperplanes $\vec{h}_i$ such that the following equality holds:

$$m\vec{f} = \lambda_1\vec{h}_1 + \cdots + \lambda_k\vec{h}_k \qquad \text{with } \lambda_i, m \in \mathbb{Z}_{\geqslant 0}. \tag{20}$$

The main focus of the diamond tiling paper is to prove the conditions necessary to ensure that the calculated hyperplanes can be used to construct a concurrent start schedule as well as to give an algorithm that actually calculates such hyperplanes. We therefore refer to this publication for details. One question that was explored less is under which conditions, especially for which tile sizes and for which wavefront coefficients, the rectangularly tiled schedule achieves concurrent start. Specifically, the paper does not investigate for which values of λ_i, s_j the following holds:

$$m\vec{x} \cdot \vec{f} = \lambda_0\lfloor(\vec{h}_0 \cdot \vec{x})/s_0\rfloor + \cdots + \lambda_k\lfloor(\vec{h}_k \cdot \vec{x})/s_k\rfloor \tag{21}$$

6.1.3 *Relation between tile sizes and wavefronts*

Even though the diamond tiling yields tiling hyperplanes that allow concurrent start, to construct the full tile schedule the tile sizes s_i as well as the wavefront coefficients λ_i still need to be chosen. Choosing the correct values is important, not only to ensure that the tiles executed within the wavefront are started concurrently, but also to control the horizontal distance between neighboring tiles in the parallel wavefront and its ratio to the size of the tiles. We call this ratio the density of the schedule, a property important to understand the amount of computation that can be performed in parallel. Before suggesting good values, we explore the impact of different choices. Let us first consider a simple example with symmetric dependences:

```
for t:
  for i:
    A[t+1][i] = A[t][i-1] + A[t][i+1]
```

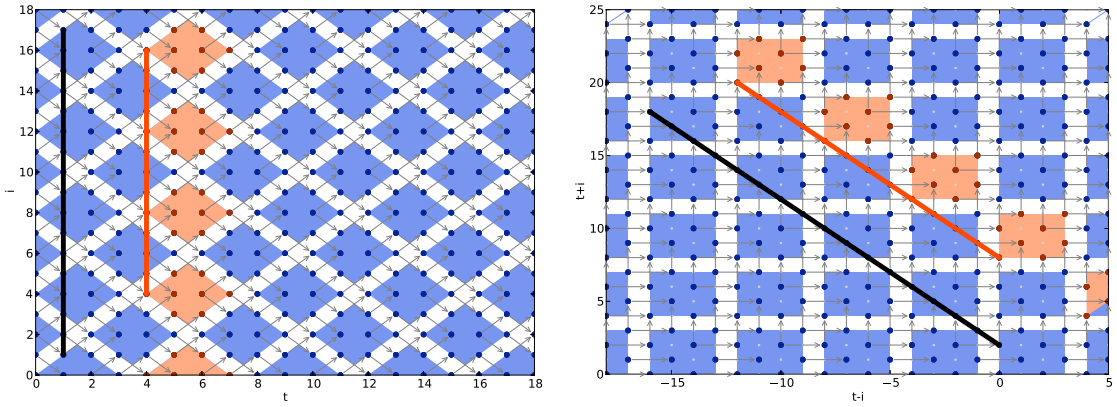


Figure 21: Symmetric dependences & square tiling (original/transformed)

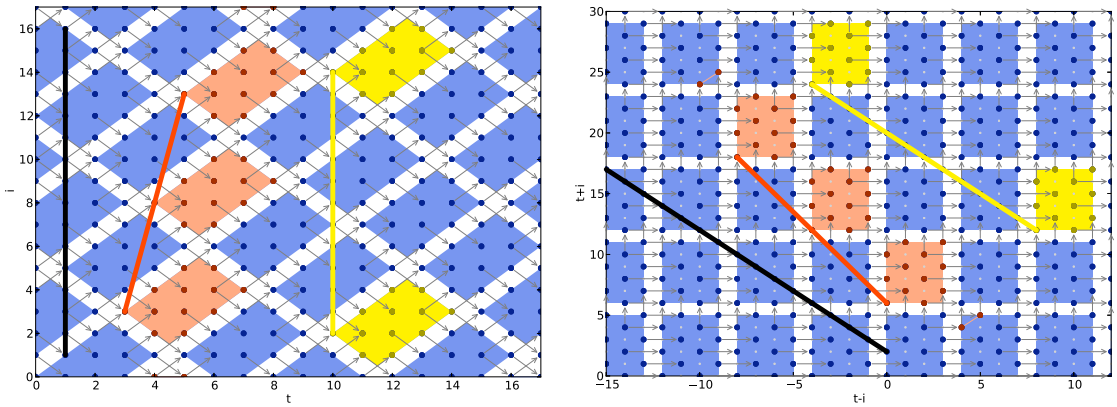


Figure 22: Symmetric dependences & non-square tiling (original/transformed)

Pluto's diamond tiling implementation[3] calculates for this kernel the transformation $\{(t,i) \to (t-i, t+i)\}$ and applies rectangular tiling in the transformed space. The default wavefront coefficients $\lambda_0 = \lambda_1 = 1$ are then used to enable parallel execution. This results in the tile schedule $\{(t,i) \to (\lfloor (t-i)/s_0 \rfloor + \lfloor (t+i)/s_1 \rfloor, \lfloor (t+i)/s_1 \rfloor)\}$. The default square tile shapes ($s_0 = s_1$) yield both concurrent start as well as a high density of tiles. Figure 21 illustrates this for $s_0 = s_1 = 4$ with the tile wavefront highlighted in red and the concurrent start hyperplane highlighted in black. The two hyperplanes being parallel tells us that the tile wavefront has concurrent start. When different tile sizes are chosen for the two dimensions, the default wavefront no longer yields concurrent start. In Figure 22 we illustrate for $s_0 = 4, s_1 = 6$ that the default wavefront (red) is no longer parallel to the concurrent start hyperplane (black). Concurrent start is still possible with the non-default wavefront coefficients $\lambda_0 = 2, \lambda_1 = 3$, which yield the schedule $\{(t,i) \to (2\lfloor (t-i)/6 \rfloor + 3\lfloor (t+i)/4 \rfloor, \lfloor (t+i)/4 \rfloor)\}$. Unfortunately, a non-default wavefront causes a large loss in tile-level parallelism throughout the computation. This effect is illustrated by the yellow wavefront in Figure 22, which is parallel to the concurrent start hyperplane (black). Next we analyze a kernel with asymmetric dependences:

```
for t:
  for i:
    A[t+1][i] = A[t][i-1] + A[t][i+2]
```

Pluto derives from this kernel the transformation $\{(t,i) \to (t-i, 2t+i)\}$. This transformation combined with square tiling and the default wavefront coefficients allows concurrent start as shown in Figure 23 for $s_0 = s_1 = 4$. The reason for this, possibly surprising, result is that for a 2 dimensional stencil (1 space, 1 time) with dependence distance 1 in the time direction, the coefficient of the space dimension in the normal will always be ± 1. This ensures that when adding the two hyperplanes together their coefficients for the space dimension cancel out and we get again the concurrent start hyperplane. The default wavefront coefficients combined with square tile sizes therefore yield a concurrent start wavefront. As already found earlier, non-square tile sizes will prevent concurrent start with these coefficients.

Another interesting observation is that even though the rational tile shapes in Figure 23 are identical throughout the original iteration space, the set of contained integer points is not. The reason for this difference is that even though we use integral tile sizes in the transformed space, the borders may become non-integral in the original space. Varying integer point placements between tiles can cause problems due to additional conditions in the generated code. As a

3 Tested with version `0.10.0-50-g1a4ac17` from `git://repo.or.cz/pluto.git`

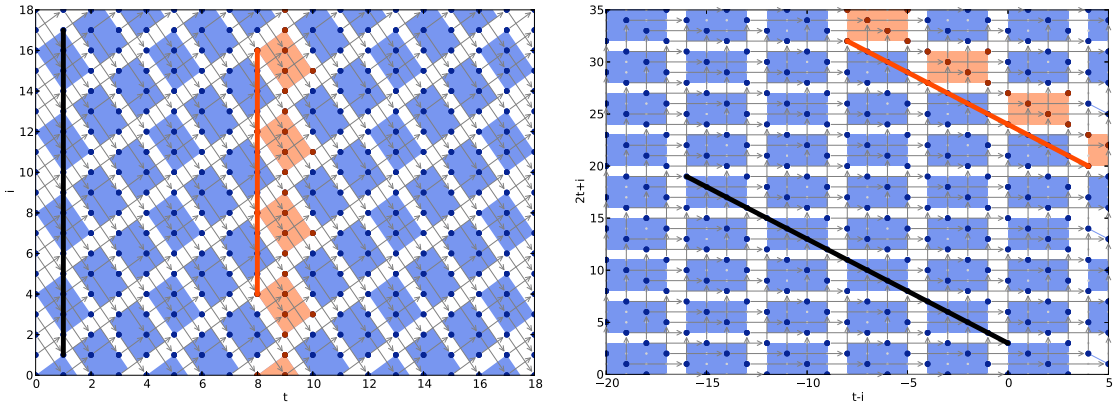


Figure 23: Asymmetric dependences & square tiling (original/transformed)

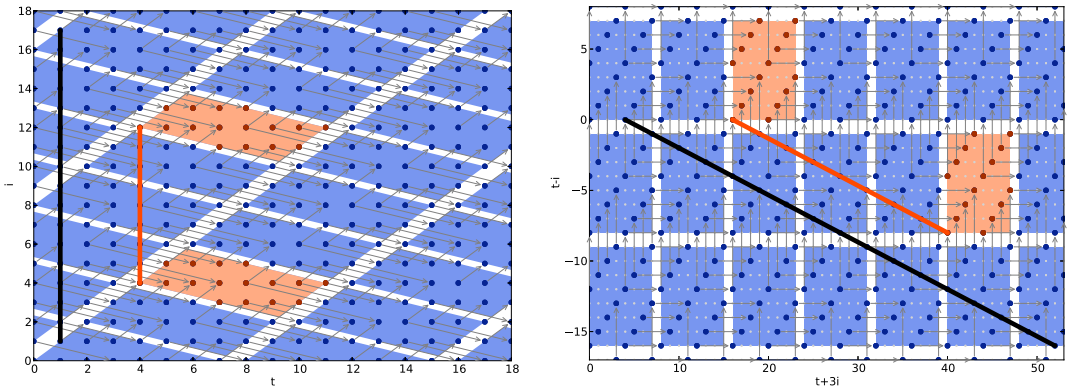


Figure 24: Multiple time steps. Square tiles reduce parallelism. (original/transformed)

next step we consider a case of dependence distances with different lengths on the time dimension.

```
for t:
  for i:
    A[t+1][i] = A[t][i-1] + A[t-2][i+1]
```

For this kernel, the Pluto implementation derives the transformation $\{(t,i) \to (t-i, t+i)\}$. The same transformation was already chosen for the example illustrated in Figure 21. Even though differences in the implementation of Pluto and the published algorithm may be possible, according to our understanding of the cost function in Pluto, this is in fact the transformation that the algorithm of [21] would choose. The resulting tiling yields 8 computations for a per-tile memory footprint of 3.

Another valid diamond tiling transformation is $\{(t,i) \to (t+3i, t-i)\}$. The hyperplanes in this transformation are the ones hybrid hexagonal/parallelogram tiling would read off directly from the dependence cone. Given a different cost function, Pluto may also choose this transformation. The interesting point here is, that the normal of the concurrent start hyperplane in the transformed space is not any-

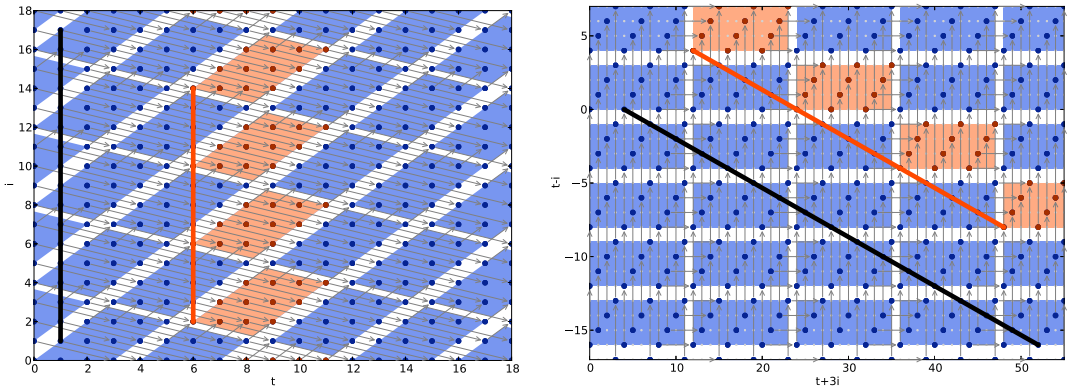


Figure 25: Multiple time steps. Non-square tiles maximize parallelism. (orig./trans.)

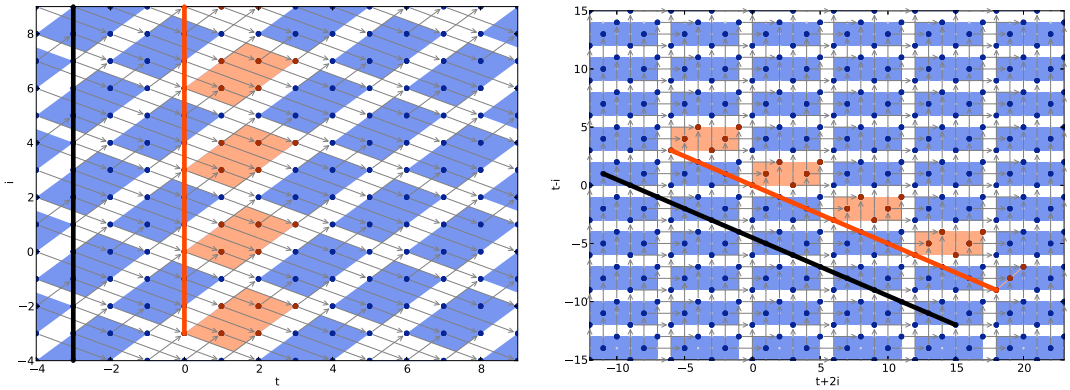


Figure 26: Diamond tiling (original/transformed)

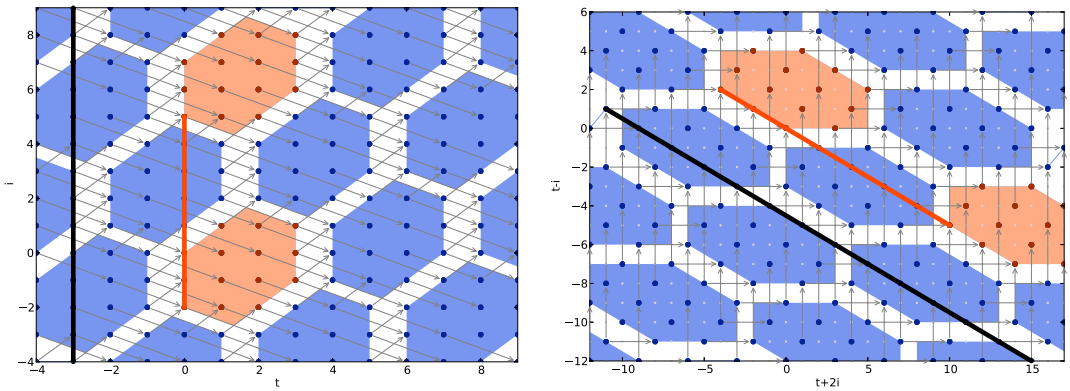


Figure 27: Hexagonal-tiling (original/transformed)

more (1,1), but rather (1,3). In this case, the standard square tiling illustrated in Figure 24 only yields concurrent start if, instead of the default wavefront coefficients, $\lambda_0 = 1, \lambda_1 = 3$ are chosen. As shown earlier, this severely reduces tile-level parallelism. On the other hand, for the same memory footprint as before, this tiling executes 16 computations.

We can restore concurrent start with the default wavefront by using non-square tile sizes. Figure 25 shows a non-square tiling ($s_0 = 12, s_1 = 4$) which enables concurrent start, has maximal tile-level parallelism and reaches 12 computations for a memory footprint of three. We therefore prefer this tiling over the previous two.

6.1.4 *Optimal tiles with default wavefront*

As seen in the previous section, the use of the default wavefront coefficients is necessary to ensure high tile-density. However, by itself this choice guarantees neither concurrent start nor a shared integer point placement for all tiles. As those properties are important, we present the conditions under which they can be reached.

We first explore the integer point placement. Forming the rows of matrix H from the tiling hyperplane normals $\vec{h}_i$, then tile sizes that are multiples of the determinant of H will ensure that all tiles have the same configuration of integer points since $\det(H) \cdot H^{-1}$ is an integer matrix. For example, the hyperplanes used in Figure 23 yield

$$H = \begin{pmatrix} 1 & -1 \\ 2 & 1 \end{pmatrix}$$

with $\det(H) = 3$. As $s_0 = s_1 = 4$ are not multiples of 3, the tiles may differ in integer point placement, as illustrated in the figure. On the other hand, tile sizes $s_0 = s_1 = 3$ would ensure a uniform integer placement across all tiles. The above condition is sufficient independently of the chosen wavefront schedule.

Next, we investigate the conditions on tile sizes to ensure concurrent start with the default default wavefront coefficients. Let $h_{x,0}$ be the first component of $\vec{h}_x$ and $h_{x,1}$ the second. The default wavefront then is $\lfloor (h_{0,0}t + h_{0,1}i)/s_0 \rfloor + \lfloor (h_{1,0}t + h_{1,1}i)/s_1 \rfloor$. Now, to achieve concurrent start, we need to ensure that the default wavefront schedule only depends on the time dimension t and that all space dimensions (i.e., i) are eliminated. This is true under the condition $s_0/|h_{0,1}| = s_1/|h_{1,1}|$. Note that the wavefront may still depend on the fractional part of the space dimension, but this only results in a variation within a fixed range, independently of the size of the domain. We can see that in Figure 21, where we reach concurrent start for the default wavefront, this condition holds with $4/1 = 4/1$. On the other hand, when changing the tile sizes to $s_0 = 4$ and $s_1 = 6$ as in Figure 22, the previous condition turns into $4/1 = 6/1$ and concurrent

start is not possible with the default wavefront. The above shows that to obtain concurrent start the two tile sizes cannot be chosen independently, but need to be scaled together. To make this more clear we introduce a new variable s which can be chosen freely and which is then used to define $s_0 = s|h_{0,1}|$ and $s_1 = s|h_{1,1}|$ such that concurrent start is obtained. Interestingly, this condition of a single parameter defining the tile size is exactly what is required by the parametric tiling approach of Iooss et. al [73].

6.2 UNIFIED DIAMOND AND HEXAGONAL TILING

In this section we present for a two-dimensional iteration space an extended formulation of diamond tiling which allows the creation of hexagonal tiles. The hexagonal tiles calculated are similar to those presented in Chapter 5, but are not identical in shape.

To obtain such a schedule we start from the diamond tiling approach, which means we first calculate a set of tiling hyperplanes, transform the index space with these hyperplanes and then apply rectangular tiling in the transformed space. We then (optionally) transform the rectangular tiling by "stretching" the rectangular tiles along the concurrent start hyperplane. The stretched rectangular tiles in the transformed space form hexagonal tiles in the original space. As a result we have a single schedule that describes diamond tiling, if tiles are stretched by a vector of length zero, and hexagonal tiling, if they are stretched by a non-zero-length vector.

In the following description, we assume that the tiling hyperplanes h_0, h_1 are computed by the diamond tiling algorithm as described in [21]. We focus on the description of the (possibly) stretched tiling scheme in the transformed space. As input for the stretched tiling scheme, we take the tile sizes s_0, s_1 as well as a vector $\vec{v} = (v_0, v_1)$, which is parallel to the concurrent start hyperplane (in the transformed space). We also require this hyperplane to have a normal $\vec{n} = (n_0, n_1)$ that is strictly positive in all components, as guaranteed by the algorithm of [21].

We first model diamond tiling using a standard 2D rectangular tiling in the transformed space. In this tiling the symbols s_0, s_1 define the tile sizes along the dimensions d_0, d_1 while T_0, T_1 are the resulting tile schedule dimensions (we ignore the point schedule dimensions, as this mapping is not relevant to this discussion). The following map describes such a rectangular tiling.

$$\{(d_0, d_1) \to (T_0, T_1) \mid \\ s_0 T_0 \leqslant d_0 < s_0(T_0 + 1) \wedge s_1 T_1 \leqslant d_1 < s_1(T_1 + 1)\} \tag{22}$$

Our goal is to achieve and maintain concurrent start using the default wavefront. Consequently s_0 and s_1 cannot be chosen freely (see Section 6.1.4). We require the user to choose tile sizes that ensure con-

current start. Figure 26 illustrates the above rectangular tiling using the transformation $\{(t,i) \rightarrow (t+2i, t-i)\}$, as well as the tile sizes $s_0 = 6, s_1 = 3$. The red tiles show the concurrent start wavefront.

Starting from this rectangular tiling we want to stretch the contained tiles by a vector $\vec{v}$ with components v_0, v_1, where $\vec{v}$ is parallel to the concurrent start hyperplane. More formally, we want to compute for each tile represented by T the set of points contained in a new tile T′. T′ is defined as the Minkowsky sum $T + V$, where $V = \{t\vec{v} \mid 0 \leqslant t \leqslant 1\}$ and the Minkowsky sum of A and B is defined as $A + B = \{\vec{a} + \vec{b} \mid \vec{a} \in A,\ \vec{b} \in B\}$. In addition we translate the new tiles to again reach a space filling tiling. In principle, $\vec{v}$ can have either of two possible directions, but to simplify the schedule formulation we choose $\vec{v}$ such that $v_0 < 0 \wedge v_1 > 0$. Figure 27 shows a stretching as we obtain it for $\vec{v} = (-4, 2)$ and $\vec{n} = (1, 2)$.

Before we implement the actual stretching, we first add two additional constraints to each tile. The first one bounds each tile at its lexicographic minimal point with the concurrent start hyperplane, the second one bounds each tile at its lexicographic maximal point with the same (but translated) hyperplane. We implement the lower boundary by placing the hyperplane at the origin and by offsetting it for each tile according to the tile sizes. To offset the tile along d_0 we adjust the right hand side of the lower bound by $n_0 s_0 T_0$ and $n_1 s_1 T_1$. The upper boundary is implemented by reversing the lower hyperplane. The location of the upper hyperplanes for tile (T_0, T_1) is the origin of tile $(T_0 + 1, T_1 + 1)$.

$$
\begin{aligned}
&\{(d_0, d_1) \rightarrow (T_0, T_1) \mid \qquad (23)\\
&\qquad s_0 T_0 \leqslant d_0 < s_0(T_0+1) \quad \wedge\\
&\qquad s_1 T_1 \leqslant d_1 < s_1(T_1+1) \quad \wedge\\
&\qquad n_0 s_0 T_0 + n_1 s_1 T_1 \leqslant n_0 d_0 + n_1 d_1 \quad \wedge\\
&\qquad n_0 d_0 + n_1 d_1 < n_0 s_0(T_0+1) + n_1 s_1(T_1+1)\}
\end{aligned}
$$

As a last step, we now stretch the tiles along $\vec{v}$. This requires us to increase the size of the rectangular tiles by v_0 in the d_0 dimension and v_1 in the d_1 dimension. We also account for the shifted positions of the rectangular tiles by adding some offsets o_0, o_1 to the upper and lower tile boundaries that will be derived later in this section. Finally we adjust the locations of the concurrent start planes by using $c_0 = n_1(s_0 + v_0) + n_0 v_1$ and $c_1 = n_1(s_1 + v_1) + n_0 v_0$.

$$
\begin{aligned}
&\{(d_0, d_1) \rightarrow (T_0, T_1) \mid \qquad (24)\\
&\qquad \exists o_0 = -v_0 T_0 + v_0 T_1, o_1 = -v_1 T_0 + v_1 T_1 :\\
&\qquad s_0 T_0 + o_0 + v_0 \leqslant d_0 < s_0(T_0+1) + o_0 \quad \wedge\\
&\qquad s_1 T_1 + o_1 \leqslant d_1 < s_1(T_1+1) + v_1 + o_1 \quad \wedge\\
&\qquad c_0 T_0 + c_1 T_1 \leqslant n_0 d_0 + n_1 d_1 \quad \wedge\\
&\qquad n_0 d_0 + n_1 d_1 < c_0(T_0+1) + c_1(T_1+1)\}
\end{aligned}
$$

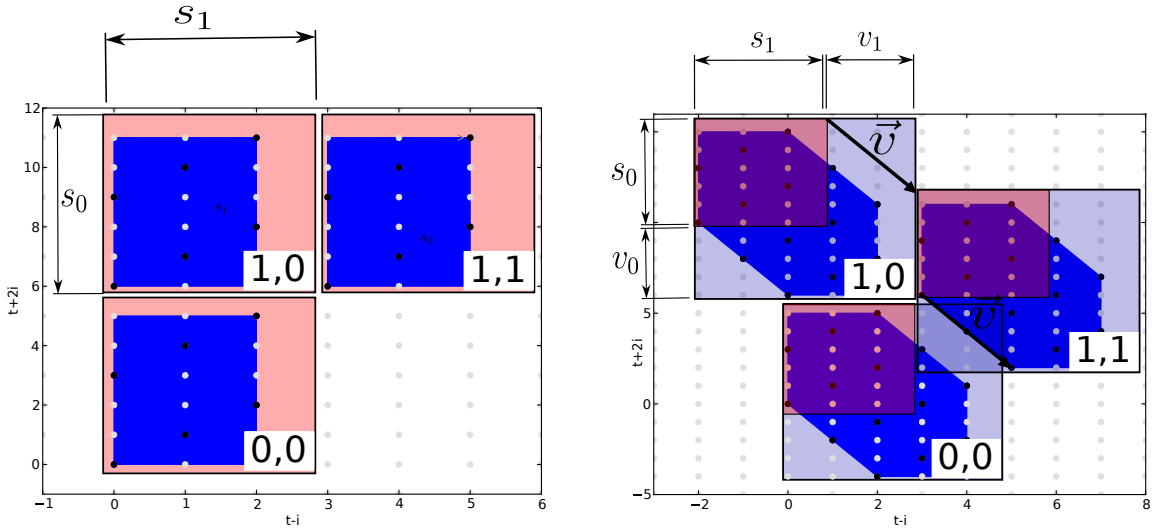


Figure 28: The stretching in the transformed space (unstretched/stretched)

Figure 28 illustrates the last step in detail. On the left side, the red tiles are the original square tiles (0,0), (1,0) and (1,1), each of size 6×4. On the right side, the same tiles have been stretched along $\vec{v}$. The rectangular tile shapes have been extended by 4 along d_0 and by 2 along d_1 resulting in the light blue tile shapes (the dark blue tile shapes illustrate the contained integer points). We can also see that the position of the red tile shape of tile (0,0) has not moved. However, when going one step up to tile (1,0) which means increasing the tile number T_0 by one, we offset the tile by $-v_0$ along d_0 as well as $-v_1$ along d_1. Similarly, when going from tile (1,0) to tile (1,1) which means increasing the tile number T_1 by one, we offset the tile by v_0 along d_0 and v_1 along d_1. Combined this yields the offset $o_0 = -v_0T_0 + v_0T_1$ for d_0 and $o_1 = -v_1T_0 + v_1T_1$ for d_1. The new values c_0 and c_1 do now also take into account the offset of the plane. When varying T_0 we now do not only need to take the vertical tile size s_0 into account, but in addition we include the additional vertical offset v_0 as well as the changed horizontal offset v_1. To support concurrent start hyperplanes of different orientations such offsets are scaled by the relevant components of $\vec{n}$. The corresponding changes have been added when adjusting c_1.

A very important observation is that tiles (T_0, T_1) as well as $(T_0 + 1, T_1 + 1)$ have overlapping rectangular parts. However, the concurrent start hyperplanes added at the position of $\vec{v}$ ensure that tiles are non-overlapping and still tile the full space. Also, as stretching and translation are carried along the concurrent start hyperplane, no dependences are violated. Finally, if the previous tiling had concurrent start, stretching along the concurrent start hyperplane preserves this property.

6.3 TILE SIZES THAT MAXIMIZE COMPUTE/COMMUNICATION

In this section we analyze how to maximize the compute to communication ratio and show how the tiling strategy choice affects both compute to communication ratio and synchronization overhead. Using a simple theoretical compute model, we derive for diamond and hexagonal tiles basic characteristics such as the number of operations executed, the amount of communication, the usage of local memory as well as the amount of synchronization needed for parallelism. We use these characteristics to understand the effectiveness of the different tile shapes. We use the following 3-point heat stencil as illustrative compute pattern.

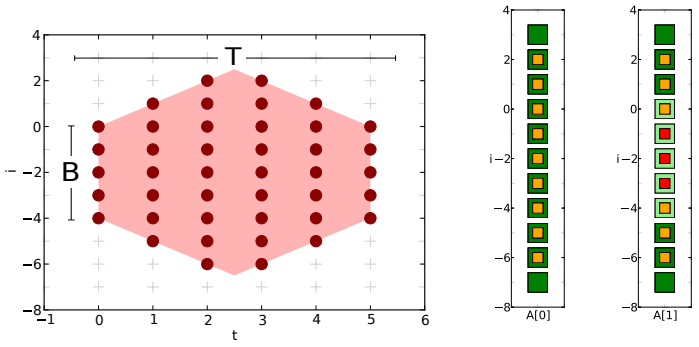
```
for t:
  for i:
S:    A[(t+1)%2][i] =  A[t%2][i-1] + A[t%2][i] + A[t%2][i+1];
```

For this analysis the iteration space boundaries are not of relevance, but we obtain the dependences $D = \{ S(t,i) \to S(t+1,i') \mid i'-1 \leqslant i \leqslant i'+1\}$ (transitively covered dependences removed), which are for this specific example code identical to the set of flow dependences ($D_F = D$). Furthermore, we obtain mappings $A_R = \{S(t,i) \to A(t \bmod 2, i') \mid i-1 \leqslant i' \leqslant i+1\}$ and $A_W = \{S(t,i) \to A((t+1) \bmod 2, i)\}$ specifying for each statement instance the data locations read from and written to. From this information we compute, e.g., using Pluto, the concurrent start tiling hyperplanes $t+i$ and $t-i$. However, instead of using now the full unified schedule (24) to tile the space, we derive a description of the set of iterations that belong to the single tile that will be placed at the origin. We use this tile as our tile shape model. To obtain its description we take the unified schedule from (24) and extract the set of input values that correspond to $T_0 = T_1 = 0$:

$$\begin{aligned}\{(d_0,d_1) \mid\ & v_0 \leqslant d_0 < s_0 \\ & \wedge\, o_1 \leqslant d_1 < s_1 + v_1 \\ & \wedge\, 0 \leqslant n_0 d_0 + n_1 d_1 \\ & \wedge\, n_0 d_0 + n_1 d_1 < n_1(s_0+v_0) + n_0 v_1 + n_1(s_1+v_1) + n_0 v_0\}\end{aligned} \tag{25}$$

We then set $(n_0,n_1) = (1,1)$ according to the tiling hyperplanes we computed for our example and we set $(s_0,s_1) = (T,T)$ and $(v_0,v_1) = (-B,B)$ to introduce two variables T and B that control the size of the tile.

$$\begin{aligned}\{(d_0,d_1) \mid\ & -B \leqslant d_0 < T \wedge 0 \leqslant d_1 < T+B \\ & \wedge 0 \leqslant d_0+d_1 \wedge d_0+d_1 < 2T\}\end{aligned} \tag{26}$$



(a) Iteration space (b) Accessed data

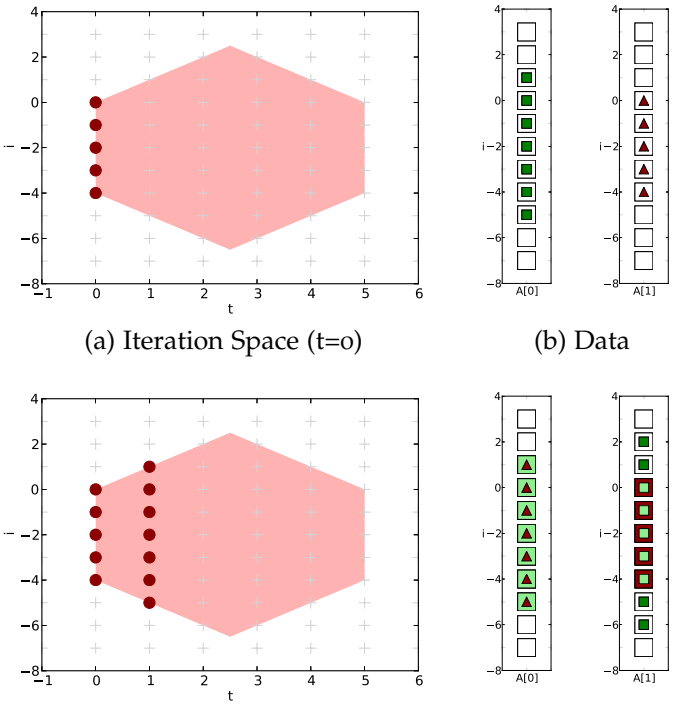
Figure 29: 1D hexagonal tiling ($T = 6, B = 4$)

As a final step, we use our tiling hyperplanes to translate this tile shape description back into the original space.

$$I = \{S(t,i) \mid -B \leqslant (t+i) < T \wedge 0 \leqslant (t-i) < T+B \\ \wedge\ 0 \leqslant (t+i)+(t-i) \wedge (t+i)+(t-i) < 2T\} \tag{27}$$

The result is I, a parametric tile shape description. It can be expressed with affine constraints despite the fact that a parametric description of all tiles can not. The tile shape we obtained has the form of a hexagon. In the illustration in Figure 29a we can see the two parameters that define its size: T defines the width of the tile along the time dimension t, and B defines the distance by which the tile is extended along the space dimension i. When $B = 0$ the tile shape degenerates to a diamond shape.

We now present the data access model we use for the computations in this tile. Figure 30 illustrates the first steps. At the first time step ($t = 0$), five elements are computed. Seven elements are read from array A[0] (dark small squares) and the resulting five elements are written into the scratchpad (dark small triangles). At the second time step ($t = 1$) seven elements are computed. We already have seven elements in our scratchpad that have been previously read (light large squares) and five elements that have been previously computed (dark large squares). For the computation we need to read nine elements from array A[1] and seven elements are stored back to the scratchpad. From the nine elements read five can be read from the scratchpad (light small squares). Similarly, when computing the nine new elements of the third time step ($t = 2$), from the eleven elements read, seven have been computed in a previous time step. Going further in time we see that with exception of $t = 3$ all time steps in the decreasing phase ($t = 4$ and $t = 5$) only read data that has been computed in previous time steps. Combining the memory accesses we can derive the set of data elements read (large squares) and the set of data elements written (small squares) in a tile. For the data elements read,



(a) Iteration Space (t=0) (b) Data

(c) Iteration Space (t=1) (d) Data

Figure 30: The first two time steps of 1D hexagonal tiling ($T = 6, B = 4$)

we can distinguish between the data elements that need to be read at least once from external memory (dark large square) and those that can be read directly from the local scratchpad (light large square). Similarly, for the data elements written we distinguish between the data elements that will be used by later tiles (light small square) and the ones that are only needed in this very tile, but do not need to be written out (dark small square). This is illustrated in Figure 29b.

We compute for each tile the following characteristics: The number of compute operations (O), the number of data locations read (R_{ALL}), the number of data locations read not considering data previously computed in the same tile ($R_{REDUCED}$), the number of data locations written to (W_{ALL}), the number of data locations written to and needed by later computations ($W_{REDUCED}$), the number of synchronization steps (S), as well as the footprint (F), i.e., the set of all data locations accessed. As these characteristics correspond to the number of integer points in certain sets, we can derive parametric closed form expressions by using barvinok [132] to count these points. We perform the following computations: $O = |I|$, $R_{ALL} = |A_R(I)|$, $W_{ALL} = |A_W(I)|$, $F = |A_R(I) \cup A_W(I)|$. There are only two slightly more complicated computations. First, $R_{REDUCED} = |A_W(D_F^{-1}(I) \setminus I)|$, the set of data locations that are read from within the tile without any statement in the tile having written to them previously. We compute this as the set of data locations written from statements that are at the origin of a

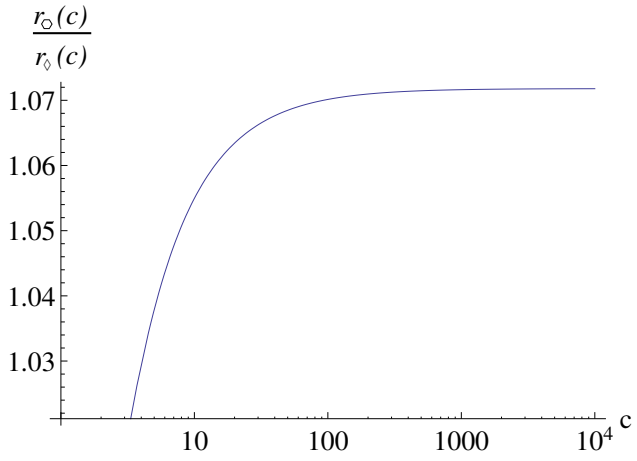


Figure 31: Compute-to-read ratio - Hexagonal vs. diamond tiling

flow dependence that ends in the tile, but are themselves not within the tile. Second, $W_{\text{REDUCED}} = |A_W(D_F^{-1}(U \setminus I) \cap I)|$, the set of data locations written to inside the tile and later read by a statement instance outside of the tile without being overwritten in between with data computed after the execution of the tile finished. We compute this starting from the universal set from which we remove the statement instances inside the tile to obtain the instances that are not in this tile. By applying the reverse flow dependences and intersecting with the tile again, we obtain the set of instances in the tile that write values that are used outside of the tile. We get the corresponding data locations by applying A_W.

The results[4] we obtain from barvinok are the following formulas:

$$\begin{aligned} O(T,B) &= \frac{1}{2}T^2 + TB\,, & S(T) &= T-1 \\ R_{\text{ALL}}(T,B) &= 2T+2B+2\,, & W_{\text{ALL}}(T,B) &= 2T+2B-2 \\ R_{\text{REDUCED}}(T,B) &= 2T+B+1\,, & W_{\text{REDUCED}}(T,B) &= 2T+B-1 \\ F(T,B) &= 2T+2B+2 \end{aligned}$$

As a next step we compute certain properties. We start with the compute-per-read ratio $r(T,B) = O(T,B)/R_{\text{ALL}}(T,B)$ and maximize it for a fixed "cache size" c, i.e., the number of data elements that fit in the scratchpad. For this we formulate an optimization problem and use the Mathematica [72] to solve it symbolically. The result is $\max_{T,B} r(T,B)\big|_{F\leqslant c} = \frac{(c-2)^2}{8c}$ and $\arg\max_{T,B} r(T,B)\Big|_{F\leqslant c} = (1/2(c-2), 0)$. We make two observations here: first, $r(c)$ increases linearly with the available cache size; second, the optimal tile shape has $B = 0$. This means it degenerates to a diamond.

When maximizing the ratio $r'(T,B) = O(T,B)/R_{\text{REDUCED}}(T,B)$ we obtain: $r'_\circ(c) = \max_{T,B:F(T,B)\leqslant c} r'(T,B) = c - \frac{1}{2}\sqrt{c(3c-4)} - 1$ at $T_\circ =$

4 The formulas have been simplified under the assumption $T \bmod 2 = B \bmod 2 = 0$.

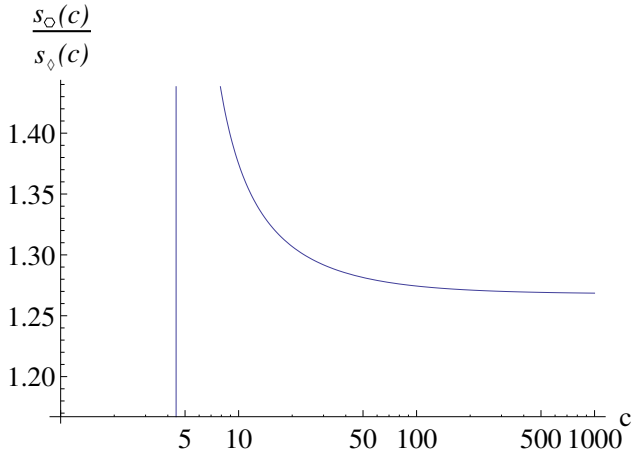


Figure 32: Compute-to-sync ratio - Hexagonal vs. diamond tiling

$\frac{1}{2}\left(\sqrt{21c^2-4\left(3\sqrt{c(3c-4)}+7\right)c+8\sqrt{c(3c-4)}+4}+2c-\sqrt{c(3c-4)}-2\right)$ and $B_\bigcirc = c-\frac{1}{2}\sqrt{c(3c-4)}-1$. We see that the optimal ratio is obtained with $B \neq 0$, a hexagonal tile shape not degenerated to a diamond. We now derive the optimal ratio with the additional constraint $B = 0$, which limits our search to diamond shaped tiles. We obtain $r'_\diamond(c) = \max_{T:F(T,0)\leqslant c} r'(T,0) = \frac{(c-2)^2}{8c}$ at $T_\diamond = 1/2(c-2)$. We can see that both $r'_\bigcirc$ and $r'_\diamond$ increase linearly with c, showing that hexagonal tiles are more efficient than diamond tiles. To understand how much more efficient they are, Figure 31 captures the evolution of $r'_\bigcirc/r'_\diamond$, making clear that as soon as the scratchpad can hold more than about 100 data elements, hexagonal tiles yield a ratio O/R_{REDUCED} that is more than 7% higher than diamond tiles. As W_{ALL} and W_{REDUCED} only differ by a small constant from the corresponding read properties, for sufficiently large scratchpads ratios similar to the ones we computed for the read properties will be obtained and similar conclusions can be taken.

Another interesting property is the amount of synchronization steps necessary per tile. For this we compute $s_\bigcirc = O(T_\bigcirc, B_\bigcirc)/S(T_\bigcirc)$ and $s_\diamond = O(T_\diamond, B_\diamond)/S(T_\diamond)$ which give an idea of how much computation can be performed for a certain number of synchronization steps. Figure 32 illustrates $s_\bigcirc/s_\diamond$ to compare the hexagonal and diamond tiling strategies. We see that the graph quickly converges to a value above 26%. This means 26% more computations can be executed for the same amount of synchronization when using hexagonal tiling. As these results are computed on a model, they obviously do not directly translate to a specific piece of hardware, but they show that for hardware where in-tile synchronization is costly hexagonal tiling can result in significant improvements.

6.4 SUMMARY

We presented a formulation of hexagonal tiling that combines the benefits of diamond tiling and hybrid-hexagonal tiling.

For diamond tiling, we formalized conditions on tile sizes and wavefront coefficients to ensure concurrent start among tiles. We also formulated a condition to ensure the same integer point placement across all tiles. And most importantly, we extended the original diamond tiling algorithm to hexagonal tiles. Using a simple cost model, we performed a comparative analysis of the compute to communication ratio of diamond and hexagonal tiling schemes as a function of tile sizes and characterized the optimal aspect ratio for hexagonal tiles and the benefit over diamond tiles. The analyses and approaches developed in this chapter could serve as the basis for practical implementations that improve on the current state-of-the art in tiling stencil computations.

The techniques in this chapter have been collaboratively developed and an earlier version of the text in this chapter has been accepted for publication in [7].

7 EXPERIMENTAL RESULTS

After having extensively discussed different tiling strategies, we evaluate in the following sections the performance we obtain with both split-tiling and hybrid-hexagonal tiling schemes.

7.1 SPLIT-TILING

To evaluate the performance of our split-tiling implementation we compare against two compilers: (i) PPCG [135], a state-of-the-art GPU code generator; (and) (ii) `overtile`. Holewinski et al.'s [70] GPU stencil compiler. PPCG uses simple space tiling to generate parallelized GPU code whereas `overtile` uses overlapped tiling to exploit reuse along the time dimension.

We evaluate split tiling on a couple of different kernels consisting of jacobi-1d (3-point, 5-point, 7-point), jacobi-2d (5-point) as well as a 9-point poisson solver. The experiments are run on a NVIDIA GeForce GTX 470 desktop GPU as well as on a NVIDIA NVS 5200M mobile GPU. The performance is shown in GFLOPS and includes both the time of the computation itself as well as the data transfer to the GPU.

The results in Figure 33 and Figure 34 show that the split-tiled code is consistently faster than both PPCG and `overtile`. Especially for the one dimensional test cases, we can see an almost 4x improvement over PPCG and between 30% and 40% over `overtile`. For the 2D cases on the mobile GPU, the performance of the different tools is again closer. On the desktop GPU, on the other hand, the difference between split tiling as well as PPCG and `overtile` is again considerable.[1]

7.2 HYBRID-HEXAGONAL

To assess the effectiveness of the hexagonal tiling schemes described in Chapter 5 we compare in Section 7.2.1 hybrid hexagonal tiling with state-of-the-art tools, and we analyze in Section 7.2.2 the performance impact of the different optimization strategies.

7.2.1 *Comparison with state-of-the-art tools*

To evaluate our hybrid hexagonal/parallelogram tiling scheme we extend the set of tools used. Besides overtile [70] and PPCG [135] itself we compare also against Patus-0.1.3 [41] and Par4All-1.4.1 [18].

1 Unfortunately, we cannot report `overtile` numbers for the poisson solver, as nvcc 4 and 5 both crashed on the CUDA file that `overtile` produced.

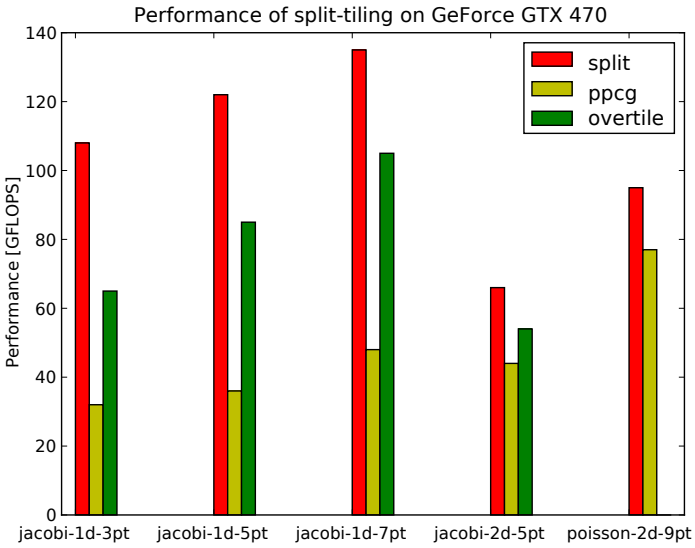


Figure 33: Split-tiling performance (desktop GPU)

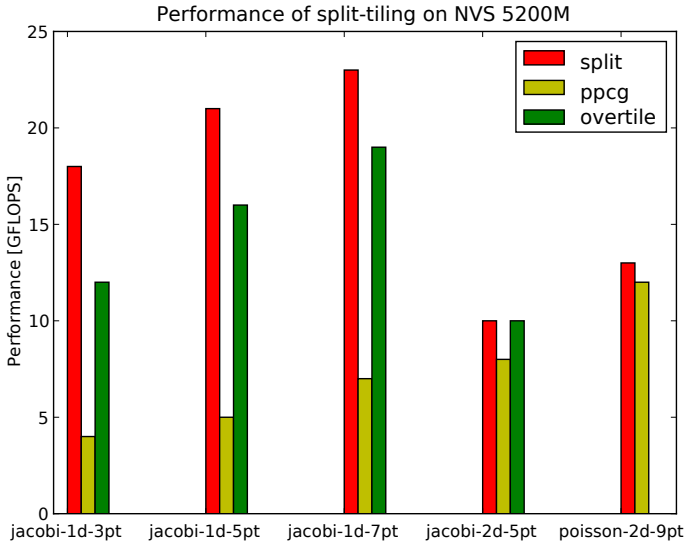


Figure 34: Split tiling performance (mobile GPU)

	laplacian 2D		heat 2D		gradient 2D		ftdt 2D	
PPCG	5.4		5.1		3.9		0.76	
Par4All	7.0	+30%	5.4	+2%	5.5	+41%	invalid CUDA	
Overtile	10.6	+96%	6.9	+35%	6.7	+72%	5.3	+597%
hybrid	**15.0**	**+177%**	**15.0**	**+194%**	**7.3**	**+87%**	**7.3**	**+860%**

(a) Two dimensional tests

	laplacian 3D		heat 3D		gradient 3D	
PPCG	2.0		1.8		2.1	
Par4All	2.0	±0%	1.9	+6%	3.1	+ 48%
Overtile	3.1	+55%	2.6	+44%	**3.6**	**+71%**
hybrid	**4.3**	**+115%**	**3.9**	**+116%**	**3.6**	**+71%**

(b) Three dimensional tests

Table 1: Performance on NVIDIA GTX 470: GStencils/second & Speedup

	laplacian 2D		heat 2D		gradient 2D		fdtd 2D	
PPCG	1.0		0.97		0.61		0.098	
Par4All	1.1	+10%	0.79	-18%	0.9	+55%	invalid CUDA	
Overtile	2.1	+90%	1.5	+54%	1.1	+80%	0.9	+818%
hybrid	**3.2**	**+211%**	**2.9**	**+198%**	**1.4**	**+130%**	**1.0**	**+920%**

(a) Two dimensional tests

	laplacian 3D		heat 3D		gradient 3D	
PPCG	0.32		0.29		0.32	
Par4All	0.34	+6%	0.35	+20%	0.69	+116%
Overtile	0.66	+106%	0.37	+30%	0.61	+90%
hybrid	**0.91**	**+184%**	**0.73**	**+150%**	**0.73**	**+128%**

(b) Three dimensional tests

Table 2: Performance on NVS 5200: GStencils/second & Speedup

We were not able to obtain a license for comparative evaluation with R-Stream [87].

	Loads	FLOPs/Stencil	Data-size	Steps
laplacian 2D	5	6	3072^2	512
heat 2D	9	9	3072^2	512
gradient 2D	5	15	3072^2	512
fdtd 2D	3	3	3072^2	512
	3	3	3072^2	512
	5	5	3072^2	512
laplacian 3D	7	8	384^3	128
heat 3D	27	27	384^3	128
gradient 3D	7	20	384^3	128

Table 3: Characteristics of Stencils

For benchmarks we use a Laplace kernel with two space dimensions, a 2D heat and a 2D gradient stencil as well as a two-dimensional, multi-stencil fdtd kernel. We also evaluate Laplace, heat and gradient kernels each having three space dimensions. Table 3 provides detailed characteristics of the stencils used. We did not evaluate our approach on one dimensional examples, because the hybrid method boils down to existing hexagonal or split tiling in this case [2]. All calculations were performed as single precision floating point computations and all timings include the data transfer overhead to and from the GPU. The experiments were conducted on NVIDIA GPUs: the NVS 5200M for mobile devices and a more powerful GeForce GTX 470.

For each tool, we sought to tune for the optimal tile sizes for the implemented tiling scheme and a specific benchmark. For PPCG, we used empirically optimized tile sizes used by the developers of the tool [135]. For Patus and overtile we used the provided autotuner. The Patus autotuner was run until completion, while we explored 800 tile sizes for each benchmark with overtile. For hybrid tiling we selected tile sizes aiming for a low load-to-compute ratio. Par4All was run with its dynamic tile sizing heuristic, using the options `-cuda -com-optimization` to enable GPU code generation. The flags defined in [135] were used for PPCG, and the hybrid tiling approach was combined with the optimizations discussed in Section 7.2.2. All other tools were used in the default configuration.

Table 1 and Table 2 show the results for the GTX 470 and NVS 5200, respectively. As a baseline, the general purpose compiler PPCG is able to create code for all benchmarks, but does not reach optimal speed. We do not include performance numbers for Patus, because due to its experimental CUDA support, only laplacian and heat 3D code could be generated. However, it should be noted that Patus reaches 3.5 GStencils/second for laplacian 3D on the GTX 470 and

0.50 GSTencils/second on the NVS5200, a 75% (56%) of speedup over PPCG. Except for some slowness on the heat-2D kernel, Par4All produces reasonably well performing code with good performance on the gradient 2D and 3D kernels. Par4All uses an internal heuristic to derive tile sizes. Overtile shows consistently good performance, attaining speedups over PPCG code of up to 96% for 2D kernels, very high speedups of up to 818% for fdtd 2D and up to 106% on 3D kernels. These results demonstrate the performance a stencil DSL compiler combined with auto-tuning can reach. Looking at the auto-tuned tile sizes we see that Overtile is not able to effectively exploit time tiling for 3D kernels. Instead, it falls back to a space-tiled version. This is also in line with Patus, Par4All and PPCG, which do not support time-tiling in general.

The last row presents results from our hexagonal-hybrid tiling compiler. For all 2D kernels, on both the GTX470 and the NVS 5200, we observe better performance than all previous techniques. Compared to base PPCG, we observe speedups ranging from 71% and 211%, with an exceptional 920% speedup for fdtd-2d. The consistently superior performance for 2D and 3D kernels across the board demonstrates the effectiveness of our approach. The 2D and 3D heat kernels showcase our hybrid-hexagonal tiling with performance results that are in three cases more than two times faster than the second best implementation.

One of the main reasons for the good performance is that we have been able to effectively exploit time-tiling for all benchmarks. Each 2D kernel executes eight time steps per tile and each 3D kernel executes four time steps per tile. Exploiting time tiling has only become beneficial due to the careful management of shared memory, as well as the reduction of overhead due to full-partial tile separation, code specialization and unrolling. Combined together, this enables excellent performance.

7.2.2 *Hybrid tiling and shared memory*

Even though hybrid tiling can be beneficial by itself, its full benefits only manifest when combined with explicitly managed shared memory. In this section, we analyze how shared memory usage as well as different shared memory optimizations impact the performance of a hybrid tiled kernel. As explicit cache management has proven to be especially challenging for 3D kernels, we choose to analyze the three dimensional heat kernel.

Table 4 gives an overview of the different configurations we analyzed and their performance on an NVS 5200 as well as a GTX 470 GPU. All configurations where run with 1x10x32 threads and hybrid tiles of size $h = 2, w_0 = 7, w_1 = 10, w_2 = 32$. As described in Section 5.2.5, tile sizes have been selected to minimize the load-to-

		NVS 5200		GTX 470	
(a)	no shared memory	8		39	
(b)	shared memory	8	±0%	44	+12%
(c)	(b) + interleave copy-out	11	+37%	65	+47%
(d)	(c) + align loads	12	+9%	70	+7%
(e)	(d) + value reuse (static)	11	-8%	73	+5%
(f)	(d) + value reuse (dynamic)	19	+58%	105	+50%

Table 4: Optimization steps: GFLOPS & Speedup

compute ratio and to ensure that the inner dimension is a multiple of the warp size.

Configuration (a) only uses global memory, but no shared memory. (b) uses shared memory. For each tile we first copy all required values into shared memory, we then perform the computation within shared memory and finally we copy the results back to global memory. (c) eliminates the explicit copy out phase. Instead, results are copied out as soon as they have been calculated. In (d) we adjust the position of the tiles in the data space such that all loads from global memory are aligned. Finally, (e) and (f) show two different approaches that both enable the reuse of values used and loaded in one tile and used in a subsequently executed tile. In (e) we eliminate the need to reload values by statically assigning each global value to a shared memory location. In (f) we allow a single global value to be dynamically placed for different tiles at different shared memory locations. To still enable reuse we add an explicit copy phase scheduled between two subsequent tiles. This phase moves values from their old shared memory location to the location where the next tile expects them to be.

To understand the performance results shown in Table 4 we analyze the different configurations together with relevant performance counters. The results are shown in Table 5, in units of 10^9. The first one, configuration (a) gives a solid performance baseline. Introducing explicit shared memory in (b) does not change performance on the NVS 5200 and gives a 12% performance increase on the GTX470. The small performance difference is not surprising. Even though the number of global load instructions is reduced by a factor of 20, the actual reads from DRAM are mostly unaffected. This shows that our shared memory management is as effective in avoiding DRAM loads as the automatic caches are. Looking at the L2 transactions we see large benefits due to our explicit shared memory management. Unfortunately, the almost unchanged performance suggests that other effects such as a reduced global load efficiency and the explicit cache management overhead itself hide the benefits. One cache management problem is the missing overlap of computation and data-transfers. (c) shows that that by overlapping copy-out and the actual computation, we can in-

crease performance by 37-47% without changing the amount of data transferred. Another inefficiency we see is the global load efficiency of only 30%. (d) partially addresses this by ensuring that all loads from global memory are fully aligned. However, only after removing partial global loads in (e) and (f) we are able to fully achieve 100% global load efficiency. Interestingly, at this point our kernel has been moved from being bound by global loads to being bound by shared memory loads. (f) has as efficient global loads as (e), but due to the way memory is accessed, it is very likely to cause bank conflicts in shared memory. This is reflected by the number of shared memory load transactions, which is twice that of all other kernels. The overhead caused by these bank conflicts unfortunately hides the gains from the reduction in global loads. On the other hand, (f) shows that we are able to create a highly performing kernel that achieves 100% global load efficiency, 100% shared load efficiency and that significantly reduces the requests that reach the L2 cache and global memory.

The overall speedup of 250% for this kernel was only possible due to the combination of hybrid-hexagonal tiling with careful shared memory management. Our optimization reaches a point where the kernel is mostly bound by shared memory. Further reducing the number of shared memory loads through register tiling would be an interesting angle to increase performance even further.

	gld_inst_32bit	dram_read_transactions	l2_read_transactions	shared loads per request	gld_efficiency
(a)	171.0	1.7	12.0	n/a	54%
(b)	8.7	1.8	1.4	1.0	30%
(c)	8.7	1.8	1.4	1.0	30%
(d)	8.8	1.0	0.95	1.0	56%
(e)	7.6	0.97	0.49	1.8	100.00%
(f)	7.6	0.95	0.48	1.0	100.00%

Table 5: Performance counters (units of 10^9 events)

7.3 SUMMARY

We have shown in our experimental evaluation on split tiling that the integration of domain specific optimizations in general purpose compilers can yield notable speedups. With split tiling we reach those speedups not only over PPCG, our general purpose compiler, but we

also outperform overtile, a specialized stencil compiler, on 1D kernels and show competitive performance on 2D kernels.

With hybrid-hexagonal tiling we have been able to significantly increase performance for both 2D and 3D kernels. Our tiling technique has not only been able to outperform general purpose compilers such as PPCG and Par4All, but we also showed good results over overtile, a domain specific stencil compiler especially written to generate fast CUDA code. Especially our experiments on 3D kernels have been interesting, as we showed that only when carefully addressing GPU specific concerns, the use shared memory can be made efficient enough to be profitable. As a result we have been able to exploit time tiling benefits even for 3D kernels.

Part IV

POLYHEDRAL BUILDING BLOCKS

8

THE CONCEPT

The concept of using modular components to build something bigger is well known even outside of computer science. Computer scientists themselves get to know this concept in many places, one of the first being the individual tools available on a Unix shell, where more complex operations such as `cat filename | sort | uniq` are created by composing smaller specialized tools. Similarly, optimizing compilers are often assembled from different building blocks which commonly include compiler front ends, a set of optimizers, as well as code generators for different targets.

In the context of high-level loop optimizations, the use of building blocks is rare. Instead compilers use compiler specific implementations or, possibly due to the high implementation costs, they omit high-level loop optimizations altogether (e.g., luajit, V8 - Google's Javascript Engine, Julia). As a result, high-level loop optimizations are only available in a low number of compilers, commonly those with a strong focus on high-performance computing. Even for these compilers, the set of high-level loop optimizations performed is mostly limited to classical loop optimizations. More complex transformations are commonly left to specialized domain specific compilers (e.g., Halide [108], Pochoir [121]). We agree that domain specific languages are a great way to provide a familiar user experience to domain experts and we also understand that in certain cases the implementation of a fully specialized domain specific compiler is the quickest path forward. However, the use of generic compiler infrastructure components can give big benefits. Taking Halide as an example, their use of the LLVM compiler infrastructure [86] for low-level optimizations, target code generation, scheduling and instruction selection enables them to generate highly optimized low-level code for various architectures (X86, ARM, NVIDIA PTX) without writing any target specific code. They can therefore focus on the development of a good domain specific language and the development of domain specific optimization strategies. Similarly, other domain specific compilers often target C/C++ code, instead of generating platform specific assembly. Even though slightly indirect, this strategy allows them to reuse existing general purpose compiler back ends.

We believe that decoupling high-level loop optimizations from both individual compilers as well as individual optimization strategies can show similar benefits, as the reuse of compiler back ends shows today. Not only could a reusable high-level loop optimization infrastructure reduce the amount of implementation work necessary on the side of

compiler developers, but the use of such an infrastructure could also enable them to perform optimizations too complex for a single DSL compiler or research project. However, beyond being a convenient infrastructure component, decoupled high-level loop optimizations can also facilitate the transfer of domain specific optimizations back into general purpose compilers. Especially in the context of loop optimizations, general purpose compilers are doing reasonably well in detecting and understanding certain compute patterns, such that highly specialized domain specific optimizations can be applied within such general purpose compilers. Even though such optimizations may be less intuitive than the use of a DSL compiler, being automatically available in widely used general purpose languages possibly in the system compilers of many important devices may be intriguing for various use cases, e.g., (performance) portable image processing computations on mobile devices.

The polyhedral model (Chapter 2), as developed within the last 25 years of compiler research, is a model for high-level loop optimizations that is entirely independent of specific compilers and which can be used to describe a large variety of loop transformations. When working with this model it is common to use a set of nicely decoupled building blocks. Such building blocks consist of different front ends that extract polyhedral descriptions from different input languages, different schedulers which perform general purpose or domain specific optimizations and AST generators which help to recover imperative program code. When developing tools that provide the different building blocks, the focus of existing research has most of the time been on source-to-source translators with Loopo [62], Pluto [35], PoCC/PolyOpt [103], Clan [28], Chill [40], but there has also been previous work on integrating such kinds of loop optimizations in general purpose compilers such as IBM/XL [36] and Open64/ORC [44], with us heavily contributing to the development of high level loop optimizations in gcc (Graphite [122]) and later LLVM (Polly [8]).

In this part of the thesis we present a set of polyhedral building blocks used in the development of the domain specific optimizations presented in Part iii. Implementing our optimizations such that domain specific transformations and general purpose infrastructure are carefully separated adds certain development overhead, but as a result we are now not only able to decouple our optimizations from specific compilers, but we also obtain a set of infrastructure building blocks that facilitate the development of new general purpose and domain specific loop optimizations. As these tools already today form the core of an optimization infrastructure shared between various research tools and open source compiler projects, we hope to contribute to an increasing portability of high-level loop optimizations. Several of the building blocks we present follow the tradition of earlier research prototypes, this time however with a strong focus on making

them as generic as possible and embeddable in low-level compilers as well as domain specific optimizers. We also put a strong focus on correctness and robustness in corner cases commonly ignored by research prototypes. Our hope is that this set of building blocks enables research to be both closer to production compilers and that it facilitates the transfer of research results to such compilers.

We present in Chapter 9 pet, a tool to extract a polyhedral representation from C code, which we used as the front end of our domain specific compiler. In Chapter 10 we introduce our new polyhedral AST generator, which is configurable to ensure its low-level optimization strategies complement and exploit our high-level optimizations and provides facilities for extensive specializations which enable us to generate high performance code without writing a specialized domain specific code generator. We finish this part of this thesis with Chapter 11 where we present the concept of polyhedral schedule trees as a more practical way to represent complex schedule descriptions as they may result from the GPU specific optimizations we perform.

9 POLYHEDRAL EXTRACTOR

To apply polyhedral high-level loop optimizations on a certain input program it is necessary to extract a polyhedral description (Section 2.2.3) of the input program. This description can be extracted in various ways with different approaches having certain advantages and drawbacks.

When applying polyhedral transformations in general purpose compilers such as IBM-XL [36], gcc (through graphite [100]) or LLVM/clang (through Polly [3]) the polyhedral model is extracted from a low to medium level IR. This makes the extraction independent of the input programming language and allows to leverage compiler internal canonicalization and analysis passes to increase the amount of code that can be analyzed. However, when analyzing a compiler internal IR, relating the analysis to the input program becomes difficult. Hence, direct user feedback about unsupported constructs is normally not available. Also, the generated code is in most cases again a compiler internal IR. Synthesizing a higher level language is hard and even if successful, relating the resulting high level code to the original program remains difficult. We conclude that analyzing a compiler internal IR may not be the best approach if user feedback is required or a high level language should be generated.

For some applications, including polyhedral source-to-source compilers such as Pluto [35] and PoCC,[1] it is therefore more convenient to extract a polyhedral model from the source language, typically C or Fortran, or, more commonly, a subset and/or mixture of these languages. The parsers for these tools usually only analyze those program fragments (SCoPs) that need to be converted to the polyhedral model. This means that information from outside of these SCoPs, such as the types of variables or the sizes of arrays, is not available in the output or has to be redeclared using special annotations. By far the most popular such parser is `clan` [28], which is used by both Pluto and PoCC.

For our goal, research on the generation of effective GPU code, none of the previous approaches is fully satisfactory. Working from within a compiler IR is interesting when planning to scale to a large number of diverse applications, but, in the context of our research, the absence of information about array shapes, the limited user feedback and the very indirect translation from source code to the polyhedral model, complicates the evaluation of new high-level optimizations. Existing source level parsers provide a well suited direct translation,

1 `http://pocc.sourceforge.net`

but their inability to extract array sizes defined outside of the SCoPs causes problems in the context GPU code generation. Furthermore, their rather limited support for the parsing of subscript expressions makes it hard to transform and specialize index expressions in a way we believe is necessary for the generation of efficient GPU code. Even though it would be theoretically possible to extend a parser such as `clan` to provide the information we need, we believe this will be a significant effort with most of the effort spent on improving an incomplete C parser.

Instead of developing a better C parser, we present a different approach. `pet`,[2] the library discussed in this chapter, uses a real C compiler to parse the input source code into a high-level AST from which it then directly extracts the polyhedral model. The use of a real C parser, in our case clang, has many advantages. In particular, clang has full support for C99 [75], including variable length arrays (VLAs). This support is crucial for properly parsing the dynamically sized array versions of the PolyBench 3.1 benchmarks.[3] Additionally, clang generates very nice diagnostics, allowing us to clearly communicate to the user which constructs (if any) in the input code are (currently) not supported by `pet`.

As the use of a real C compiler enables us to parse a large number of different language constructs, we are now faced with the question of how to translate them into a polyhedral model. As explained in Section 2.2.3 the work presented here is based on isl, an integer library that models integer sets and maps in their full generality. pet exploits this generality to model a wide range of C language constructs, many of which have not been modeled by existing tools either due to limitations in their parser or due to the use of a less general representation of integer sets. pet is for example able to parse interesting expressions such as `a % b` or the ternary expression `cond ? a : b`, both very useful for describing iterative stencil computations and boundary condition handling. Special care was also taken to ensure that `pet` can parse the code generated by both CLooG and our AST generator (Chapter 10).

Having more information about our input available also helps towards our goal of bringing polyhedral transformations closer to production compilers. As `pet` has access to the full type information of the C code, it can model its semantics more precisely. C has for example different semantics regarding the wrapping of signed and unsigned integers in the presence of overflow. Taking them into account is important as they are not only commonly exploited by production compilers to improve their optimizations, but they also prevent transformations that may break standard conforming programs that rely on certain overflow behavior. With `pet` we aim to model these precise

2 `http://pet.gforge.inria.fr/`

3 `http://www.cse.ohio-state.edu/~pouchet/software/polybench/`

semantics, such that tools based on pet are able to match the semantics of standard compliant compilers even in corner cases.

9.1 OVERVIEW

The input to `pet` is valid C source code. A polyhedral model will be constructed for a fragment of this C code. In its default mode of operation, the fragment to be analyzed needs to be delimited by pragmas, in particular `#pragma scop` and `#pragma endscop`. If any construct inside this fragment cannot be analyzed by `pet`, then a warning is produced pointing out the unsupported construct. If the user specifies the `--autodetect` option then `pet` will try to detect a fragment that fits inside the polyhedral model. In this case, no warnings will be produced by `pet` as any code containing unsupported constructs will be considered to lie outside of the extracted code fragment. Supported constructs include expression statements, **`if`** conditions (see Section 9.2.2), **`for`** loops (see Section 9.2.3) and piecewise quasi-affine index expressions (see Section 9.2.1).

The output that is generated for a SCoP (Section 2.2.2) consists of a context, a list of arrays and a list of statements. The context is a parameter set containing those parameters values for which the SCoP can be executed. Currently, the context is mainly used to ensure that all arrays have non-negative sizes. Following C99 6.7.5.2 `pet` can assume that a SCoP will only be executed with parameters that yield array sizes larger than zero. However, as both clang and gcc allow zero sized arrays in non pedantic mode, we also permit parameter values yielding arrays of size zero. For each array, we keep track of its "extent", its element type and (optionally) the set of possible values. The extent is a set defined in a space named after the array and containing all allowed array indices. In other words, the extent describes the size of the array. The set of possible values of an array may be specified by the user through a `#pragma value_bounds`.

Each statement consists of a line number, an iteration domain, (part of) a schedule and a parse tree of the corresponding statement in the input program. In this parse tree, each access is represented by an integer map mapping elements from the iteration domain to their corresponding index. Additionally, we keep track of whether the access is a read or a write (or both). The name of the iteration domain may be specified by a label on the statement. Otherwise, the name is generated to be of the form `S_i`. Each statement keeps track of its part of a global schedule. The entire global schedule corresponds to the original execution order.

As a trivial example, a dump of the representation of the program in Listing 3 extracted by `pet` is as follows.

```
context: '[N] -> {  : N >= -1 }'
arrays:
```

```
void foo(int N)
{
        int a[N + 1];
#pragma scop
        for (int i = 0; i <= N; ++i)
U:                a[i] += i;
#pragma endscop
}
```

Listing 3: A trivial program

```
- context: '[N] -> {  : N >= -1 }'
  extent: '[N] -> { a[i0] : i0 >= 0 and i0 <= N }'
  element_type: int
statements:
- line: 6
  domain: '[N] -> { U[i] : i >= 0 and i <= N }'
  schedule: '[N] -> { U[i] -> [0, i] }'
  body:
    type: binary
    operation: +=
    arguments:
    - type: access
      relation: '[N] -> { U[i] -> a[i] }'
      read: 1
      write: 1
    - type: access
      relation: '[N] -> { U[i] -> [i] }'
      read: 1
      write: 0
```

9.2 CONSTRUCTING A POLYHEDRAL REPRESENTATION

In this section we describe in detail how we construct a polyhedral representation (Section 2.2.3) that models a piece of C code. Specifically, we describe how we obtain access relations, iteration domains and schedules from **if** statements, **for** loops and accesses to C arrays. The discussion in this section is limited to the core polyhedral model and only applies to code where all relevant properties can be derived statically. However, within these bounds we try to be as generic as possible by exploiting the full generality of integer maps and sets as provided by isl. A set of interesting additions to our core translation are discussed in Section 9.3.

9.2.1 *Access relations*

An access relation maps an iteration vector to one or more array elements and is modeled with a named integer map (represented by an `isl_map`). This means that the array elements are described by affine constraints, possibly involving existentially quantified variables and parameters. Since the model is constructed bottom-up, the initially constructed access relations have a zero-dimensional domain and therefore do not involve any iterators. Instead, some of the initial parameters may be converted to iterators at a later stage.

The access relation is constructed by considering each index expression individually, one for each dimension of the accessed array. Each of these index expressions is first recursively constructed as an `isl_pw_aff`, i.e., a piecewise quasi-affine expression. These objects are then converted into maps and combined into a single map. Each index expression may involve integer constants, parameters and the following operators: `+`, `-` (both unary and binary), `*`, `/`, `%` and the ternary `?:` operator. The second argument of the `/` and the `%` operators is required to be a (positive) integer literal, while at least one of the arguments of the `*` operator is required to be a piecewise constant expression. For example, an index expression of the form `i * (i < 5 ? 2 : 1)` is allowed, while an expression of the form `(i > 10 ? i : 1) * (i < 5 ? i : 1)` is not, even though it is equivalent to `(i > 10 || i < 5) ? i : 1`. The first argument of the `?:` operator needs to satisfy the requirements of Section 9.2.2.

Each of the above operators has a corresponding operation in `isl` on `isl_pw_affs` and therefore requires no extra computations in `pet`. The only exceptions are the `/` and `%` operators. `isl` does not provide any operation that directly corresponds to the C integer division (which rounds to zero), but instead provides a "floor" (which rounds to negative infinity) and a "ceil" operation (which rounds to infinity). An expression of the form `a / b` is therefore constructed as `a >= 0 ? floord(a,b) : ceild(a,b)`, with `floord` and `ceild` functions that correspond to the floor and ceil operations applied to the quotient of the arguments. The `%` is treated in a similar way. For compatibility with `CLooG` (see Section 9.3.1), the functions `floord` and `ceild` are also accepted inside the SCoP, even if they are not explicitly defined in the input. Similarly, the functions `min` and `max` are also allowed.

Depending on how a statement accesses memory `pet` marks the array accesses as read-only, write-only or a combined read and write. In function calls it is also possible to pass (the address of) an entire array or array slice as a parameter, e.g., `f(A)` or `f(A[i])` in case of a two-dimensional array `A`. In such cases, the access relation is constructed to read/write the entire array (slice).

The input program may also contain affine expressions that are not used as index expressions, but that are instead used directly as arguments to functions calls or operators. Since a program transformation may change the iterators in these expressions, we also represent them as access relation. Since there is no array involved, the range of these relations is a nameless one-dimensional space. This space can be thought of as the set of integers, with element `i` having value `i`.

9.2.2 *Conditions*

A condition is modeled by an integer set (represented by an `isl_set`) containing those elements that satisfy the condition. The input expression may be any boolean expression involving the `&&`, `||` and `!` operators and comparisons. A comparison is an expression that applies one of `<`, `<=`, `>`, `>=`, `==` or `!=` to two affine expressions. Such an affine expression needs to satisfy the same requirements as the index expressions in Section 9.2.1. An affine expression `e` itself may also be used where a comparison is expected, in which case it is treated as the comparison `e != 0`. As before, each of the above operations has a direct counterpart in `isl`.

9.2.3 *Loops*

Currently, `pet` only supports **`for`** loops. The only exception is the infinite loop, which may be written as either **`for`**`(;;)` or **`while`**`(1)`. The contribution of an infinite loop to the iteration domain is of the form $\{ t \mid t \geqslant 0 \}$. Recall that the iteration domains are constructed bottom-up and that each enclosing loop prepends a dimension to the iteration domain.

Since `pet` does not yet perform any induction variable recognition, the induction variable needs to be explicitly available in the **`for`** loop. That is, the loop needs to be written such that it has the form **`for`**`(i = init(n); condition(n,i); i += v)`, where `n` is any number of parameters. In particular, the initialization part needs to assign an expression to a single variable (or initialize a single newly declared variable) and this same variable needs to be incremented by a (signed) constant in the increment part. The increment may also be written `i -= -v`, `i = i + v`, `++i` or `--i` (in case `v` is 1 or -1).

The condition may be any condition that satisfies the requirements of Section 9.2.2. Note in particular that this means that the condition may *not* involve any variables that are being written inside the loop body as they are not considered to be parameters. In principle, the condition does not need to involve the induction variable, but such a condition will result in either an empty or an infinite loop. Let us now assume that `v` is positive. A minor variation of the construction below is used when `v` is negative. The constraints on the loop iterator

imposed by the initialization and the stride can be expressed as $D = \{ i \mid \exists \alpha : \alpha \geqslant 0 \wedge i = \texttt{init}(\vec{n}) + \alpha v \}$. If v is equal to 1, this is simplified to $D = \{ i \mid i \geqslant \texttt{init}(\vec{n}) \}$. This simplification is performed by `pet` and does not have any effect on the constructed iteration domain, but it may result in a simpler *representation* of this iteration domain.

As to the condition of the **`for`** loop, a value of the iterator belongs to the iteration domain if the condition is satisfied by that value *and* all previous values. Define the set $C = \{ i \mid \texttt{condition}(\vec{n}, i) \}$. The contribution of the loop to the iteration domain is the set $\{ i \in D \mid \forall i' \in D : i' \leqslant i \implies i' \in C \}$, or, in other words, $\{ i \in D \mid \neg \exists i' \in D : i' \leqslant i \wedge i' \notin C \}$. In `isl`, we can compute this set as

$$D \setminus \left(\{ i' \to i \mid i' \leqslant i \} (D \setminus C) \right).$$

That is, we take the elements in D that do not satisfy the condition (C), map them to later iterations and subtract those later iterations from D. If `condition(n,i)` does not involve any lower bounds on `i` then any condition satisfied by `i` is also satisfied by earlier iterations and the above computation can be simplified to $D \cap C$. This simplification may again result in a simpler representation of the result.

9.2.4 *Schedule*

As explained before, each statement maintains its part of the global schedule. These parts of the schedule are constructed together with the iteration domains. The initial iteration domains are zero-dimensional and the schedule simply maps this domain to a nameless zero-dimensional space. If a statement appears in a sequence of statements, the schedule for these statements is extended with an initial constant range (i.e., schedule) dimension. The values of these dimensions correspond to the order of the statements in the sequence. If a statement appears as the body of a loop, then the schedule is extended with both an initial domain dimension and an initial range dimension. If the increment on the loop is positive then these new dimensions are equated. Otherwise they are made to be opposite. That is, the schedule is extended with either $\{ i \to i \}$ or $\{ i \to -i \}$.

9.3 ADDITIONAL FEATURES

Besides the basic constructs described above, `pet` also supports some additional features. Data dependent constructs have been used in Verdoolaege's work on equivalence checking [134] and the construction of dynamic polyhedral process networks. As they are not relevant to the content of this thesis, we refer the reader to [10, Section 4.1-4.3]. Features to parse `CLooG` and isl AST generator output are helpful e.g., to reanalyze the generated code. As we strive to support the use of pet also in (semi)automatic modes on possibly preexisting source

code, it is important to correctly model C99 even in corner cases. One interesting corner case we handle successfully are unsigned integers and their wrapping semantics in case of overflow.

9.3.1 *CLooG specific features*

```
for (c1=ceild(n,3);c1<=floord(2*n,3);c1++) {
  for (c2=0;c2<=n-1;c2++) {
    for (j=max(1,3*c1-n);j<=min(n,3*c1-n+4);j++) {
      p = max(ceild(3*c1-j,3),ceild(n-2,3));
      if (p <= min(floord(n,3),floord(3*c1-j+2,3))) {
        S2(c2+1,j,0,p,c1-p);
      }
    }
  }
}
```

Listing 4: Part of `CLooG` output for `thomasset` test case

The output of `CLooG` may contain special functions and constructs that require special care. Most of these have been mentioned before, but here we provide further details. First of all, as explained in Section 9.2.1, the output may contain the "operators" `floord`, `ceild`, `min` and `max`. Although macro definitions can be provided for these operators, it is more efficient to recognize them directly inside `pet`. For example, a macro definition for `floord` would have to encode this operation in terms of integer divisions and would therefore have to introduce several cases, while there is no need for different cases if it is recognized directly. Similarly, if the condition of a for loop is of the form `i <= min(a,b)`, it can be directly encoded as $\{ i \mid i \leqslant a \wedge i \leqslant b \}$ instead of introducing cases depending on the difference between `a` and `b`. This is especially important if there are many nested `min`s or `max`s as in the `classen2` test case. Finally, the simple forward substitution of scalars discussed in Section 9.1 is also essential for parsing some `CLooG` outputs. Consider for example part of the output for the `thomasset` test case, reproduced in Listing 4. At first sight, the **`if`** statement looks like it involves a data-dependent condition, but by plugging in the expression assigned to `p` in the previous statement, it can be analyzed as a static affine condition.

9.3.2 *Support for unsigned integers*

In C99 signed and unsigned integer types do not only define different sets of values, but they also behave differently. Signed values yield undefined behavior if the result of an expression is not within the range of representable values (C99 6.5). Like other compilers pet can

and does assume that undefined behavior is never triggered by a valid program. Consequently it assumes that for signed types the results of all expressions fit in the corresponding type. This means pet can directly translate such expressions to `isl_pw_aff` expressions.

```
for (unsigned i = 0; i < n; i++)
        A[0] += i;

for (unsigned j = 0; j < n + 1; j++)
        B[0] += j;
```

Listing 5: Unsigned operation in loop bound

```
for (unsigned i = 0; i < n; i++) {
        A[0] += i;
        B[0] += i;
}

B[0] += n;
```

Listing 6: Invalid fusion of program in Listing 5

For unsigned types C99 6.2.5 includes the following exception: *"[..] a result that cannot be represented by the resulting unsigned integer type is reduced modulo the number that is one greater than the largest value that can be represented by the resulting type"*. The program in Figure 5 consists of two loops with n and $n+1$ iterations. If the loop bounds are represented as $i \leqslant n$ and $j \leqslant n+1$, fusing the two loops to the code in Figure 6 is possible. Yet, this is invalid in the presence of integer wrapping. If n is the maximal unsigned value, the expression $n+1$ will evaluate to 0 such that in the original code `B[0]` is not accessed at all. However, the transformed code accesses `B[0]` many times.

To ensure correctness it is necessary to perform all unsigned operations modulo the number of elements in the integer type. This means for the example above that the loop bounds should not be n and $n+1$, but n mod (`UINT_MAX`$+1$)) and $(n+1)$ mod (`UINT_MAX`$+1$). `pet` knows about the types of variables and automatically introduces the necessary modulo operations.

```
for (unsigned char k=252; (k%9) <= 5; ++k)
        S(k);
```

Listing 7: Loop with unsigned iterator

Let us now consider in a bit more detail how an unsigned loop iterator affects the way the iteration domain and the schedule are constructed. The iteration domain is first constructed as explained in

Section 9.2.3, but in terms of a virtual loop iterator, with the condition of the loop changed to apply to the modulo of this virtual loop iterator instead of the virtual iterator itself. Afterwards, a mapping is applied to the iteration domain and the domain of the schedule that wraps the virtual iterator to the real iterator, but only after intersecting the domain of the schedule with the iteration domain. This intersection is needed to ensure that we do not lose any information as some iterations in the wrapped domain may be scheduled several times, typically an infinite number of times. As an example of a loop with an unsigned iterator, consider the loop in Listing 7. The corresponding domain and schedule are as follows.

```
domain: '{ S[k] : exists
 (e0 = [(507 - k)/256]: k >= 0 and
  k <= 255 and 256e0 >= 252 - k and
  256e0 <= 261 - k) }'
schedule: '{ S[k] -> [0, o1] : exists
 (e0 = [(-k + o1)/256]: 256e0 = -k + o1 and
  o1 >= 252 and k <= 255 and k >= 0 and
  o1 <= 261)'
```

It may be difficult to see, but this loop has 10 iterations, first from 252 to 255 and then from 0 to 5. Applying the `scan` operator to the schedule in `iscc` makes this clear:

```
{ S[5] -> [0, 261]; S[4] -> [0, 260];
  S[3] -> [0, 259]; S[2] -> [0, 258];
  S[1] -> [0, 257]; S[0] -> [0, 256];
  S[255] -> [0, 255]; S[254] -> [0, 254];
  S[253] -> [0, 253]; S[252] -> [0, 252] }
```

9.4 RELATED WORK

We are aware of several proprietary compilers, including Cosy [53], ATOMIUM [38], R-Stream [114] and IBM-XL [36], that use polyhedral techniques. As these systems are not available to us and there is little documentation on their polyhedral model extractors, we cannot perform any detailed comparison with these compilers. At least three polyhedral optimizers, Bee [13], PolyOpt [103], and CHiLL [40] have been integrated in the ROSE compiler. Bee is to our understanding not publicly available. As at the time of performing this work neither CHiLL for ROSE nor PolyOpt were available, we did not perform a detailed evaluation of their features. However, to our understanding in 2012 unlike clang, ROSE did not fully support C99. In particular, ROSE did not support VLAs. Moreover, judging from the documentation [103], PolyOpt imposes somewhat severe restrictions on the allowed iteration domains, in particular requiring them to be convex. This means for example that **`else`** and `!=` are not supported. On the

other hand, the system does include an optimization engine and code synthesis. The insieme compiler[4] is reported to use an extraction tool that is somewhat similar to `pet` in that it also uses clang for parsing C code. It does however *not* use `isl` to construct and represent the polyhedral model, but instead a custom constraints based representation. The supported features appear to be similar to those supported by `clan`.

The Omega Project contains a dependence analysis tool called petit [77]. Although it does not appear to be possible to have petit dump a polyhedral model, the tool necessarily does include a parser. The input language is similar to Fortran and the parser includes some advanced features such as induction variable recognition and forward substitution of scalars. There exists also two polyhedral extractors built on top of SUIF [16], a compiler framework, which unfortunately is no longer being actively maintained and has no support for C99. The first is `pers`, a frontend used for equivalence checking [133]. pers uses a combination of SUIF passes, the Omega library [81] and an ad-hoc constraints based internal representation. Similarly to `pers` there exists a version of CHiLL [40] which uses SUIF for parsing. It seems to be superseeded by the newer ROSE based infrastructure mentioned above. The LooPo [62] project includes a polyhedral parser, which accepts subsets of both C and Fortran (or a combination) as input. Index expressions are required to be affine (rather than piecewise quasi-affine), but there is support for generic **`while`** loops [59]. The parser that comes with FADAlib [29] also supports **`while`** loops, but does not support many of the constructs supported by `pet`. PIPS [17] also performs polyhedral analysis, but mostly in the sense of abstract interpretation. Although earlier versions allowed for the extraction of a polyhedral model in our sense, i.e., with access relations and iteration domains, this functionality appears no longer to be supported.

Several open source compilers have support for extracting a polyhedral model from an internal representation, including WRaP-IT [60] in ORC, graphite [100] in gcc and Polly [3] in LLVM/clang. As explained in the introduction, such low-level parsers have their advantages and disadvantages when compared to source level parsers such as `pet`. In Table 6, we perform a more detailed comparison with Polly 3.0, which is based on the same compiler infrastructure as `pet`. The table also compares against `clan` [28], which to the best of our knowledge is currently the most popular source level parser. There are however many different versions of `clan`, including some such as irClan [30] that include some support for data dependent constructs. This irClan system does not appear to be publicly available though. Here, we compare against the latest official release (version 0.6.0) as of 2012. Some of the shortcomings pointed out here may have improved in later versions of clan.

4 `http://www.dps.uibk.ac.at/insieme/index.html`

Feature	Polly 3.0	clan 0.6	pet 0.1
General			
SCoP Detection (auto)	**yes**	**no**	**yes**
SCoP Detection (pragma)	**no**	**yes**	**yes**
Highlight unsupported code	**yes**[a]	**no**	**yes**
Parse entire source files	**yes**	**yes**[b]	**yes**
Parse isolated SCoPs	**no**	**yes**	**no**
Input language	various[c]	C-like	C99
Code synthesis	**yes**	**no**	**no**
Classical SCoP			
Affine Expressions			
- add, multiply	**yes**	**yes**	**yes**
- max	**yes**	**yes**	**yes**
- min	**no**	**yes**	**yes**
- modulo	**no**	**no**	**yes**
- division	**yes**	**no**	**yes**
- floord/ceild	**no**	**yes**	**yes**
- conditional (a ? b : c)	**no**	**no**	**yes**
Comparisons			
- $<, \leqslant, =, \geqslant, >$	**yes**	**yes**	**yes**
- $\neq$	**yes**	**no**	**yes**
- Implicit Comparison to Zero	**yes**	**no**	**yes**
Boolean Combinations			
- and (&&)	**no**	**yes**	**yes**
- or (||)	**no**	**no**	**yes**
- not (!)	**no**	**no**	**yes**
Loops			
- for-loop	**yes**	**yes**	**yes**
- Stride > 1	**yes**	**no**	**yes**
- Negative Stride	**yes**	**no**	**yes**
- Loop bound restriction	SCEV[d]	syntactic	isl
Conditions			
- if	**yes**	**yes**	**yes**
- else	**yes**	**no**	**yes**
Memory Access			
- Array (Static Size)	**yes**	**yes**	**yes**
- Array (Variable Size)	**no**	**yes**	**yes**
- Pointer Arithmetic	**yes**	**no**	**no**
Parse any CLooG Output	**no**	**no**[e]	**yes**
Semantic Analysis			
Propagate Expressions	**yes**	**no**	**yes**
Recognize Induction Variables	**yes**	**no**	**no**
Semantic Loops	**yes**	**no**	**no**
Alias Analysis	**yes**	**no**	**no**
Extensions			
Derive Array Sizes	**no**	**no**	**yes**
Infinite Loops	**no**	**no**	**yes**
Wrapping (Unsigned Ops)	**no**	**no**	**yes**
Wrapping (Casts)	**no**	**no**	**no**
Inlining	**yes**	**no**	**no**

a On intermediate language
b clan ignores everything outside the SCoP
c Any language that can be lowered to LLVM IR.
d Scalar Evolution Analysis built around the ideas in [123]
e For example, the fragment in Listing 4

Table 6: Features of different polyhedral extractors

9.5 LIMITATIONS AND FUTURE WORK

Even though pet is already very powerful, it has several limitations worth addressing.

The set of acceptable inputs could be expanded by performing more extensive analyses to obtain more relations between variables. At the moment, we only perform a very simple form of forward substitution. There is also currently no support for **`switch`** statements, although it should be fairly easy to add.

Other issues include the following. Like most high-level polyhedral model extractors, `pet` does not perform any alias analysis, but instead assumes that none of the arrays accessed in the SCoP overlap. `pet` is based on `clang`, such that conceptually C, C++ or Objective-C code can be analyzed. At the moment only C is supported. Other languages such as Fortran are not supported and are unlikely to ever be supported. Another area where `pet` can be improved is that of (implicit) casts. Support for them is not yet available, even though adding this support should not impose any difficulties. The AST generated by `clang` contains the relevant information and the use of `isl` makes adding the relevant modulo operations trivial.

9.6 SUMMARY

`pet`, the polyhedral extraction tool presented in this chapter, combines the strengths of both `clang` and `isl` to form a new tool that exploits the generality of integer sets and maps to significantly widen the set of C programs that can be modeled and reasoned about. We believe that the improved coverage will help other researchers to increase the generality of the optimizers and transformations they implement. In the context of stencil computations `pet` has shown to be immensely helpful as it allowed us to parse stencil kernels written with efficient modulo notations and/or with specialized boundary condition handling. Even though pet is still missing some correctness features such as the correct handling of array aliasing, `pet`'s support for unsigned integer operations shows clearly the goal of following existing standards precisely, such that tools based on pet are enabled to provide correct transformations even in corner cases. We also believe that the very same modeling concepts can be used to advance polyhedral optimizers in production compilers.

The work presented in this chapter has been important to enable the extraction of compute kernels as used in our work on stencil computations and was also influential as in providing insights on how to correctly model complex language semantics. It was collaboratively developed and has in parts been presented in [10]. The initial version of pet, as described in this chapter, has been implemented by Sven Verdoolaege and was subsequently extended continuously.

10 AST GENERATION

AST generation, the translation from a polyhedral model to an imperative AST, is a concept essential for enabling the application of polyhedral high-level loop optimizations on imperative code. For many years, this conceptual step has been factored out to specialized tools [26, 81, 39] which act as nice building blocks on our way of decoupling high-level loop optimizations from specific compilers.

As discussed earlier, our work on generating optimized stencil codes (Part iii) has been developed with the aim of decoupling the design of the high-level optimization strategy and the generation of efficient program code that implements this optimization strategy. Even though there are many good reasons to separate these two concerns, doing so while still ensuring the generation of highly efficient code is non-trivial. Most domain specific optimizers use specialized code generators to reach optimal performance. Being able to compete with them without implementing specialized AST generation strategies sets high requirements on various areas of the AST generator. Existing generators only partially meet these requirements, a fact which does not only limit their usefulness in the context of our optimization scenario, but which generally limits the situations in which generic AST generators can be used.

Existing AST generation approaches focus today mostly on the generation of control flow, aiming for generating the most efficient control flow for a given schedule. However, they do not offer facilities to generate user-provided AST expressions, e.g., to describe memory locations. As a result, such AST generators can not be used for data-layout transformations [68] or for mapping data to software managed caches [70], as they appear on GPUs. Despite specialization being very important to fully exploit certain optimization schemes, existing AST generators also support only certain simple forms of specialization. They commonly produce multiple code versions that are specialized for certain parameter values with the goal of reducing control overhead, but they do not allow the user to influence this versioning to drive the generation of specialized and possibly better performing code. The separation of full and partial tiles [19, 61, 82] or the specialization for boundary conditions as they may appear in stencil computations are use cases that would directly benefit from such user controlled specialization. Users can still enforce such kinds of versioning by generating several distinct copies of each statement in the input description [40], but this kind of statement duplication seems to only work around a shortcoming of the AST generator. Similar

workarounds are necessary when performing unrolling during AST generation, where users again revert to duplicating the statements in the input description [40, 35, 115]. Besides being conceptually unsatisfying, duplicating statements causes serious problems. First, by purposefully hiding the fact that statements are identical, the AST generator is forced to generate duplicate code for them in all cases, missing redundancies in complex expressions and missing opportunities to factor colder parts of the code. Secondly, the need to duplicate statements significantly complicates the construction of the AST generation input. Already the combination of specialization for full/partial tile separation with possible unrolling of the full tiles requires duplicating statements for the separation and then reduplicating the statements that should be unrolled. Even though computationally this reduplication may be cheap, it quickly becomes difficult to understand what kind and how many statement copies actually need to be computed. If we now wish to minimize code size for colder parts of the iteration space (e.g., the partial tiles), we run into the next limitation. Even though AST generators provide basic control over the desirable aggressiveness in separating statements or control flow specialization (conditional hoisting), the level of control is way too coarse-grained in existing methods and tools. Also, no guarantees are given about the maximal number of loop nests and the maximal number of statements generated, which is problematic for scenarios where code size is a major concern, such as AST generation for many-core targets with software-managed caches, embedded processors, and high-level synthesis [142]. Overall, existing approaches and tools are in many ways not yet mature for complex AST generation problems.

In this chapter we present an integrated AST generation approach that besides classical control flow generation from arbitrary integer maps, allows the generation of AST expressions from arbitrary user-provided piecewise affine expressions. We define a fine-grained "option" mechanism that enables the user to request maximal specialization where needed while retaining control over code size. To enable aggressive specialization, we allow the user to instruct the AST generator how to version the code, we provide an integrated polyhedral unrolling facility, and we make sure that AST expressions are specialized according to the context they are generated in. Doing so is essential to correctly model the floor-division and modulo arithmetic arising from abstract transformations of the program, and to cast these expressions to efficient remainder and integer divisions, or to lower the complexity of operations, as provided by existing instruction set architectures and programming languages.

10.1 A NEW APPROACH TO AST GENERATION

To give an idea of the new AST generation concepts proposed in this work, we present them in the context of our work on hexagonal/parallelogram tiling (Chapter 5). In the following paragraphs we recapture this work from the code generation perspective highlighting which optimizations need to be performed to obtain efficient code and which features of a general purpose AST generator are required to perform these optimizations.

When translating stencil kernels from C code to CUDA, we start from code consisting of compute statements and loops, which in the simplest case consists of a single perfectly nested set of loops, with one outer sequential loop, a set of inner parallel loops and a single compute statement. To generate CUDA code for this computation it is necessary to obtain a set of kernels that can be launched sequentially and that each expose two levels of parallelism: coarse grained parallelism, which will be be mapped to so-called CUDA thread blocks, and fine grained parallelism, which will be be mapped to so-called CUDA threads. To obtain these two levels of parallelism we divide the set of individual computations (statement instances) enumerated by these loops into subsets (tiles). We do this by computing a polyhedral schedule that enumerates the set of statement instances with two groups of loops. A set of outer loops that enumerate the tiles (tile loops) and a set of inner loops (point loops) that enumerate the statement instances that belong to a certain tile. The first AST generation problem we encounter is that the hybrid-hexagonal schedule defining the tile shapes decomposes the computation into phases and applies to each phase a different schedule. This results in a *piecewise schedule* from which an AST needs to be generated.

As a next step, we map the tile and point loops to a fixed number of thread blocks and threads. We start by looking for a set of parallel point loops and a set of parallel tile loops. We then strip-mine each loop by the number of thread blocks and threads. For instance to map a point loop with n iterations to a set of 1024 kernel threads, we strip-mine the loop by a factor of 1024 such that each 1024^{th} iteration is executed by the same thread. The next step is to produce a piece of CPU code that schedules instances of an accelerated kernel, and the kernel code itself that defines the computation of a specific thread in a specific thread block. No actual loops are generated that enumerate the set of thread blocks and threads, but instead the CUDA run-time and hardware spawns a set of blocks and threads, and provides the block and thread ID as a parameter to each thread executing the kernel code. To model this, we first generate the outer loop, then we introduce the block and thread identifiers and, finally, we generate kernel code that can reference values in the outer CPU code, taking into account the AST generation context of the outer C code as well

```
for (c2 = 0; c2 <= 1; c2 += 1)
  for (c3 = 1; c3 <= 4; c3 += 1)
    for (c4 = max(((t1 - c3 + 130) % 128) + c3 - 2,
                  ((t1 + c3 + 125) % 128) - c3 + 3);
         c4 <= min(((c2 + c3) % 2) + c3 + 128,
                   -((c2 + c3) % 2) - c3 + 134);
         c4 += 128)
    if (c3 + c4 >= 7
        || (c4 == t1 && c3 + 2 >= t1 && t1 + c3 <= 6
            && t1 + c3 >= ((t1 + c2 + 2 * c3 + 1) % 2) + 3
            && t1 + 2 >= ((t1 + c2 + 2 * c3 + 1) % 2) + c3)
        || (c4 == t1 && c3 == 1 && t1 <= 5
            && t1 >= 4 && c2 <= 1 && c2 >= 0))
      A[c2][6 * b0 + c3][128 * g7 + c4 - 4] = ...;
```

Figure 35: Copy code from hybrid hexagonal/parallelogram tiling (a single loop)

as the constraints on the kernel and thread identifiers. Exploiting this information is very important to generate high-quality code.

When generating kernel code we also need to rewrite all array subscripts in our compute statement. Traditionally this is done textually by replacing all references to old induction variables with expressions that compute the values of the old induction variables from new induction variables. When translating an access `A[i+1]` where `i` now is expressed as $c0 + 1$, a classical rewrite would yield `A[(c0 + 1) + 1]`. With our new approach we represent the expression $i + 1$ itself as a piecewise quasi affine expression, perform the translation on the piecewise quasi affine expression, simplify the resulting expression and use our AST generator to *generate an AST expression from this piecewise quasi-affine expression*. As a result we obtain the code `A[c0 + 2]`. In this example the only benefit is increased readability, as any compiler would constant fold the two additions. However, in general, this concept is a lot more powerful. It allows the specialization of expressions according to the context in which they are generated. If, for instance, an access `A[i == 0 ? N - 1 : i - 1]` is scheduled in a tile where we know `i` is never 0, we can simplify the access to `A[i - 1]`. This simplification removes the overhead of boundary condition handling from the core computation, a transformation for which a normal compiler misses context information and which traditionally requires specialized statements for boundary and core computations. With our AST generation approach, statements are automatically specialized as soon as boundary computations and core computations are generated as specialized AST subtrees. This is very natural for an AST generator that allows *user-directed versioning*.

After having generated basic CUDA code including the rewritten data accesses, we can start to optimize the code. An essential opti-

mization is to switch from the use of slow "global memory" to the use of fast, manually managed "shared memory" (Section 5.3.2). To do so we need to change the code of each tile such that, before the actual computation takes place, the relevant data from global memory is copied into shared memory, and at the end, the modified data is copied back from shared to global memory. To perform the computation in shared memory, we need to adjust all memory accesses such that they point to the new shared memory arrays using equally changed index expressions. How exactly the mapping is computed [135] is not of importance here. The important point in this section is that we can rely fully on our AST generator to generate the code for the memory accesses from this mapping. To do so we only need to convert our data mapping into a set of piecewise quasi-affine expressions that define the new data locations. We then *generate AST expressions* for them, relying on the AST generator to ensure that these expressions are translated into efficient code. This approach enables us to use possibly complex mappings, without writing specialized code generation routines. To create the code that moves the data we create new statements that copy data from a given global memory location to a given shared memory location and vice versa. In case there is more data to copy than there are threads we use a modulo mapping to assign data locations to threads. Figure 35 shows the code generated to copy data back to global memory. There are various interesting observations to be made. First, we see that our modulo expressions have been mapped to the C remainder operator `%`, which will be translated to fast bitwise operations. This is only possible because we have context information about the value of `t1`. Otherwise we would need to fall back to expensive `floord` or `intMod` expressions, dealing with arbitrary (possibly negative) integers, like the state-of-the-art AST generators `CLooG` and `CodeGen+` do. Secondly, we see that we generate a reasonably dense loop nest that enumerates the statements. Because of the presence of existentially quantified variables in the input description, this is by itself non-trivial (see Section 10.5.1).

Nevertheless, we observe that the generated code is not very efficient. Every loop iteration performs very little computation and evaluates a complex condition. One might hope the condition could be simplified further, but unfortunately the data modified when moving a 5-point stencil forming a cross over a hexagonal tile shape is by itself already non-convex. Applying another level of modulo scheduling makes the necessary compute pattern even more complex, such that obtaining a simpler loop structure is difficult. However, by using *polyhedral unrolling* on the inner three loops and by specializing the statements according to the iteration they are unrolled for we can remove almost all control overhead. The result is shown in Figure 36. The code is very smooth and each array subscript is specialized to the specific location. We can also see that for the conditionally executed

```
A[0][6 * b0 + 1][128 * g7 + (t1 + 125) % 128) - 1] = ...;
A[0][6 * b0 + 2][128 * g7 + (t1 + 127) % 128) - 3] = ...;
if (t1 <= 2 && t1 >= 1)
    A[0][6 * b0 + 2][128 * g7 + t1 + 128] = ...;
A[0][6 * b0 + 3][128 * g7 + (t1 + 127) % 128) - 3] = ...;
if (t1 <= 2 && t1 >= 1)
    A[0][6 * b0 + 3][128 * g7 + t1 + 128] = ...;
A[0][6 * b0 + 4][128 * g7 + (t1 + 125) % 128) - 1] = ...;
A[1][6 * b0 + 1][128 * g7 + (t1 + 126) % 128) - 2] = ...;
A[1][6 * b0 + 2][128 * g7 + (t1 + 126) % 128) - 2] = ...;
if (t1 <= 3 && t1 >= 2)
    A[1][6 * b0 + 2][128 * g7 + t1 + 128] = ...;
A[1][6 * b0 + 3][128 * g7 + (t1 + 126) % 128) - 2] = ...;
if (t1 <= 3 && t1 >= 2)
    A[1][6 * b0 + 3][128 * g7 + t1 + 128] = ...;
A[1][6 * b0 + 4][128 * g7 + (t1 + 126) % 128) - 2] = ...;
```

Figure 36: Copy code from hybrid hexagonal/parallelogram tiling (unrolled)

statements the subscripts are optimized according to the conditions such that the remainder operations disappear entirely. Unrolling this code is not trivial, as it needs to be performed in the presence of multiple loop boundaries as well as strides and we need to support the generation of guarded instructions when unrolling. The guarded instructions at the innermost level are very cheap on a GPU, as they can be implemented as predicated instructions. In this small example this is not very visible, but for realistic tile sizes a larger number of statements share the same conditions. We perform similar unrolling for the compute code in our kernel to ensure sufficient instruction level parallelism is available.

The code in Figure 36 is now close to optimal. However, so far we only looked at a simplified example, a single tile which does not touch any iteration space boundaries. In case iteration space boundaries are taken into account the generated code is a lot more complex. To ensure we can still use the "close to optimal" code most of the time, we use *user directed versioning* to isolate the core computation (the full tiles) from the set of tiles that need to take into account the boundary conditions (partial tiles). Doing so gives us maximal specialization and best performance. However, we now specialize and unroll not only the core computation, but also the code that was introduced to handle the boundary cases which increases the size of the generated code as well as the time necessary to generate it. When targeting a GPU this may be acceptable, but for FPGAs [142] the cost may be prohibitive. This problem can be easily addressed by using *fine-grained options* to limit the amount of unrolling and specialization in the boundary tiles.

```
for (int i = 0; i < n; ++i) {
S1: s[i] = 0;
    for (int j = 0; j < i; ++j)
S2:         s[i] = s[i] + a[j][i] * b[j];
S3: b[i] = b[i] - s[i];
}
```

Figure 37: Example Program

In summary, extending AST generation beyond the creation of control flow makes it possible to use automatic AST generation in complex scenarios. Even though existing AST generators combined with workarounds such as duplicating statements before running the AST generator can be used to solve some of the previously mentioned AST generation issues, such workarounds only exist for some features, they apply only in simple special cases and often inhibit other necessary transformations. By instead carefully integrating several important new extensions into a single AST generation approach, we significantly extend the concept of automatic AST generation such that it is usable in complex AST generation scenarios. We ensure that the different features do not block each other, but when combined provide novel opportunities and solutions to complex AST generation problems. As a result we hope to not only significantly simplify AST generation, but to enable its use in new optimization scenarios.

10.2 INPUT

The input of our AST generator is a polyhedral model of a SCoP as defined in Section 2.2.3. For AST generation the most relevant elements of this model are the iteration domain and the schedule. We recapture again, the iteration domain describes the statement instances that need to be executed and the schedule describes the order in which they should be executed.

For the simple example program in Figure 37, the iteration domain is

$$\{S_1(i) : 0 \leqslant i < n; S_2(i,j) : 0 \leqslant j < i < n; S_3(i) : 0 \leqslant i < n\}. \tag{28}$$

and the following is a possible schedule describing the original execution order.

$$\{S_1(i) \to (i,0,0); S_2(i,j) \to (i,1,j); S_3(i) \to (i,2,0)\}. \tag{29}$$

An alternative execution order may be obtained through the schedule

$$\{S_1(i) \to (0,i,0,0); S_2(i,j) \to (1,i,0,j); S_3(i) \to (1,i-1,1,0)\}. \tag{30}$$

The purpose of the AST generator is to construct an AST that visits the elements of the iteration domain in the lexicographic order of the integer tuples assigned to the iteration domain elements by the schedule.

10.3 ABSTRACT SYNTAX TREE

The generated AST contains only syntactical information and has been designed to be easily translatable to both C and compiler IR. Each node of the AST is of one of four types, an *if node*, a *for node*, a *block node* and a *user node*. An if node has an AST expression as condition, a then node and optionally an else node. A for node has initialization, condition and increment expressions and a body node. A block node represents a compound statement and maintains a list of nodes. Finally, the statement expressed by a user node is represented as an AST expression.

An AST expression is itself a tree with operators in the internal nodes and integer constants or identifiers as the leaves. The set of operators contains the standard operators found in C-like programming languages, but also higher level operators such as min and max. Boolean logical operators and the conditional operator (`cond ? a : b`) are available in two forms, lazy and eager. We found in our work on low-level compilers [3] that eagerly evaluating operands, instead of using C's lazy evaluation, is often beneficial as it reduces control overhead and simplifies the hoisting of loop invariant subexpressions.

The integer division operator also comes in different forms, one of them corresponding to the mathematical operation $\lfloor a/b \rfloor$. Unfortunately, this operation cannot be translated directly into `a / b` in C because the `/`-operator in C rounds toward zero rather than toward negative infinity. A correct translation to C involves a condition on the sign of a, which can bring significant extra costs on some architectures such as GPU devices. We therefore also have a form of the integer division where the result is known to be an integer (such that rounding becomes irrelevant) and one where the dividend is known to be non-negative. The user can specify a preference for these latter forms in which case the AST expression generator will look for opportunities to use them (see Section 10.4.5).

10.4 NEW AST GENERATION FEATURES

The core control flow generation algorithm and infrastructure developed in this work uses algorithmic foundations from "Quilleré et al." which it extends to handle arbitrary integer maps and combines with several new optimization strategies, e.g., for component detection. As improvements to classical AST generation are out of scope for this thesis, we refer the interested reader to [4] for a precise description of

```
for (int c0 = 0; c0 <= 99; c0 += 1) {
  if (n >= c0 + 1) {
    for (int c1 = 0; c1 <= 99; c1 += 1) {
      if (c1 >= 30)
        B(c0, c1);
      A(c0, c1);
    }
    for (int c1 = 100; c1 <= 199; c1 += 1)
      B(c0, c1);
  } else
    for (int c1 = 20; c1 <= 199; c1 += 1) {
      if (c1 >= 30)
        B(c0, c1);
      A(c0, c1);
    }
}
```

Figure 38: Interleaved schedule without code generation options

the internal data structures, a description of how they are used as well as a couple of interesting optimizations. In this section we focus on the functionality that goes beyond AST generation. This includes our new-fine grained option mechanism, the isolation of components, polyhedral unrolling as well as the generation of user provided AST expressions.

10.4.1 *Fine grained option mechanism*

Our new AST generator has an option mechanism that is significantly more fine-grained than previous approaches. Where previous tools allowed certain options to be set only globally or at most on a per-statement and/or per-dimension level, our new option scheme allows us to set options on a per instance level. To enable such fine grained options we describe the options themselves as an integer map $\{(s_0, \ldots, s_{d-1}) \to \text{optionname}(\text{dim}) \mid \text{cond}(s_0, \ldots, s_{d-1}, \text{dim})\}$, a mapping from the individual elements of the schedule space into the option space. The semantics of this mapping is such that for a specific instance a certain option, as defined by the name in the option space, is set in case the condition $\text{cond}(s_0, \ldots, s_{d-1}, \text{dim})$ is true for this instance. As visible in the option map shown, options can be dimension specific, which means for different dimensions "dim" different options can be set. The two options easiest to understand are 'atomic' (minimize code size) and 'separate' (minimize control overhead), which control the amount of separation used during the

generation of control flow. We will use them to explain our option mechanism in combination with the following schedule

$$
\begin{aligned}
&\{\\
&\quad A(i,j] \to (i,j] \mid \quad 0 \leqslant i < 100 \wedge ((i < n \wedge 0 \leqslant j < 100) \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad (i \geqslant n \wedge 20 \leqslant j < 200))\\
&\quad B(i,j) \to (i,j) \mid \quad 0 \leqslant i < 100 \wedge 30 \leqslant j < 200\\
&\},
\end{aligned}
$$

a slightly contrived example that allows us to reason about different choices during AST generation. Figure 38 shows the code generated by our AST generator without any options provided. We can see that the code is specialized for different values of n and that each case contains reasonable code. However, depending on our definition of "optimal" the choices taken by the AST generator may not yet be perfect. Assuming we know n is commonly small, we may want to further optimize the code such that iterations in the branch that is rarely executed are optimized for minimal code size, whereas iterations in the other branch are optimized for minimal control overhead. With our new option mechanism we can do so easily by providing the option map $\{(i,j) \to \text{atomic}(1) \mid i < n; (i,j) \to \text{separate}(1) \mid i \geqslant n\}$. It specifies that for all iterations with $i < n$ the loop at dimension one should be optimized for code size, whereas for iterations with $i \geqslant n$ the loop at dimension one should be optimized for minimal control flow. Looking at the result shown in Figure 39, we removed one duplicated statement and one loop from the rarely executed code at the top. In the else branch, the often executed part of our code, we eliminated the if condition in the innermost loop at the cost of increasing code size by introducing an additional for loop.

Even though the optimizations performed in this example are rather simple, the technique behind them is very powerful. Complex cases as they arise from the use of complex tile shapes or the generation of code for boundary condition handling, can often cause significant code growth which is not only costly in terms of code size, but can also increase compilation time. Being able to apply different code generation strategies is consequently very useful. We also believe that the interface chosen is very convenient, as often the set of iterations that are within the actual core computation may be rather complex to describe, but possibly easy to compute using generic operations on integer maps. Being able to directly feed the integer set description of the relevant iterations to our AST generator makes it easy to use specific options just for the iterations in these sets. Despite these benefits, we would like to note that this interface is rather new and we still need to gain more experience in how beneficial these fine-grained options are for a wider set of use cases. As a result the interface as well as the option granularity may still change.

```
for (int c0 = 0; c0 <= 99; c0 += 1) {
  if (n >= c0 + 1) {
    for (int c1 = 0; c1 <= 199; c1 += 1) {
      if (c1 >= 30)
        B(c0, c1);
      if (c1 <= 99)
        A(c0, c1);
    }
  } else {
    for (int c1 = 20; c1 <= 29; c1 += 1)
      A(c0, c1);
    for (int c1 = 30; c1 <= 199; c1 += 1) {
      B(c0, c1);
      A(c0, c1);
    }
  }
}
```

Figure 39: Interleaved schedule with code generation options

10.4.2 *Isolation*

The option mechanism also allows the user to ask our AST generator to isolate a subset of the schedule domain from the other parts of the schedule domain. A trivial example is the schedule $\{A(i) \rightarrow (\text{floor}(i/4), i) \mid 0 \leqslant i < n\}$ as it results from strip mining a one dimensional loop by four. The code generated by default looks as follows:

```
for (int ii = 0; ii <= floord(n - 1, 4); ii += 1)
  for (int i = 4 * ii; i <= min(n - 1, 4 * ii + 3); i += 1)
    A(i);
```

After strip-mining with a factor of four the inner loop has at most four iterations. However, we notice that due to a `min` condition in the upper bound this very loop may also execute less than four times in certain cases. Specifically, in case `n` is not a multiple of four, the last time the inner loop is executed, only $n \bmod 4$ iterations will be executed. The additional complexity introduced just for the last iterations often prevents the generation of efficient code for these cases. In case we strip-mine, for example to vectorize the inner loop, we need a way to ensure the inner loop has exactly four iterations. Our AST generator enables us now to provide an option of the form $\{(i,j) \rightarrow \text{separation\_class}((0) \rightarrow (1)) \mid 0 \leqslant i < \lfloor n/4 \rfloor\}$, which ensures that the iterations of a given set of iterations are at dimension zero isolated from the remaining set of iterations. The result is the following code.

```
for (int c0 = 0; c0 < floord(n, 4); c0 += 1)
  for (int c1 = 4 * c0; c1 <= 4 * c0 + 3; c1 += 1)
```

```
    A(c1);
if (n >= 0 && n >= 4 * floord(n, 4) + 1)
  for (int c1 = -((n + 3) % 4) + n - 1; c1 < n; c1 += 1)
    A(c1);
```

We can see that the core computation has been isolated from the handling of the remaining iterations. As a result, the inner loop of the core computations has now always exactly four iterations with no complexity added due to the handling of the remainder. The epilog generated to handle the remainder looks rather complex. However, as it is rarely executed this complexity has a low price.

As just shown, the isolation support provided by our AST generator enables the automatic generation of possibly complex prolog/epilog. Even though we believe this concept is already highly useful when applied to one dimensional inner loops, its real power is due to it being universally supported across all dimensions and for subsets of arbitrary shape. One use case that immediately benefits is the separation of full and partial tile shapes.

10.4.3 *Polyhedral unrolling*

Loop unrolling is an optimization commonly used to increase instruction level parallelism as well as to reduce loop management overhead. In its simplest form, a loop with a known constant number of loop iterations n is unrolled by duplicating the loop body n times, by adjusting the copied bodies such that each performs one of the original loop iterations and by removing the original loop management structure. This transformation is called "full unrolling". It is natively implemented in our AST generator.

As loop unrolling is well supported by existing C/C++ compilers, one may wonder if there is any benefit from performing loop unrolling directly in the AST generator. One obvious benefit is that unrolling in the AST generator makes it possible to inspect the unrolled code. However, more importantly, during AST generation we still have all polyhedral information available which can be taken into account both when determining the number of iterations to unroll as well as when specializing the copies of the original loop body for specific loop iteration(s). In many cases the additional context information enables the AST generator to heavily simplify the copies of the replicated loop bodies and to produce less or even no code to handle boundary cases.

There was previous work on Polyhedral unrolling [124] for the special case of full unrolling for a uni-modular schedule, where a dimension is bound by a single lower and a single upper bound with a constant non-parametric difference between the two constraints defining the bounds. Listing 8 shows an example of this very specific pattern both generated as a loop and as a set of unrolled statements. It nicely

```
/*
 * Input:   {A[i] → [i] : n ⩽ i < n + 3}
 */

/* Loop */
for (i = n; i < n + 3; i++)
  A(i);

/* Unrolled */
A(n);
A(n+1);
A(n+2);
```

Listing 8: Trivial unrolling example

```
/*
 * Input:   {A[i] → [i] : 0 ⩽ i < 4096 ∧ t1 = i mod 1024}
 * Context: {0 ⩽ t1 < 1024}
 */

/* As a loop */
for (i = t1; i <= 4095; i += 1024)
  A(i);

/* Unrolled */
A(t1);
A(t1 + 1024);
A(t1 + 2048);
A(t1 + 3072);
```

Listing 9: Unrolling in the presence of strides

illustrates the idea of polyhedral unrolling. The polyhedral unrolling support in our new AST generation approach provides a highly generalized version of this polyhedral unrolling. First, it supports more complex inputs by handling arbitrary integer maps containing existentially quantified dimensions or defining piecewise schedules. Second, it increases the applicability of loop unrolling by avoiding the matching of specific constraints, but rather inspecting certain properties of the polyhedral sets. Finally, it extends polyhedral unrolling to several new use cases, all generalizations of full unrolling.

Due to our support for existentially quantified dimensions, we can easily express loops with strides. Listing 9 gives an example where the original loop has a stride of 1024. Such code is commonly created when generating code for GPUs where the 4096 iterations are mapped to 1024 threads. As the number of iterations per-thread are

commonly low, unrolling is very important to enable instruction level parallelism.

```
/*
 * Input:   {A[i] → [i] : 0 ⩽ i < n}
 * Context: {3 ⩽ n < 6}
 */

/* As a loop */
for (int i = 0; i < n; i += 1)
  A(i);

/* Unrolled */
  A(0);
  A(1);
  A(2);
  if (n >= 4) {
    A(3);
    if (n == 5)
      A(4);
  }
}
```

Listing 10: Unrolling in case of bound, non-constant number of iterations

We also support unrolling for loops where the number of iterations is not a constant, but where it is bound by a constant. Listing 10 gives an example where the loop itself is bound by a parameter n, but we know that the maximal number of loop iterations can never exceed five. In this case, we can unroll the loop five times, but need to add guards for the iterations that may possibly not be executed. Even though these guards seem similar to the cleanup code necessary for partial unrolling, this is still full unrolling. Being able to unroll a loop even though guards are needed can be very beneficial in cases where the number of guards needed is small or in cases where executing them is cheap or even free. If, for example, the number of loop iterations in Listing 9 would not have been a multiple of 1024, the last unrolled statement would need to be guarded. This guarded statement could directly be translated to a predicated instruction which on modern GPUs has a very low cost.

Another important case is loops with multiple lower bounds, as commonly introduced by loop tiling. When unrolling such loops it is very important to choose the lower bound such that the number of iterations that need to be unrolled is minimized. In Listing 11 we see two ASTs that are both generated from the schedule $\{A[i] \to [i] : 0 \leqslant i < 5 \wedge n \leqslant i < n+3\}$. The first one uses 0 as lower bound, the second one uses n. Comparing both we see that by choosing 0 instead of n as

```
/*
 * {A[i] → [i] : 0 ⩽ i < 5 ∧ n ⩽ i < n + 3}
 */

/* Loop */
for (int i = max(0, a); i <= min(4, a + 2); i += 1)
  A(i);

/* Unrolled with a as lower bound */
if (n <= 4 && n >= 0)
  A(n);
if (n <= 3 && n >= -1)
  A(n + 1);
if (n <= 2 && n >= -2)
  A(n + 2);

/* Unrolled with n as lower bound */
if (n <= 0 && n >= -2)
  A(0);
if (n <= 1 && n >= -1)
  A(1);
if (n <= 2 && n >= 0)
  A(2);
if (n <= 3 && n >= 1)
  A(3);
if (n <= 4 && n >= 2)
  A(4);
```

Listing 11: Unrolling with two lower bounds

lower bound, we can reduce the number of unrolled statements from 5 to 3. For tiled code, where the size of the original iteration space is commonly a lot larger than the tile size, this difference can become very large.

10.4.4 *Partial Unrolling*

```
// Original code
// Input: {A[i,j] → [i,j] : 0 ⩽ i < n ∧ 0 ⩽ j < m}
for (i = 0; i < n; i += 1)
  for (j = 0; j < m; j += 1)
    A(i, j);
```

Listing 12: Partial unrolling - original loop nest

```
// Tiling (2 x 2)
// Input: {A[i,j] → [⌊i/2⌋,⌊j/2⌋,i,j] : 0 ⩽ i < n ∧ 0 ⩽ j < m}
for (iT = 0; iT < (n + 1) / 2; iT += 1)
  for (jT = 0; jT < (m + 1) / 2; jT += 1)
    for (iP = 2*iT; iP <= min(n - 1, 2*iT + 1); iP += 1)
      for (jP = 2*jT; jP <= min(m - 1, 2*jT + 1); jP += 1)
        A(iP, jP);
```

Listing 13: Partial unrolling - tiled

Unrolling as supported in our AST generator is a rather sophisticated form of full unrolling (Section 10.4.3), unrolling with the goal of fully eliminating a certain loop. In cases where the number of loop iterations is unknown or where full unrolling would yield unnecessary or unacceptable code growth, it is often better to perform partial unrolling. We show now in several steps how partial unrolling possibly involving multiple loops can be implemented with the new facilities our AST generator provides.

```
// Unroll iP and iJ loops
// Input: {A[i,j] → [⌊i/2⌋,⌊j/2⌋,i,j] : 0 ⩽ i < n ∧ 0 ⩽ j < m}
for (iT = 0; iT < (n + 1) / 2; iT += 1)
  for (jT = 0; jT < (m + 1) / 2; jT += 1) {
    A(2 * iT, 2 * jT);
    if (m >= 2 * jT + 2)
      A(2 * iT, 2 * jT + 1);
    if (n >= 2 * iT + 2) {
      A(2 * iT + 1, 2 * jT);
      if (m >= 2 * jT + 2)
        A(2 * iT + 1, 2 * jT + 1);
    }
  }
```

Listing 14: Partial unrolling - tiled + unrolled

We start with a two dimensional loop nest as shown in Listing 12. As a first step, we tile this loop nest using the unroll factors as the tile sizes. The resulting code (shown in Listing 13) contains now two outer tile loops and two inner point loops, with the number of iterations in the point loops being bound by the unroll factors. However, as visible from the presence of `min` expressions in the loop bounds, the number of loop iterations is not constant, but may possibly be smaller than the unroll factor due to `n` or `m` not being a multiple of the unroll factor. We can still ask our AST generator to fully unroll the point loops of our tiling to obtain the partially unrolled loop nest shown in Listing 14, but the `min` conditions reappear as possibly undesired conditions. To remove them we ask the AST generator to isolate (Section 10.4.2) the

```
// Unroll iP and iJ loops & Isolate partial tiles
// Input: {A[i,j] → [⌊i/2⌋, ⌊j/2⌋, i, j] : 0 ⩽ i < n ∧ 0 ⩽ j < m}
for (iT = 0; iT < n / 2; iT += 1) {
  for (jT = 0; jT < m / 2; jT += 1) {
    A(2 * iT, 2 * jT);
    A(2 * iT, 2 * jT + 1);
    A(2 * iT + 1, 2 * jT);
    A(2 * iT + 1, 2 * jT + 1);
  }
  if ((m - 1) % 2 == 0) {
    A(2 * iT, m - 1);
    A(2 * iT + 1, m - 1);
  }
}
if ((n - 1) % 2 == 0)
  for (jT = 0; jT < (m + 1) / 2; jT += 1) {
    A(n - 1, 2 * jT);
    if (m >= 2 * jT + 2)
      A(n - 1, 2 * jT + 1);
  }
```

Listing 15: Partial unrolling - tiled + unrolled + isolated core computation

tiles that lie entirely in the iteration space from the tiles which are only partially executed. Doing so removes any conditions from the core unrolling and separate code is generated for the remaining iterations. The result is now a smooth implementation of register tiling. The same concept is applicable to loop nests of arbitrary depth and with possibly complex iteration space boundaries.

10.4.5 *Generating AST Expressions*

One new core functionality of our AST generator is that it provides facilities to the user to generate AST expressions from an arbitrary integer map or piecewise quasi affine expressions. Ensuring efficient expressions are computed even in the context of modulo constraints is important for those user generated integer expressions. However, due to our new AST generation approach generally having widened the support for modulo constraints, the optimizations discussed here are also beneficial for AST expressions generated as part of the control flow.

Within `isl` integer divisions are represented in terms of greatest integer parts ($\lfloor \cdot \rfloor$). In principle, these expressions can be translated directly into their AST expression counterparts, but as explained in Section 10.3, for some use cases it is important to know if the first argument of an integer division is non-negative or if the division is exact.

Moreover, we typically want an expression of the form $m \left\lfloor (a(\vec{i})/m) \right\rfloor$ to be translated to $a(\vec{i}) - (a(\vec{i}) \bmod m)$, provided again that $a(\vec{i})$ is non-negative.

Whenever generating an if or for-condition or a for initialization or upper bound expression from an expression involving greatest integer parts, we first check for opportunities to extract modulo expressions and then check the sign of the remaining greatest integer parts. Note that when generating a conjunction of constraints, we first generate expressions for the constraints not involving greatest integer parts such that we can exploit those constraints when simplifying the remaining constraints.

If we are generating an equality constraint, we first check if the equality encodes a stride. If so, the stride can be expressed in the AST using an expression of the form `x % m == 0`. In this case, the sign of `x` is of no importance. For other constraints or expressions in general, if we find an subexpression of the form $fm \left\lfloor a(\vec{i})/m \right\rfloor$ and we can prove that $a(\vec{i})$ is non-negative based on context information, then the expression is replaced by $fa(\vec{i}) - f \cdot (a(\vec{i}) \bmod m)$. If $a(\vec{i})$ may be negative, but $-a(\vec{i}) + m - 1$ can be proved to be non-negative, then it is replaced by $f \cdot (m + 1 - a(\vec{i})) - f \cdot ((m + 1 - a(\vec{i})) \bmod m)$ instead, exploiting the fact that $\lfloor a/b \rfloor = -\lceil -a/b \rceil = -\lfloor (-a + b - 1)/b \rfloor$. Moreover, the $a(\vec{i})$ inside the argument of mod can be replaced by any $a'(\vec{i}) = a(\vec{i}) + me(\vec{i})$. We therefore look for constraints $h(\vec{i}) \geqslant 0$ among the shifted shared constraints of the context with coefficients that are either equal or opposite to those of $a(\vec{i})$ modulo m. Since $h(\vec{i})$ is known to be non-negative in the context, it can be used directly as $a'(\vec{i})$. If no such constraint can be found, we check if $a(\vec{i})$ or $-a(\vec{i}) + m - 1$ themselves can be proved to be non-negative by solving an ILP problem. The latter test is also used to check if the first arguments of the remaining integer divisions are non-negative. These simple heuristics appear to work out fairly well in practice.

10.5 EXPERIMENTAL RESULTS

We also performed a set of experiments to evaluate our AST generator in comparison with `CLooG` and `CodeGen+`. Some evaluations, e.g., the quality of the generated control flow and the correctness of the generated code are outside of the scope of this thesis. We refer the interested reader again to [4]. In our experiments, we mostly focus on the new AST generation features as well as the support for existentially quantified variables.

10.5.1 *Existentially quantified variables*

```
// Simple
S(n % 128);

// Shifted
S(((t1 + 121) % 128) + 7);

// Conditional
if ((t1 + 121) % 128 <= 123)
  S(((t1 + 125) % 128) + 3);
```

(a) isl

```
// Simple
for(i = intMod(n,128); i <= 127; i += 128)
  S(i);

// Shifted
for(i = 7+intMod(t1-7,128); i <= 134; i += 128)
  S(i);

// Conditional
for(i = 7+intMod(t1-7,128); i <= 130; i += 128)
  S(i);
```

(b) codegen+

Figure 40: Modulo conditions (examples not supported by CLooG)

```
// Two e.q. variables
for (int c0 = 0; c0 <= 7; c0 += 1)
  if (2 * (2 * c0 / 3) >= c0)
    S(c0);

// Multiple bounds
for (i = 0; i <= 1; i += 1)
  for (j = max(-((-t1 + t2 - 3)%128) + t2 - 387, t1-384);
       j <= min(-t2 + 127, t2 - 383); j += 128)
    if (j + 256 == t1 || (j + 384 == t1 && t1 >= 126))
      S(i, j);
```

(a) isl

```
// Two e.q. variables
S(0); S(2); S(3);
S(4); S(5); S(6); S(7);

// Multiple bounds
if (t1 >= 126)
  S(0, t1 - 384);
S(0, t1 - 256);
if (t1 >= 126)
  S(1, t1 - 384);
S(1, t1 - 256);
```

(b) isl unrolled

Figure 41: Existentially quantified variables (examples not supported by CLooG/codegen+)

Generating a valid AST for any valid Presburger relation and ensuring that we use efficient remainder operations whenever possible is one of the design goals of our AST generation algorithm. To do so it is important to correctly handle existentially quantified variables, as they can result from modulo mappings from global to shared memory or from a full iteration space to a set of thread ids. We start with a simple modulo operation $\{[i] \to [i] : i = n \bmod 128\}$ to verify that modulo operations can be detected at all. Since older versions of `CLooG` (prior to our enhancements) do not allow existentially quantified variables, we do not compare against it in this section. For `isl` and `CodeGen+`, Figure 40 shows that `isl` uses a single statement with a remainder operation, whereas `CodeGen+` generates a loop. Using a loop is very inefficient, not only due to the call to `intMod` and the general loop overhead, but especially because the expression `n%128` is invariant of any possibly surrounding loop and has almost zero cost as the loop invariant code motion pass of a compiler normally moves it out of the loop body. Two slightly more complicated examples are mappings from a set of iterations to a set of threads `t1` with $\{0 \leqslant t1 < 128\}$. The first mapping is the one-to-one mapping $\{[i] \to [i] : 7 \leqslant i \leqslant 134 \wedge i \bmod 128 = t1\}$ which `isl` again translates into a single instruction, the second is the mapping $\{[i] \to [i] : 7 \leqslant i \leqslant 130 \wedge i \bmod 128 = t1\}$ which maps 124 iterations to 128 threads. `isl` lowers this mapping to a single conditional statement. `CodeGen+` generates for both cases a full loop nest. It is interesting to note that all previously shown loops are degenerate loops with just a single iteration. `CodeGen+` is not able to detect those loops, whereas `isl` is designed to always recognize degenerate loops.

For the previous test cases only a single existentially quantified variable was introduced due to the one modulo operation in the schedule. For more complex use cases, e.g., the modulo mapping of access functions that already contain modulo expressions or nested modulo mappings, it is often possible that multiple existentially quantified variables are introduced. The first test case $\{[i] \to [i] : \exists(\alpha, \beta : i = 2\alpha + 3\beta \wedge 0 \leqslant \alpha < 3 \wedge 0 \leqslant \beta \wedge 0 \leqslant i < 8)\}$ involves two existentially quantified variables in a single equality. `CodeGen+` aborts here with the message *guard condition too complex to handle*. In Figure 41 we see that `isl` is able to generate valid code, which can be unrolled both for better efficiency and to better understand the computation that is performed. The next test case is $\{[i, j] \to [i, j] : \exists(\alpha, \beta : 0 \leqslant i \leqslant 1 \wedge t1 = j + 128\alpha \wedge 0 \leqslant j + 2\beta < 128 \wedge 510 \leqslant t2 + 2\beta \leqslant 514 \wedge 0 \leqslant 2\beta - t2 \leqslant 5)\}$, which was reduced from the example in Figure 35. `CodeGen+` aborts with *Can't generate multiple wildcard GEQ guards right now.* `isl` either generates a loop with multiple loop bounds and remainder conditions or, if unrolled, a set of conditional statements. As Chen [39] does not discuss how existentially quantified variables are handled, the scope of support in `CodeGen+` is unclear. When inspecting the source

AST generation options		heat 2D		heat 3D	
		GFLOPS	speedup	GFLOPS	speedup
a:	no options enabled	1.9	1.0x	4.9	1.0x
b:	all optimizations enabled	26.4	13.9x	19.6	4.0x
c_1:	all, except full/partial separation	19.4	10.2x	18.2	3.7x
c_2:	all, except IO unrolling	4.5	2.3x	9.6	2.0x
c_3:	all, except compute unrolling	14.1	7.4x	10.1	2.1x
c_4:	all, except modulo detection	27.5	14.1x	16.9	3.4x

Table 7: AST generation strategy based performance (GFLOPS)

code of `CodeGen+` we found several code paths that require a single existentially quantified variable per constraint. isl has no limitations on the number of existentially quantified variables per constraint.

10.5.2 *Performance of AST generation strategies*

To understand the performance implications of our new AST generation strategies, we analyze their impact on the run-time of generated code. We ensure a realistic scenario by analyzing a full end-to-end domain specific compiler. As compiler we choose the stencil compiler introduced in Section 10.1. We remind the reader that this compiler is based on the general purpose compiler PPCG. To create code that is optimized for the domain of stencil computations the computation of a generic execution schedule is replaced with the computation of a hybrid hexagonal/parallelogram execution schedule specifically optimized for the domain of stencil computations. Besides the domain specific schedule, the only other domain specific piece is the parametrization of our AST generator to isolate (Section 10.4.2) full tiles from partial tiles as well as to unroll (Section 10.4.3) compute and IO code. AST expression generation (Section 10.4.5) is used to specialize the access functions of statements, e.g., after unrolling or separation.

We perform the evaluation on a NVIDIA NVS 5200M GPU using a heat 2D and a heat 3D stencil as benchmark. As performance results have shown large differences between two and three dimensional stencils, we choose two benchmarks to cover the most common dimensionalities. We limit ourselves to a single type of stencil, as the general tendency between different types of stencils does not vary enough to give additional insights for this analysis. For further performance results on different hardware and different stencil types, we refer to Chapter 5. Table 7 shows the results of our analysis. We see in a that normal AST generation with no further specialization enabled yields very low performance with just 1.9 GFLOPS in the 2D

case and 4.9 GFLOPS in the 3D case compared to b where we enable all optimizations and obtain 26.4 GFLOPS in the 2D case and 19.6 GFLOPS in the 3D case, a 13.9x speedup in the 2D case and a 4.0x speedup in the 3D case. To understand better where this speedup comes from we individually disable certain optimizations. In c_1 we disable full/partial tile separation, which reduces the performance by 27% for heat-2D and 7% for heat-3D. The larger change on 2D is due to the higher percentage of full tiles. In 3D, already a large amount of time is spent in partial tiles, so optimizations that speed up the execution of full tiles are less visible. In c_2 and c_3 we see that for heat 3D disabling either unrolling of IO or unrolling of compute reduces the performance by 50%. For the 2D case, disabling unrolling of the compute code also reduces the performance by 50% and, even more importantly, without unrolling of the IO code over 80% of the performance is lost. This large performance difference is both due to the increased ILP after unrolling and because of the simplifications enabled by unrolling (see Figure 36). In c_4 we see that without modulo detection the performance for heat-3D is reduced by 14% and, surprisingly, slightly increased by 4% in 2D. The increase for heat-2D is due to register spilling caused by loop invariant code motion which again was made possible due to the simpler code after modulo detection. Allowing `nvcc`, the NVIDIA compiler, to use more registers prevents register spilling and modulo detection becomes again beneficial with a new peak performance of 28.4 GFLOPS for heat 2D, an 8% performance improvement over the previous peak value. Overall, we see that just generating control flow using polyhedral scanning is by far not enough to generate high-performant GPU code. Instead, both polyhedral unrolling and specialization for full and partial tiles are highly important to obtain code of competitive performance.

10.5.3 *Generation Time*

Although we have mostly focused on the newly introduced features, for completeness we also report some AST generation times. For this experiment, we take 64 of the test cases distributed with `CLooG` (those that can be handled by both `CLooG` 0.14.1 and `CodeGen+`) and sum the total AST generation time. For `CLooG` 0.14.1 (before our enhancements, using `PolyLib` as a backend), we obtain 0.3s using fixed size integer computations and 1.0s for arbitrary precision integers. For `CLooG` 0.18.1 (including some of our enhancements and using `isl` for set operations with arbitrary precision integers), we obtain 0.9s. For `CodeGen+`, we find 3.1s and for `isl`, 1.5s. We attribute the time difference with respect to `CLooG` to the fact that we have not yet implemented some of the heuristics of Vasilache et al. [124] and that we are much more aggressive in our optimizations, resulting in better output code.

10.6 RELATED WORK

There are two major approaches to generic AST generation, one that is based on a library for Presburger relations and that focuses on lifting control overhead up [80, 39] and one that is based on rational polyhedra and that mainly tries to eliminate overhead top-down [26, 107]. Even though we did not present technical details on control flow generation, we would like to note that the algorithm used can be seen as a combination of the previous two approaches, using the same separation algorithm of [26, 107], but built on top of a library for Presburger relations. Sven Verdoolaege started the original work on this new AST generator by porting `CLooG` to `isl` and improving `CLooG`. The knowledge gained from then influenced the development of the new AST generation approach presented here. Both the original approaches only allow single disjunct contexts and schedules, with `CLooG` also not supporting existentially quantified variables. `CodeGen+` can handle such variables in certain cases, but as Chen [39] does not discuss how such such variables are handled in general, the extent of support is unclear. In contrast, our AST generator supports the full generality of Presburger arithmetic, including existentially quantified variables and piecewise schedules.

Kelly et al. [80] and Chen [39] as well as Quilleré et al. [107] and Bastoul [26] generate AST expressions as necessary to generate control flow for scanning the iteration space, but they do not expose any functionality to generate AST expressions for arbitrary user-provided piecewise quasi-affine expressions. We also are not aware of any work that uses the AST generation context to specialize AST expressions. In particular, we are aware of no work that uses context information to optimize modulo operations and divisions as they appear in quasi-affine expressions. `CodeGen+` always generates expensive `intMod` calls and `CLooG` only introduces a `%` operator in cases where the result of the operator is compared to zero.

Polyhedral unrolling in an AST generator has been proposed (without software being made available) by Vasilache et al. [124] for the special case of a unimodular schedule where a dimension that has a single lower and single upper bound offset by a constant non-parametric distance can be fully unrolled. In our work we presented polyhedral unrolling for schedules defined by arbitrary Presburger maps, with support for unrolling in presence of multiple lower bounds, unrolling in the presence of strides and unrolling for loops with bound, non-constant number of iterations using conditional statements. User-directed isolation of arbitrary subsets of the iteration space as such has not been implemented in polyhedral AST generators. The automatic separation used by Bastoul [26] regularly introduces specialized code versions, but the user can only control the amount of separation and not the subsets that are separated from each other. Full/partial

tiling has been discussed as an independent transformation by Ancourt and Irigoin [19] as well as Goumas et al. [61] and, combined with unrolling, by Jiménez et al. [76]. In the context of parametric tiling [82, 110, 67] full/partial tile separation has been researched in AST generators specialized for this use case. We are not aware of any work that uses a generic isolation feature provided by a polyhedral AST generator to perform full/partial tile separation. As parametric tiling techniques commonly rely on polyhedral AST generators, the same isolation techniques may be useful in the context of parametric tiling.

We are not aware of any work that provides configurability on such a fine grained level. Bastoul [26] originally allowed per-dimension level control over separation and recently gained per-statement control. Chen [39] allows per loop level control over the amount of control flow. Different AST generation strategies for different subtrees of the generated AST are to our knowledge unique to our work. Also, giving the user the ability to enforce an "atomic" AST generation strategy to minimize code size or to enforce unrolling is new.

10.7 SUMMARY

In this chapter we widened the scope of polyhedral AST generation, by presenting an AST generator that extends traditional control flow generation to the full generality of Presburger arithmetic based integer maps. In particular, we provide support for piecewise affine schedules as well as schedules with complex uses of existentially quantified variables, opening AST generation to new application areas and more sophisticated program optimizations, and enhancing its reliability—the ability to predictably generate highly efficient control flow. We also acknowledge that optimization problems are not limited to control flow restructuring, but also require changes to data access functions: to support such optimizations, we propose facilities to generate efficient AST expressions from piecewise quasi-affine forms. Finally, we improve on the state of the art techniques to recover divisions and modulo expressions in the generated code, and apply these to the optimization of index expression that commonly appear in the context of explicitly managed caches. Overall, we widened the scope of generic AST generators.

However, to implement domain or target specific optimizations that reach peak performance, it is often necessary to heavily specialize the generated code. For this we allowed the AST generator to be parameterized to perform loop unrolling and partial evaluation of loop iterators in a very general, polyhedral setting. Furthermore, we presented how to separate certain parts of the code and show how to use this separation to generate specialized code for full and partial tiles. By allowing the specialization of user-provided AST expressions

according to the context they are generated in, the same feature can also be used to generate specialized code for boundary conditions. As maximal specialization may not always be best, we make AST generation choices such as separation, unrolling and also atomic execution configurable on a fine-grain level. Each individual contribution is by itself useful, but only the integration in a single AST generator ensures their seamless interaction. As demonstrated on hybrid hexagonal/classical tiling, the result is an AST generator that can be used to implement complex domain specific optimizations.

The work presented in this chapter has been motivated by the need for advanced AST generation techniques that enable us to exploit the advantages of our new tiling techniques and that simplify the generation of complex run-time checks in the context of LLVM/Polly. It was collaboratively developed and will, possibly with modifications, be submitted for publication [4]. The new AST generation techniques presented in this chapter have been implemented in isl by Sven Verdoolaege, with isl 0.11 being the first isl release with AST generation support.

11

SCHEDULE TREES

In the previous parts of this thesis we used integer maps to describe schedule transformations, as this is currently the best understood way to represent schedules and schedule transformations. However, just because it is best understood does not mean it is without shortcomings. Quite the contrary, when constructing complex schedule transformations as they arise from advanced tiling schemes, the generation of GPU mappings or the exploitation of software manged caches flat schedule descriptions such as the named integer maps we used have shown to complicate the reasoning over and the construction of more complex schedules. To make it easier to understand schedules and to transformation them we looked into new ways to represent them.

To get an idea how such a representation may look like we looked into how different algorithms generate a schedule and if this generation may suggest a specific schedule description. Most scheduling algorithms recursively decompose a dependence graph, at each level separating the graph into (strongly connected) components and computing a partial schedule for each component separately. This partial schedule is usually an affine function (possibly quasi-affine and/or piece-wise). The complete schedule is then obtained as some form of concatenation of the partial schedules. The order of two statement instances in the complete schedule is determined by the outer partial schedule that yields a different value for the two instances or the outer pair of components that separates the two instances, whichever appears outermost. An overview of several of such early algorithms is provided by Darte et al. [47]. When constructing tilable bands (e.g., the Pluto algorithm [35]), the partial schedules are multi-dimensional affine functions and the order determined by a partial schedule is given by the lexicographic order on its function values.

The above description suggests a representation of a schedule in the form of a *tree*, with each node representing a partial schedule and the order of the components determined by the order of the children of a node. Such a representation also seems the most natural way to represent the original order of a program, when extracted from some form of AST. We are however unaware of any prior work that explicitly operates on such schedule trees (apart from our own "band forests" in isl). Instead, the schedule trees are *encoded* in one way or another and all operations are performed on the *encodings* of the schedule trees. Depending on the chosen encoding, these operations quickly become cumbersome and/or hard to understand.

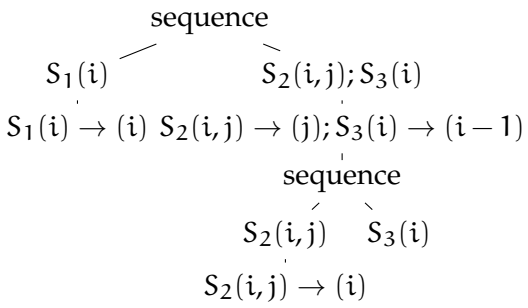


Figure 42: Example schedule tree representation

In this chapter, we propose to use an *explicit* representation of a schedule as a tree and to perform all operations directly on this tree. We argue that such a representation is *more natural*, *more practical* and *easier to understand*. Figure 42 illustrates a schedule tree representation of the schedule

$$\begin{aligned} T_1 : \{(i) &\to (0, i, &)\} \\ T_2 : \{(i, j) &\to (1, j, 0, i &)\} \\ T_3 : \{(i) &\to (1, i-1, 1 &)\} \end{aligned}$$

in Kelly's abstraction [78, 79] or

$$\Theta^{S_1} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} \quad \Theta^{S_2} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \Theta^{S_3} = \begin{bmatrix} 0 & 1 \\ 1 & -1 \\ 0 & 1 \end{bmatrix}$$

in the representation of Girbal et al. [60]. We first describe the general concept of a schedule tree and show how it generalizes the different schedule representations that have been proposed in the past. We subsequently propose a specific *instance* of the schedule tree concept, which we aim to implement in and integrate into our tools (isl [128], Polly [3], PPCG [135]). As we plan to refine our design in this process, the currently presented representation is not yet final and may change. However, looking at the current version it can already be noted that it is *more general* than earlier proposals, allowing an explicit representation of subtrees that can be executed in parallel and the introduction of additional symbolic constants in subtrees. Finally, we show how we used this new representation to *simplify* the implementation of PPCG [135], significantly improving the maintainability of the tool and enabling future extensions.

11.1 SCHEDULE USES

Depending on the framework used, there may be a single iteration domain containing all statement instances or there may be an iteration domain per statement. Invariably, though, the elements of an iteration domain are (possibly named) vectors of integers, called *iteration vectors*. In some approaches, these elements determine an implicit execution order, but this means that whenever a transformation is applied that changes the execution order, the iteration domains themselves and everything that depends on the iteration domains (such as access relations and dependences) need to be updated. In this paper, we will therefore assume that, as is common practice, the execution order is determined by an explicit schedule that maps the elements of the iteration domains to some other objects with an implicit execution order. Note that a schedule only prescribes a *relative* execution order and that there are therefore typically an infinite number of ways to express the same execution order, independently of the chosen schedule representation. In this section, we describe some uses of schedules in general that we will use to compare the different schedule representations in Section 11.2.

11.1.1 *Original execution order*

Although much work has been devoted to automatic scheduling techniques that construct a schedule directly from the dependence graph, the ability to interactively perform polyhedral transformations starting from the original execution order is useful for teaching or manual exploration. Some implementations of dataflow analysis [56] (e.g., that of isl) also start from a polyhedral representation involving some form of schedule (even though it may be more advisable to perform the dataflow analysis before or during the extraction of a polyhedral representation from an AST using techniques similar to lazy array data-flow analysis [92] or array region analysis [46]). Our schedules therefore need to be able to represent the original execution order.

When extracting a polyhedral model from an imperative program, we mainly need to deal with two constructs in terms of execution order, compound statements and loops. In order to model the effect of compound statements, a schedule needs to be able to express a sequence, i.e., that one set of statement instances should be executed after some other set of statement instances. To model the effect of a loop, the schedule needs to be able to express an order defined by an affine function on the iteration vectors. In the simplest case, the iteration vectors are composed of the values of the iterators of the enclosing loops. In this case, the affine function simply projects the iteration vector onto the iterator of the loop that needs to be modeled. In general, the iteration vectors can be any sequence of integers

that uniquely identify a statement instance and a more complicated function may be required to express the effect of a loop.

11.1.2 *Transformations*

A polyhedral representation can be transformed either by constructing a new schedule or by modifying some previously obtained schedule. This other schedule may be a schedule representing the original execution order or it may be the result of an earlier transformation.

11.1.2.1 *Schedule Construction*

As an example of an automatic scheduling algorithm, let us sketch a general overview of the "Pluto algorithm" [35]. The algorithm takes a dependence graph as input and recursively constructs schedule *bands*. The dependence graph expresses which statement instances depend on which other statement instances and therefore need to be executed after those other statement instances. The nodes of the dependence graph are composed of the statements, while the edges carry the dependence relations.

At each level of the recursion, the algorithm first checks for (weakly connected) components in the dependence graph. These components do not depend on each other in any way and can therefore be scheduled independently. Within each component, the algorithm looks for strongly connected components (SCCs) and (optionally) marks them to be executed in their topological order. For each (group of) SCC(s), the algorithm then constructs a sequence (or band) of one-dimensional affine functions such that each of these functions respects the dependences independently of the other functions in the same band, they are linearly independent of each of the other functions in the same and in outer bands, and such that they optimize some optimization criterion. After the construction of a band is completed, the dependence graph is updated to only contain dependences between pairs of statement instances that are mapped to the same function values by the current band and the process repeats.

The constructed schedule is therefore (at least conceptually) a tree that recursively consists of collections of statement instances that can be executed in any order, sequences of statement instances that need to be executed in the specified order and multi-dimensional affine functions.

11.1.2.2 *Schedule modification*

Given a schedule (either corresponding to the original execution order or constructed from a dependence graph), we may want to apply a series of additional transformations. In keeping in line with a clear separation between the statement instances (in the iteration domain)

and the order in which they are executed (defined by the schedule), these transformations need to be expressed as transformations on the schedule itself.

The transformations that we may want to apply include affine (typically uniform) transformations, statement reordering, fusion, distribution, index set splitting, strip-mining and tiling. Most of these transformations do not require any additional constructs beyond collection, sequence and multi-dimensional affine functions. The only exceptions are strip-mining and tiling. These transformations require the use of integer divisions and/or modulo operations and therefore require (explicit or implicit) *quasi-affine* expressions. Note that we only consider non-parametric strip-mining and tiling here.

11.1.3 *AST generation*

AST generation takes an iteration domain and a schedule as input and produces an AST that visits each element of the iteration domain in the order specified by the schedule. This operation is also known as polyhedral scanning [19, 39] or code generation [26]. The constraints on the schedule representation imposed by AST generation are not so much in what the schedule should be able to express, but in the kind of constructs for which an AST can be generated. Clearly, generating an AST for a collection of statement instance groups or for a sequence of such groups is trivial. Piecewise quasi-affine schedule functions can also be handled by standard AST generators [26, 39].

The iteration domain and the schedule may refer to symbolic constants (also known as parameters). If the iteration domain is non-empty for only some values of these symbolic constants, then the generated AST may contain explicit conditions on the symbolic constants. Most AST generators allow the user to avoid the generation of such conditions by providing the AST generator with known constraints on the symbolic constants. This additional piece of information is known as the *context* and is usually passed separately to the AST generator. A final piece of information required by the AST generator is a set of options that control the way the AST is generated.

11.2 SCHEDULE REPRESENTATIONS

Many different schedule representations have been proposed in different contexts. Some of these proposed representations only serve as the input or output to a given algorithm and are therefore typically unsuitable as a generic schedule representation. Other proposals, such as "Kelly's abstraction" [81, 79], "$2d+1$-schedules" [60] and "union maps" [129], have been specifically designed as generic representations. In this section, we compare some of these representations.

11.2.1 *Properties*

In particular, we compare the following aspects of the schedule representation

SCATTEREDNESS Some representation consist of a single schedule object, while in other representation the schedule information is spread over different objects, typically one object for each statement.

COMPOSITIONALITY Compositionality is usually interpreted to mean that the same schedule representation can be used as both the input and the output of schedule transformations. In some cases, the schedule transformations themselves can be composed before being applied to the schedule.

PARTIAL SCHEDULES Some schedule representations are very restrictive and only allow a limited set of predefined, implicit partial schedules. Other representations allow affine, quasi-affine or even piecewise quasi-affine partial schedules. Recall that quasi-affine schedules are required to express strip-mining or tiling. Some representations also restrict the partial schedules to a single dimension.

SEQUENCE Many schedule representations do not support an explicit representation of sequence. Instead, each group of statement instances in the sequence is assigned a distinct increasing number. These increasing numbers may be assigned as (part of) a regular partial schedule, or they may be specified through a dedicated mechanism, typically only allowing a single constant for all instances of a statement. That is, these dedicated mechanisms typically do not allow the set of instances of a given statement to be broken up into two or more parts.

COLLECTION Very few schedule representations are able to express that groups of statement instances can be executed in arbitrary order with respect to the other groups. Instead, such collections are usually encoded in the same way as sequences, fixing a particular execution order of the groups.

INJECTIVITY Some early schedule representations explicitly allowed different statement instances to be assigned the same value in order to express inner parallelism. Other representations treat inner parallelism in the same way as other forms of parallelism and expect the statement instances that can be executed in parallel (at a given position in the schedule) to be assigned different values using a partial schedule, but somehow mark one or more dimensions in this partial schedule as parallel.

SINGLEVALUEDNESS Some schedule representations allow a given statement instance to be assigned more than one value by the schedule, i.e., they allow the statement instance to be executed more than once.

LEXICOGRAPHIC ORDER Two schedule values (either within a partial schedule or over the entire schedule) are usually compared based on the lexicographic order. In some representations, this lexicographic order is only defined on vectors of the same dimension. That is, the (partial) schedule is expected to have the same dimension across all statements. This corresponds to the standard definition of lexicographic order as found in most textbooks. We will call this a strict interpretation of lexicographic order. In other representations, the schedule vectors can have different dimensions and then the shortest vector is compared to the prefix of the longest vector of the same size as the shortest vector. We will call this a relaxed interpretation of lexicographic order.

11.2.2 *Comparison*

Some of the earliest schedule representations within the context of polyhedral compilation appear in Feautrier's work [57]. The input schedule is mostly implicitly encoded in the iteration domains, augmented with the number of shared loops for each pair of statements and the textual order of each pair of statements. The relative order of two statement instances is determined by the lexicographic order on the prefixes of their iteration vectors of length equal to the number of shared loops and by the textual order of the statements. The output schedule is a multi-dimensional piecewise quasi-affine schedule, with relaxed lexicographic order, implicit inner parallelism and sequence encoded as any other schedule row.

The first compositional representation appears to have been proposed by Kelly [78, 79]. Each statement keeps track of its part of the schedule. The partial schedules may be any multi-dimensional piecewise quasi-affine functions. Sequence is encoded by special schedule dimensions that are marked as "syntactical" and that assign the same constant value to all instances of a statement. Schedule values are compared using a relaxed lexicographic order. Although an explicit index set splitting operation is provided, index set splitting typically happens implicitly through the use of a piecewise partial schedule. When transforming a subtree of the schedule, the subtree is identified by the statements that are transformed by that subtree. The schedules appear to be padded with zeros prior to being sent to the AST generator.

The "$2d + 1$" representation [60] can be seen as a further specialization of Kelly's abstraction. In particular, the partial schedules are

	Kelly	$2d+1$	union map
scatteredness	per statement	per statement	single object
compositional	schedule	schedule	transformation
partial schedule			
- representation	p.w.q.a.	affine	p.w.q.a.
-dimension	arbitrary	1	arbitrary
sequence	cst per statement	cst per statement	p.w.q.a.
collection	n/a	n/a	n/a
injective	yes	yes	yes
single-valued	yes	yes	no
total	possibly	yes	no
lexicographic order	relaxed	relaxed	strict

	band forest	schedule tree
scatteredness	single object	single object
compositional	schedule	schedule
partial schedule		
- representation	p.w.q.a.	p.w.q.a.
-dimension	arbitrary	arbitrary
sequence	p.w.q.a.	explicit
collection	implicit	explicit
injective	yes	yes
single-valued	yes	mostly
total	yes	internally
lexicographic order	relaxed	relaxed

Table 8: Comparison of some generic schedule representations

restricted to one-dimensional purely affine functions (compared to the multi-dimensional piecewise quasi-affine functions of Kelly). The single-dimensional partial schedules are interleaved with constant statement level dimensions that express sequence. The restriction to purely affine functions means that they are unable to express strip-mining and tiling in the schedule itself and instead have to resort to a modification of the iteration domains, undermining the separation between iteration domains and schedule and the compositionality of their approach. The restriction to one-dimensional partial schedules (between statement level dimensions) means that unimodular transformations involving more than one loop dimension need to be applied across statement level dimensions. When transforming a subtree of the schedule, the subtree is identified by the values of the outer shared statement level dimensions (the "β-prefix"). Although this may appear to be more generic than using the statements involved, this is in fact not the case as each statement can only have a single β-prefix.

The "union map" representation [129] uses named integer union maps (Section 2.1.2) to essentially pad the per-statement schedules of Kelly with zeros to ensure that all schedules have the same dimension and then combines the per-statement schedules into a single schedule object. This single object is a binary relation on tuples that maps named integer vectors (with the names representing the statements), to integer vectors of a fixed length. The main advantage of this representation is that it is not tailored to schedules. In particular, the same abstraction is also used to represent access relations and dependence relations, allowing for a uniform manipulation. Moreover, the *changes* to the schedule are also represented using the same abstraction and can be combined prior to being applied to the schedule. The non-specificity to schedules is also the main disadvantage of the union map representation. Whereas the tree structure is still visible in Kelly's abstraction and in the $2d + 1$ representation through the marking as syntactical dimensions and the implicit statement level dimensions, this structure is completely hidden in the union map representation. The mapping to a single schedule space also causes local transformations to potentially have a global effect. For example, if some part of the schedule tree is tiled, increasing the total number of schedule dimensions, then the other parts of the schedule need to be padded to maintain a single schedule space.

The "band forest" abstraction that was available in versions 0.07 to 0.12 of isl builds on top of the union maps, but makes the tree structure explicit. It was used to represent the schedules computed by isl's scheduler, which is very similar to the Pluto scheduler. Each node in the tree represents a tilable band, with a partial schedule represented by a union map. Siblings (including the roots of the forest) represent groups of statement instances that can be executed in parallel.

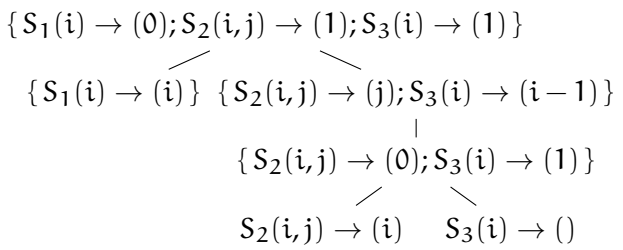


Figure 43: Band forest representation of the schedule in Figure 42

Sequence is still encoded, in particular as a single-dimensional band with a union map that assigns constant values to groups of statement instances. The children of such a band can be executed in parallel because the band itself takes care of the ordering. The only operations that were made available for this abstraction are splitting a band into two nested bands and tiling a band. Figure 43 shows an example of a band forest.

Table 8 provides an overview of the above comparison, including a comparison to the schedule trees of Section 11.3. Note that while union map schedules are typically single-valued, this is not strictly required. While the band forest is essentially a tree of union maps and it is technically possible for the band forest not to be single-valued, the band forests that are constructed in practice are always single-valued. The "total" row indicates whether the schedule is a total function. When using union maps, it is customary to encode the iteration domain in the domain of the schedule so that no separate iteration domain object is required. It is then also possible to select a subset of some larger external iteration domain. This is also possible in the case of Kelly's abstraction, but it is not clear if this is the intended use. The abstractions with a relaxed lexicographic order use a strict order within the partial schedules.

CLooG [26] and the union maps based isl code generator impose a strict lexicographic order on its input, requiring users to pad the schedules. Omega's code generation [39] does not impose a strict order. URUK [60] uses "2d + 1" schedules, with subtrees identified by prefixes. Pluto [35] allows partial schedules (i.e., bands) of any dimension, but maintains the restriction of purely affine partial schedules, therefore also requiring a modification of the iteration domains to express tiling (this restriction can be optionally disabled in newer versions to allow integration into Polly [3]). A strict lexicographic order is imposed and subtrees are identified by sets of statements and a loop depth. PoCC [102] appears to be using the schedule representation of older versions of Pluto where bands were defined globally across the entire schedule tree. It also has some support for "reentrancy" [126], i.e., a conversion back and forth between polyhedral representation and AST. The Graphite [122] internal representation

is similar to Kelly's, with the "syntactic" label replaced by a "loop nest tree" (only for the original schedule). The CHiLL [40] representation is also similar to Kelly's, except that partial schedules are single-dimensional as in the $2d+1$-schedules, with unimodular transformations applied across partial schedules. Subtrees are identified by sets of statements and a loop depth. Polly [3] essentially uses union maps, but breaks them up over the statements. Incremental transformations in AlphaZ [140] are performed through a modification of the iteration domains. It is also possible to specify an additional purely affine schedule function with strict lexicographic order.

11.3 SCHEDULE TREE REPRESENTATION

Based on our experience with the different kinds of schedule representation, we designed a new representation that, like the band forest, maintains an explicit tree structure. Unlike the band forest, however, the schedule tree has different types of nodes that allow for an easier manipulation and the ability to attach more information to the tree. In particular, sequence and collection are represented explicitly as nodes rather than being encoded in a band or implied by the tree structure. Our description is still somewhat preliminary, but our use of schedule trees in PPCG provides some initial evidence of the usefulness of this representation.

11.3.1 *Nodes*

The following types of nodes are available in the new schedule trees.

CONTEXT A context node introduces symbolic constants and known constraints on those symbolic constants. The introduced symbolic constants can be used in the descendants of the context node. The context node typically appears as the root of the schedule tree, but it can also be useful to introduce additional context nodes in the tree. As a convenience to the user, the outer context node may also be left out, in which case it is assumed that the symbolic constants used in the tree can take on any value.

DOMAIN A domain node introduces the statement instances that are scheduled by the descendants of the domain node.

FILTER A filter node selects a subset of the statement instances introduced by outer domain nodes and retained by outer filter nodes. Filter nodes are typically used as children of set and sequence nodes (described next), where the siblings select the other statement instances.

SEQUENCE A sequence node expresses that its children should be executed in order. These children must be filter nodes, with typically mutually disjoint filters.

SET A set node is similar to a sequence node except that its children may be executed in any order.

BAND A band node contains a partial schedule on the statement instances introduced by outer domain nodes and retained by outer filter nodes. This partial schedule may be piecewise quasi-affine, but is total on those statement instances. Additionally, a band node contains properties of the band and options that control the AST generation. The set of properties includes whether the band is tilable and which of the band dimensions may be executed in parallel. The AST generation options mainly control whether a band dimension should be separated or whether it should be unrolled.

MARK A mark node allows the user to mark specific subtrees of the schedule tree.

Sequence and set nodes have one or more children. The other types of nodes have at most one child.

The inclusion of context, domain and AST generation options in the schedule tree means that only a single object needs to be passed to the AST generator. This is especially important for the options. The original interface to the isl AST generator allowed for a very generic specification of options based on constraints on the schedule dimensions. The only purpose of this generic mechanism, however, was to be able to express that some options should be applied to a specific node of the schedule tree encoded in the union map. By attaching the options directly to the band nodes, the complexity of the generic option mechanism (mostly for the user, but also for the implementation), can be avoided completely. In particular, if the schedule is modified after some options have been set, then there is no longer any need to try and apply the same transformation to the options description as the local options automatically remain attached to the correct band node.

Note that while the schedule tree now contains both domain information (in the domain nodes) and schedule information (in the band nodes), the information is still kept in separate nodes so that, internally, the schedule is a total function on the domain. The explicit representation of a sequence means that the scheduler does not need to encode the sequence as a piecewise constant partial schedule only to have the AST generator identify this piecewise constant partial schedule as a sequence. Note that this type of node is only meant as a convenience for cases where the scheduler or user wants to impose an explicit sequence. If an automatic scheduler constructs a partial

piecewise affine schedule in one of its substeps that just happens to be piecewise constant, then the corresponding band node does not have to be converted to a sequence node. The explicit representation of a collection rather than forcing an arbitrary order allows the user and the AST generator to reorder the children, without having to re-analyze the dependences.

The basic schedule tree representation has the same expressivity as Kelly's abstraction, union maps and band forests. The main advantages are the ease of manipulation and the potential for extensions.

11.3.2 *Operations*

The following operations are available to modify schedule trees. It is important to note that none of these operations modify the domain node(s). That is, even though the domain information is integrated in the schedule object, it is still kept separate from the actual scheduling information.

- Insert a context, domain, filter or mark node at a given position.
- Apply a piecewise quasi-affine transformation to a band node. The input space of the transformation is equal to the schedule space of the original partial schedule. The output space of the transformation becomes the new schedule space. This operation can be used to implement any unimodular transformation on the band, but also strip-mining and tiling within the band, possibly combined with index set splitting.
- Split a band node into two nested band nodes, each node holding a part of the original output domain.
- Combine two nested band nodes into a single band node. This is the opposite of a Split.
- Tile a band node. This operation is essentially the same as tiling the band within the node through the application of a piecewise quasi-affine transformation and then splitting the node into a band corresponding to the tile loops and a band corresponding to the point loops.
- Fuse two bands. This operation pushes an explicit order on a pair of bands down, combining the two bands into a single band. This operation typically has the effect of loop fusion on AST generation. The bands need to be grandchildren of the same sequence with adjacent (filter) parents or grandchildren of the same set. Additionally, the schedule spaces of the bands need to be the same since the two schedules will be combined into a single schedule. The two parents are replaced by a single filter node with as filter the union of the filters. The two bands

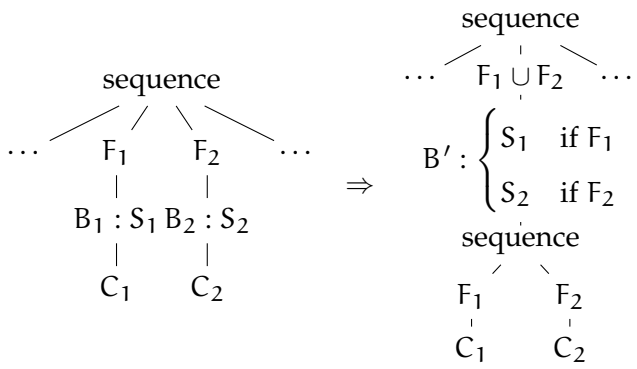


Figure 44: Fuse bands B_1 and B_2

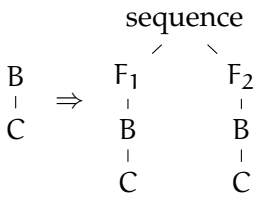


Figure 45: Order the active statement instances at B according to filters F_1 and F_2

are replaced by a single band with as partial schedule the partial schedules of the original bands on the corresponding filters. The new band has a single sequence child with as children the original two filters with in turn as children the children of the original bands (if any). This transformation is shown schematically in Figure 44.

- Ordering. This operation takes a band node and two filters that partition the active statement instances at the band node as input. The tree is updated such that the elements that satisfy the first filter are executed before the elements that satisfy the second filter. That is, the subtree rooted at the band is duplicated. Each of the filters is inserted in one of the copies of this subtree and subsequently attached as children of a sequence node that replaces the original band. This transformation is shown schematically in Figure 45 and can be used to express a generalized form of loop distribution that allows for the separation of a subset of the instances of a statement.

- Reorder the children of a sequence node.

- Sink a band. This operation moves a band node down to the leaves of the subtree underneath its original position.

Figure 46: Input pattern for hybrid tiling

This section has shown how to apply each of the transformations of Section 11.1.2.2 on a schedule tree representation. Note that we do not allow for the application of affine transformations *across* bands. This restriction can be seen as a form of type safety. Instead, the bands first need to be explicitly combined and/or fused into a single band node, after which the transformation can be applied inside the single band node. In the extreme case all bands are merged into a single band node, in which case the schedule tree essentially degenerates into a union map representation.

11.4 HYBRID HEXAGONAL-PARALLELOGRAM TILING

The best way to get a feeling of how schedule trees simplify the description and modification of complex schedules is to look into how schedule trees can be used to facilitate the handling of such schedules. One use case where complex schedules are common is PPCG. We already gained first experience with the use of schedule trees in PPCG and would like to refer the interested reader to [11, Section 5]. Another situation where complex schedules appear is our hybrid hexagonal/parallelogram tiling scheme. In this section, we will take a look at the schedule trees involved in hybrid hexagonal tiling.

Hybrid tiling as described in Section 5.2 can be seen as a transformation of a subtree of a possibly larger schedule tree. For hybrid tiling to be applicable, this subtree needs to have certain properties. Specifically, it needs to consist of a single-dimensional sequential band B_1 followed by a possibly multi-dimensional parallel band B_2, where parallel means all dimensions are parallel. We illustrate such a schedule tree in Figure 46. When applying hybrid tiling, this subtree is now replaced by a more complex tree with a shape corresponding to the tree illustrated in Figure 47. The individual elements in this tree correspond to the individual parts of our hybrid schedule. The first node, B_3, enumerates the individual tiles along the time dimen-

sion. It is followed by a sequence node which enumerates the two tile phases (phase 0/phase 1). Each phase consists of a filter node (F_1/F_2) that selects the iterations belonging of the specific phase, a node that enumerates the tiles along the hexagonally tiled dimension (B_4/B_5) as well as a node that enumerates the tiles along the parallelogram tiled dimensions (B_6/B_7). Finally, both phases share the same intra tile schedule (B_1 and B_2).

The individual nodes can be mapped directly to certain parts of the hybrid hexagonal/parallelogram schedule. B_3 consists of (4) and (6) as well as the constraints on the hexagonal tile shapes. F_1 filters for iterations that belong to phase 0 using the constraints given in Section 5.2.1.3, F_2 contains corresponding constraints for phase 1. For the hexagonally tiled dimension, B_4 consists of (5), and B_5 of (7). For the parallelogram tiled dimensions, B_6 consists of (16) and (17), and B_7 consists of (16) and (18). The intra tile schedules in B_1 and B_2 correspond to the schedule in the original input.

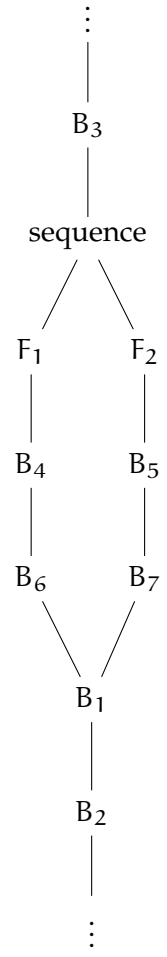


Figure 47: Output pattern for hybrid tiling

Working on a schedule tree also simplifies the description of our GPU code generation strategy, as we can simply map certain schedule tree nodes to certain parts of the GPU code. Hence, we quickly reformulate our GPU mapping in terms of schedule trees: We map B_3 and the sequence node to the host code. F_1 and F_2 do not yield specific code, but only restrict the set of iterations enumerated. As the hexagonally tiled dimension can be executed in parallel, we map B_4 and B_5 to block identifiers to enable coarse grained parallelism. B_6 and B_7 bound the shared memory usage, but can not be executed in parallel. Consequently, they are mapped to loops executed sequentially within each thread block. B_1 is also not parallel and is consequently mapped to an explicit loop as well. As B_2 is parallel we map the dimensions that belong to B_2 to parallel thread dimensions (up to three on CUDA) and add synchronization primitives to preserve the sequential order implied by B_1.

11.5 SUMMARY

In this chapter we identified that many polyhedral algorithms work conceptually with schedules of a tree like structure, but that most existing schedule representations commonly do not expose this structure. After analyzing and comparing existing schedule representations, we proposed the idea of exposing this inherent structure in the form of explicit schedule trees. We presented one instance of such a schedule tree and described the set of individual tree nodes as well as the operations available to transform this tree. Our schedule tree is not only more expressive than existing schedules due to e.g., providing information about unordered subtrees, but we believe it is conceptually easier to both understand and work with. With our work on hybrid-hexagonal tiling we showed one use case where schedule trees facilitate the description and discussion of a complex polyhedral schedule.

The work in this chapter has been shown useful to facilitate the description of schedule transformations as they arise from complex GPU code generation strategies. It has been collaboratively developed and was published in parts in [11]. Initial prototype implementations of our schedule tree work have been implemented by Sven Verdoolaege in the context of isl.

Part V

LOW-LEVEL COMPILERS

12 CONTRIBUTIONS TO LLVM / POLLY

Even though not the main focus of our research, we have been continuously involved in the development of Polly [3], a high-level loop optimizer for LLVM. Both by own contributions as well as by mentoring various students we continued to establish a polyhedral high-level loop optimization framework which enables easy transfer of our research results to industry grade compilers.

In the following sections we will give a brief overview of projects and ideas we have been involved in within the last years. One project, our work on delinearization, will be discussed in more detail in Chapter 13.

12.1 COMPUTE OUT

For production compilers it is essential that the compile time of a certain piece of code is predictable. Or said differently, the compilation of a certain source code file can at most take a couple of seconds with the compile time being linear in the size of the input file. Our polyhedral optimization techniques unfortunately can not give such guarantees and can in certain cases take several minutes to complete (e.g., dependence analysis problems with many parameters). To address this concern we added a compute-out timer to isl to bound the time we spend on our optimizations. In cases when we trigger this compute-out, we gracefully fall back to the original code.

We tested our compute out on the LLVM test suite.[1] We managed to set the compute timer such that all but two kernels finish within their compute budget. The failing ones had computation times of over a minute. Aborting them consequently seems the right choice.

12.2 AST GENERATION

In parallel to the development of our new AST generator (Chapter 10), we ported the code generation phase in Polly from CLooG to the new AST generator. This porting effort helped to shape several of the design decisions in our AST generator. Today, we can run our AST generator on the full LLVM test suite achieving performance in both compile time as well as generated code similar to CLooG. Even though many of the interesting features of our new AST generator have only been used in our research work, we are very much looking forward to exploit them in the context of LLVM as well.

1 http://llvm.org/docs/TestingGuide.html

Under my supervision, Roman Gareev also integrated our new AST generator into gcc/Graphite, such that we now have a state-of-the-art polyhedral loop optimization infrastructure available in the two leading open source compilers.

12.3 GPOLLY - AUTOMATIC GPU OFFLOADING

Yabin Hu has been developing under my supervision an extension called GPolly, which combines Polly and PPCG. GPolly is an optimization pass, that without the help of any user annotations automatically detects interesting SCoPs and translates them into CUDA kernels as well as the corresponding CUDA library calls to schedule the kernels. If GPolly is loaded into clang, a classical C/C++ compiler, the generated CUDA code is transparently embedded into the final binary. As a result, the only user interaction necessary to compile a C/C++ program in a way that it takes advantage of GPU acceleration is the need to enable our GPolly optimizer.

The general flow of GPolly starts with the detection of interesting SCoPs and their translation to a polyhedral representation. As both Polly and PPCG use isl to represent the polyhedral information, we can directly pass the resulting SCoP descriptions to PPCG. Using this description PPCG then derives a new AST for the given SCoP that describes both host code and kernel code. As Polly and PPCG use internally the isl AST generator, the existing infrastructure to translate an isl AST into LLVM IR can mostly be reused for performing this translation. Only some additional support for CUDA specific extensions such as kernel calls, synchronization instructions as well as the allocation of shared memory is still necessary. Also, some work was necessary to translate the generated CUDA kernels from LLVM-IR to PTX and to embed the PTX code correctly into the programs.

We found two of our own contributions very helpful in enabling the development of GPolly. First of all, the ability to generate user-defined AST expressions (Chapter 10) and their use to model the changes to memory accesses after mapping them to GPU global or local memory made the transfer to Polly very easy as no new code had to be written to generate these expressions in the context of Polly. Secondly, to properly derive the footprint of the data that needs to be transfered to the GPU our work on delinearization (Chapter 13) has been shown to be essential to handling arrays of parametric size.

We believe being able to develop GPU specific optimizations in the context of our source-to-source compiler and then moving them directly to a production compiler shows nicely that the polyhedral model can help to decouple high-level loop optimizations from specific compilers.

12.4 REPRESENTING PARALLELISM

Preserving information about parallel loops across compiler optimizations is necessary, but non-trivial when working on a low-level IR. When compiling a program, parallelism is often exposed or detected at one point, but still needed at a later point. In between, a large number of other optimization passes may be run that change the program structure in a way that the original parallelism information is invalid for the transformed program. To still use parallelism information in later passes, it is essential to find a way to provide those passes with up-to-date information.

```
for.body:
  ...
  %val0 = load i32* %idx1, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %val0, i32* %idx2, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %for.end, label %for.body, !llvm.loop !0

for.end:
...
!0 = metadata !{ metadata !0 }
```

Listing 16: A single loop marked parallel using LLVM metadata

```
outer.for.body:
  ...
  %val1 = load i32* %idx3, !llvm.mem.parallel_loop_access !2
  ...
  br label %inner.for.body

inner.for.body:
  ...
  %val0 = load i32* %idx1, !llvm.mem.parallel_loop_access !0
  ...
  store i32 %val0, i32* %idx2, !llvm.mem.parallel_loop_access !0
  ...
  br i1 %exitcond, label %inner.for.end, label %inner.for.body, !llvm.loop !1

inner.for.end:
  ...
  store i32 %val1, i32* %idx4, !llvm.mem.parallel_loop_access !2
  ...
  br i1 %exitcond, label %outer.for.end, label %outer.for.body, !llvm.loop !2

outer.for.end:
...
!0 = metadata !{ metadata !1, metadata !2 } ; list of identifiers
!1 = metadata !{ metadata !1 } ; identifier for the inner loop
!2 = metadata !{ metadata !2 } ; identifier for the outer loop
```

Listing 17: Nested loops marked parallel using LLVM metadata

There are three options how a later pass can get information about possible loop level parallelism: 1) (re)detect parallelism, 2) actively preserve parallelism information in intermediate passes or 3) keep parallelism information as "best-effort". (Re)detecting parallelism requires a full blown dependence and parallelism analysis to be re-

run. Unfortunately, the low-level IR generated after high-level loop transformations is rather difficult to reanalyze. Even if all information could be recovered, the dependence problem that needs to be solved is a lot harder, which means it is rather costly in terms of compile-time. We can avoid these problems, if we manage to preserve the parallelism information that is readily available in our loop transformation framework. The canonical way to preserve information in LLVM is to write an analysis pass that is updated or "preserved" by each optimization. This approach works very well for generic and rather simple information, e.g., the loop structure tree or dominance information. However, teaching low-level passes such as global value numbering about their effects on loop level parallelism is conceptually questionable and, due to the large number of passes that need to be updated, engineering wise expensive. Solution 3) addresses this issue by taking advantage of LLVM's metadata. The basic idea behind this approach is to use IR annotations to mark loops parallel. The key point here is to choose the annotations such that we can easily detect if transformations invalidate parallel execution, without requiring the transformations to actively update any annotations. Passes may still remove the annotations if they are unknown to them, but this at most prevents us from detecting a parallel loop. If such cases appear, we can teach individual transformations to actively preserve parallelism information.

In LLVM we introduced[2] the two metadata types `llvm.loop` and `llvm.mem.parallel_loop_access` which together mark a loop parallel. The `llvm.loop` metadata is attached to the instruction that forms the loop's back edge. It uniquely identifies each loop by referencing a self referencing metadata node. To mark a loop parallel, we add to each memory access in the loop the parallel loop access marker. This marker provides the information that the specific memory access does not inhibit the parallel execution of the loop it references. If all memory accesses within a loop are marked as not inhibiting parallel execution of this loop, the loop is parallel. This is the case right after marking a loop parallel. Subsequent transformations may now introduce new memory accesses which potentially prevent parallel execution. However, the newly introduced memory accesses will not be marked as safe for parallel execution. This can be easily detected and we can conservatively treat the loop as sequential. Listing 16 shows an example[3] where a single loop is marked parallel. The same annotations work also for nested loops. In this case the metadata may reference not only a single loop identifier, but a list of loop identifiers. Listing 17 gives an example containing two nested loops, both marked parallel.

2 The original idea of annotating all memory accesses came from Tobias Grosser, who also reviewed the changes that have been committed by Pekka Jaaskelainend in subversion revision 175060 of LLVM. This feature is available since LLVM 3.3 release.

3 These are modified examples from the LLVM 3.4 language reference.

13 OPTIMISTIC DELINEARIZATION

Dense multi-dimensional arrays are data structures common to many compute problems. To allow compilers to perform interesting optimizations, it is often necessary for the compiler to understand the multi-dimensional nature of these arrays. Unfortunately, while multi-dimensional arrays are native constructs in C99, Fortran or Julia [32], information about their multi-dimensionality is often lost when translating such languages to a low-level compiler IR. In addition, many programming languages (C90, C++) do not natively support variable size multi-dimensional arrays. In C90 or C++, users implement such arrays by creating their own classes, templates or macros leaving the compiler without any information about the possible multi-dimensionality of certain data accesses. As a result, accesses to arrays that are multi-dimensional in nature are seen by the compiler as single-dimensional accesses that directly correspond to how the array is laid out in memory. We call this single-dimensional view the "linearized" view of an array.

Assuming the original index expressions are affine, the linearized accesses can have different properties. For arrays of constant size, linearized expressions will contain larger integer coefficients (the sizes of the array dimensions). Despite the presence of these coefficients the linearized access expression remains affine. We illustrate this for a simple example in Figure 48. As affine expressions can be precisely modeled with integer maps, data flow analysis based on integer maps (e.g., the one provided by isl [128]) will yield optimal results. However, in cases where the size of the array is parametric (Figure 49), this is no longer true. The expressions we obtain by linearizing accesses to arrays of parametric size may now contain multiplications between loop indexes and variables such as $m * i$. Performing dependence analysis on the "linearized" view of an array is a complex problem not supported by most existing dependence analyses.

```
void constantSize(float A[1024][4096]) {
  for (int i = 0; i < 1024; i++)
    for (int j = 0; j < 4096; j++)
      A[i][j]      = i + j;
//    A[4096 i + j] = ... ⇐ expression remains affine
}
```

Figure 48: Linearized expression for multi-dimensional array of constant size

```
void parametricSize(float A[n][m]) {
  for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
      A[i][j]      = i + j;
//    A[m i + j] = ... ⇐ expression is polynomial
}
```

Figure 49: Linearized expression for multi-dimensional array of parametric size

We address this issue by presenting a new approach that for several important cases can derive from a given set of one dimensional multivariate polynomial array accesses an equivalent multi-dimensional view in which all array index expressions are affine. As existing data dependence analyses can precisely analyze the resulting arrays, we obtain precise data dependency information for kernels with such polynomial index expressions.

Compared to previous work in this area [91, 93, 116, 43] our approach is different in two main aspects. First of all, we use an optimistic approach in our delinearization, which allows us to derive valid delinearizations in cases where statically proving the validity of a certain delinearization is not possible. Instead of giving up, we derive in these situations a likely delinearization and provide a set of run-time conditions which can be verified to ensure our assumed delinearization correctly models the memory accesses. The second important difference is that we derive an actual delinearization of the memory reference and do not provide a specialized solution that only addresses the dependency analysis problem. This means that we are able to use the result of our delinearization to model our memory access with multi dimensional integer maps. Such maps can be analyzed with our existing integer set library and we can use the new AST generator (Chapter 10) to automatically generate code for possibly adjusted multi-dimensional memory accesses. This is very handy when aiming for automatic GPU code generation from within a low-level compiler.

```
void gemm(int n, int m, int p,
          float A[n][p], float B[p][m], float C[n][m]) {
L1:  for (int i = 0; i < n; i++)
L2:    for (int j = 0; j < m; j++) {
L3:      for (int k = 0; k < p; ++k)
           C[i][j] += A[i][k] * B[k][j];
     }
}
```

Listing 18: A gemm kernel written in C99 using variable length arrays

13.1 MOTIVATING EXAMPLE

Listing 18 shows a simple gemm kernel implemented with C99 variable length arrays. When compiling the code with clang and analyzing the access to `A` clang's scalar evolution analysis [99] recovers the SCEV expression $\{\{A, +, p\}_{L1}, +, 1\}_{L3}$ from the IR. The semantics of a SCEV expression $\{\text{Base}, +, \text{Inc}\}_{\text{Loop}}$ is such that the expression has the value "Base" at the first iteration of the loop "Loop" and the value of the expression is incremented by "Inc" at each subsequent loop iteration. As a result, the above expression is equivalent to an access function `A[i * p + k]` and shows the perspective of a low-level compiler on this memory access. We note that neither the original dimensionality nor the size of the individual dimensions is preserved.

As stated in the introduction, to recover the multi-dimensional structure of the array access we use an optimistic approach. In case the multi-dimensionality of an access can not be proven statically, we provide conditions to verify the delinearization at run-time. To guide our optimistic delinearization we take advantage of structural information provided by the parametric array sizes. For multi-dimensional arrays with affine access functions we observe that, after linearizing the accesses to such arrays, all parameters that appear within products of loop induction variables and parameters are derived from the sizes of the original array dimensions. In the previous example the only such product is $i * p$ and the contained parameter p directly corresponds to the inner dimension of the array `A`. So we could guess that the original array has been declared as `A[][p]` with access functions `A[i][k]`. To verify our guess, we need to check that all possible uses of this access will remain within the bounds of our assumed array shape. For the inner dimension this means that $\forall i, j, k : 0 \leqslant i < n \wedge 0 \leqslant j < m \wedge 0 \leqslant k < p : 0 \leqslant k < p$ holds. For this trivial example, this condition is statically provable. No run-time check is necessary.

The need for actual run-time checks often arises in existing code due to the fact that the source code does not provide any information on the relation between the sizes of different arrays. Listing 19 shows the very same gemm kernel, but this time implemented with 2D arrays that use dedicated structures to keep the track of the array's size, a style very similar to the implementation of boost::ublas. If we now look at the access to the array B, we get an expression `B->Base[k * B->size1 + j]`. The iteration space constraints that hold are $0 \leqslant k \leqslant$ A->size0 $\wedge\, 0 \leqslant j \leqslant$ C->size1. Showing from this information that for all possible values of j, the access to the inner dimension of B remains within bounds ($\forall j : 0 \leqslant j \leqslant$ B->size1) is not possible. Instead we require a run-time condition B->size1 $\geqslant$ A->size0 to avoid out of bounds accesses. Only in case B is large enough our delinearization models the actual run-time behavior correctly. Even though this

example looks rather contrived and one could assume that using this function with matrices `B` that have a smaller size is possibly not intended, this is in fact a very realistic example. Compilers are required to preserve the semantics of code even in unlikely, but well defined situations. As doing so commonly inhibits most useful optimizations the use of an optimistic approach which enables us to perform useful optimizations while still being able to preserve full program semantics is important.

```
struct 2DArray {
        size_t size0;
        size_t size1;
        float *Base;
}

#define ACCESS_2D(A, x, y) *(A->Base + (y) * A->size1 + (x))
#define SIZE0_2D(A) A->size0
#define SIZE1_2D(A) A->size1

void gemm(struct 2DArray *A, struct 2DArray *B,
          struct 2DArray *C) {
L1:  for (int i = 0; i < SIZE0_2D(C); i++)
L2:    for (int j = 0; j < SIZE1_2D(C); j++) {
L3:      for (int k = 0; k < SIZE0_2D(A); ++k)
           ACCESS_2D(C, i, j) += ACCESS_2D(A, i, k) *
                                 ACCESS_2D(B, k, j);
     }
}
```

Listing 19: A gemm kernel written using manually implemented multi-dimensional arrays.

13.2 PROBLEM STATEMENT

We will now state the problem we address more formally:

Given a set of single dimensional memory accesses with index expressions that are multivariate polynomials in terms of loop iterators as well as symbolic program parameters and a set of corresponding iteration domains, derive a multi-dimensional view of this array such that all index expressions are linear.

The multi-dimensional view we derive consists of:

- A multi-dimensional array definition, including:
  - The number of array dimensions

  - Sizes for all but the outermost dimension

- For each original array access, a corresponding multi-dimensional access

We also pose a set of additional requirements on the view we derive:

- The number of array dimensions is minimal. (R1)
- The array sizes are minimal. (R2)
- The new access functions are affine in loop parameters and program parameters. (R3)
- For each array access, the memory location directly obtained from the linearized subscript expression and the memory location obtained from the multi-dimensional array after lowering it using the derived array sizes and assuming a row-major array layout are identical for all loop iterators within the iteration space. (R4)
- The array subscript expressions for all but the outermost dimension are for all iterations within the iteration space within the bounds of the multi-dimensional array (R5)

For cases where a multi-dimensional array view can not be proven correct statically, derive a multi-dimensional view as discussed above and provide a set of conditions under which this view is valid.

Requirements R1 and R2 are there to ensure that no unnecessarily complicated array views are computed. There is no point in deriving array accesses of the form `A[1][1][1][n][m]` where a leading set of dimensions is always identical and consequently does not provide any interesting information. Similarly, in case there is freedom in the array sizes that can be chosen, we want to obtain the minimal array size on each dimension (prioritizing inner dimension).

Requirement R3 is necessary to ensure that we can represent the resulting access expressions as integer maps.

Requirement R4 ensures that the multi-dimensional form of the array has the same access characteristics as the single dimensional array. Ensuring the same access characteristics enables us to use the multi-dimensional view not only for dependence analysis but also for reasoning about data-locality (e.g., to find stride one accesses).

Requirement R5 ensures together with R4 that if we define a relation R between the elements of the linearized and the delinearized array, such that two elements are related iff they map to the same data-location, this relation is always bijective. This property is important as it ensures that for each actual memory location there is only

a single data location in our model, which again is necessary for the correct computation of data dependences.

13.3 ARRAY VIEWS WITH SINGLE-PARAMETER SIZES

In this section we present an algorithmic approach to derive a delinearization from a multivariate polynomial to a multi-dimensional array of the shape $A[p_0][p_1]\dots[p_{n-1}]$, $p_i \in \mathcal{P}$ with $\mathcal{P}$ referring to the set of program parameters. This means we obtain array shapes with the size of each dimension being defined by a single parameter and with multiple dimensions possibly sharing the same parameter. Arrays of such shape are common and appear e.g., in the Julia [32] code, boost::ublas as well as the Himeno benchmark [69]. The example in Listing 19 also defines such an array.

The algorithm we propose consists of the following four steps:

1. Collect possible array size parameters
2. Derive dimensionality and array size
3. Compute multi-dimensional access functions
4. Derive validity conditions

As a first step, we collect information about possible array size parameters. To do this we expand the given polynomial expression into a sum of products. From this sum, we extract all terms that contain both a loop induction variable and (possibly multiple) parameters. Those terms are interesting as the presence of a term that multiplies a parameter with a loop induction variable makes the expression non-affine. However, in case a parameter p is an array size parameter, p may be removed from the index expressions during delinearization such that the original expression is turned into an access with affine subscript expressions. Consequently, we guess that p defines the size of at least one array dimension.

As the second step, we derive the dimensionality and the size of the array. To do this we start from the terms obtained in the previous step and assume all of them form products. In case a term is not a product, we treat it as a product with just a single factor. We remove from each term all factors that are non-parametric. The resulting terms are sorted according to the number of factors they have and we check that the terms with less factors symbolically divide the larger terms. In case this is true we assume the results of these divisions are the array sizes.

As the third step, we extract the access functions of the individual dimensions. For this we start with the original polynomial expression given and first divide it by the size of the elements accessed. The resulting expression is then divided by the assumed array sizes starting

with the innermost size. The remainder is the access function of the innermost dimension, the quotient is divided again by the size of the next array dimension. The new remainder is the access function of the second array dimension and the quotient is divided further. If no more array sizes are available, the last quotient becomes the access function of the outermost dimension.

As a last step, we derive the validity conditions. Up to this step, the delinearization we propose is an educated guess. It is only valid if $\forall i \in [1, n-1] : 0 \leqslant f_i < d_i$ holds, with n being the number of array dimensions computed, f_i being the access function of dimension i and d_i being the size of dimension i. To check if these conditions hold, we can simplify them taking into account the range of the surrounding loop induction variables. In simple cases this simplification yields $\top$, which means the delinearization has been statically proven correct. In cases where this is not enough, the remaining conditions need to be emitted as run-time checks.

We illustrate the full algorithm on a more complex example, the initialization of a subset of a multi-dimensional array:

```
/// @param n, m, p: The array sizes.
/// @param A: The full array.
/// @param o1, o2, o3: The offset of the subarray.
/// @param s1, s2, s3: The size of the subarray.
void set_subarray(int n, int m, int p, float A[n][m][p],
                  int o0, int o1, int o2,
                  int s0, int s1, int s2) {
L1:  for (int i = 0; i < s0; i++)
L2:    for (int j = 0; j < s1; j++)
L3:      for (int k = 0; k < s2; k++)
           A[i+o0][j+o1][k+o2] = 1.0;
}
```

Start:
$4(p(mo_0 + o_1) + o_2) + 4mpi + 4pj + 4k$

Expand the expression
$4pmo_0 + 4po_1 + 4o_2 + 4mpi + 4pj + 4k$

Extract terms containing loop induction variables
$\{4mpi, 4pj, 4\}$

Remove non-parametric components, sort terms, derive sizes :

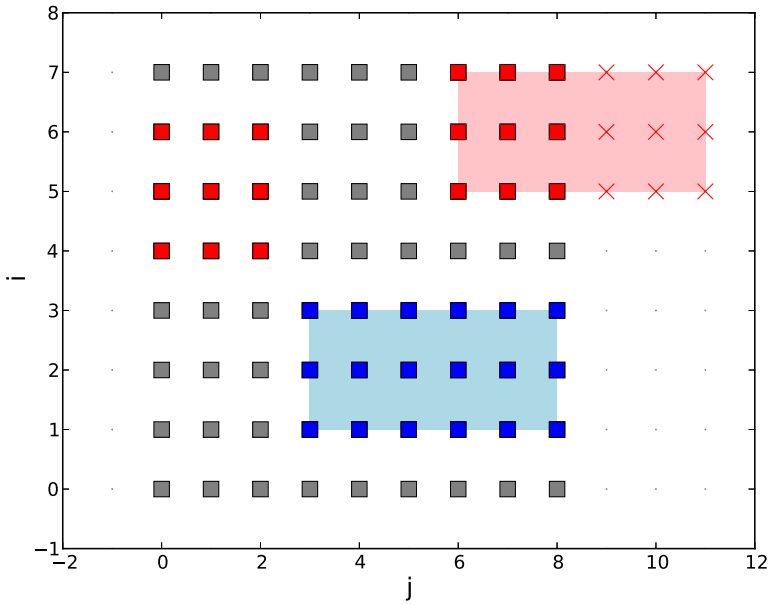


Figure 50: Subarrays accesses for different parameter values

$\{mp, p\} \rightarrow$ `A[][m][p]`

Divide by element size: $\text{sizeof}(\text{float}) = 4$
Quotient: $pmo_0 + po_1 + o_2 + mpi + pj + k$
Remainder: 0

Divide by inner dimension: p
Quotient: $mo_0 + o_1 + mi + j$
Remainder: $o_2 + k \rightarrow A[?][?][k + o_2]$

Divide by second inner dimension: m
Quotient: $o_0 + i \rightarrow A[i + o_0][?][?]$
Remainder: $o_1 + j \rightarrow A[?][j + o_1][?]$

Reconstruct the full array access:
$A[i + o_0][j + o_1][k + o_2]$

Derive the validity conditions:

$$\begin{aligned} \forall i, j, k: \; & 0 \leqslant i < s_0 \wedge 0 \leqslant j < s_1 \wedge 0 \leqslant k < s_2 : \\ & 0 \leqslant k + o_2 < p \wedge 0 \leqslant j + o_1 < m \wedge 0 \leqslant i + o_0 \\ \text{Assuming:} \; & s_0, s_1, s_2, o_0, o_1, o_2 > 0 \\ & \Rightarrow o_1 \leqslant m - s_1 \wedge o_2 \leqslant p - s_2 \end{aligned}$$

For the above example, the validity conditions can not be statically evaluated, but need to be verified at run-time. Figure 50 uses a two

dimensional version of this example ($s0 = n = 1$) to illustrate two sets of parameter values, one that satisfies the validity condition and one that does not. Both examples work on a 2D data array `A[m][p]` with $m = 8 \wedge p = 9$. The first set of parameter values is $o1 = 1 \wedge o2 = 3 \wedge s2 = 3 \wedge s3 = 6$, which yields $1 \leqslant 8 - 3 \wedge 3 \leqslant 9 - 6$ and evaluates to $\top$. The corresponding set of data elements (illustrated in blue) are all within the bounds of the 2D array. However, of the accesses that correspond to the parameter values $o1 = 5 \wedge o2 = 6 \wedge s2 = 3 \wedge s3 = 6$ (red square) only the left half is within the array bounds. The right half accesses are out-of-bounds. In this case, the out-of-bounds accesses access data-locations that correspond to the array elements $\{A[i, j] : 4 \leqslant i \leqslant 6 \wedge 0 \leqslant j \leqslant 2\}$ (red squares). This is problematic, as e.g., the data stored to `A[7][9]` affects the values read from `A[6][0]`. This relation is not visible in the delinerized program, which means the corresponding data dependences are not modeled and certain program transformations may be performed incorrectly. When checking our validity conditions we see that $o2 \leqslant p - s2 \Rightarrow 6 \leqslant 9 - 6 \Rightarrow \bot$, which correctly shows that for this set of parameters we can not rely on our delinearization.

13.3.1 *Multiple array references*

In case the piece of the program where we would like to apply delinerization contains more than one access to the same array, it is important to ensure that all accesses are delinerized using the same assumed array declaration. Ensuring this requires only a slight adjustment of our algorithm. In the case of multiple arrays, we extract the terms from all arrays and derive the assumed array size from the combined terms. Using this common array size, we can then again derive the array accesses individually. The validity conditions are also derived individually, but redundant conditions are removed in a subsequent step.

```
for (i = 0; i < p; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < m; k++)
S1:   A[i][j][k] = 1;

S2: A[1][1][1] = 0;
S3: A[0][0][M-1] = 0;
S4: A[0][N-1][0] = 0;
S5: A[0][N-1][M-1] = 0;
```

Listing 20: Array dimensions used in subscripts

13.3.2 *Array sizes in subscript expressions*

For cases where the subscript expressions of a delinearized array contain the array size itself as shown in Listing 20, the previously presented algorithm derives for an access `A[0][M-1]` with array size `M`, the access `A[1][-1]` as it associates multiples of `M` with the outer dimension. This delinearization is invalid, as the subscript in the inner dimension becomes negative.

In general there is always a set of delinearizations that differ in their derived subscript expressions, but which all compute the same address expression. For the above example, the accesses `A[-1][2*M-1]`, `A[0][M-1]`, `A[1][-1]` and `A[2][-M-1]` all compute the same address expression. In fact, for an arbitrary $k \in \mathcal{N}$ a different, but equivalent, expression can be computed from an existing expression:

$$
\begin{aligned}
& A[f_0][f_1] && \text{with } A[*][s_1] \\
=\ & A[f_0 s_1 + f_1] && \text{with } A[*] \\
=\ & A[f_0 s_1 - k s_1 + k s_1 + f_1] && \text{with } A[*] \\
=\ & A[(f_0 - k) s_1 + (k s_1 + f_1)] && \text{with } A[*] \\
=\ & A[f_0 - k][k s_1 + f_1] && \text{with } A[*][s_1]
\end{aligned}
$$

For d-dimensional accesses we can say that for any pair of neighboring dimensions $t, t+1$ with $t \in [0, d-2]$ the following equality holds for all $k_t \in \mathcal{N}$:

$$\&A[\ldots, f_t, f_{t+1}, \ldots] = \&A[\ldots, f_t - k_t, k_t s_{t+1} + f_{t+1}, \ldots]$$

This means there exists a set of values $k_t, t \in [0, d-2]$ which can be arbitrarily chosen to generate an infinite number of array accesses that all yield the same address expressions. However, for a specific set of loop iterators and parameters all but a single set of k_t values cause out of bound accesses. The question is how to find the right values of k_t, that avoid out of bounds accesses? One idea is to look at the loop bounds and to statically derive the right values of k_t. This is possible as long as the range of the subscript expression is known, but causes problems for accesses such as `A[N * i + N + p]` which might be modeled as `A[i + 1][p-1]` in case $0 \leqslant p - 1 < N$ holds, but where the right value of k can not be statically derived without knowledge about the values p can take. An alternative is to create a piecewise delinearization that chooses the correct value of k depending on the values of the subscript expressions. For the 2D case we could be tempted to use a mapping $(f_0, f_1) \to (f_0 + k, -k s_1 + f_1) : \exists k : k s_1 \leqslant f_1 < (k+1) s_1$, which models all possible values of k. Unfortunately, the product between k and s_1 is non-affine and consequently this map can not be represented as an integer map. However,

if we bound k such that $k \in [k_l, k_u]$ with k_l, k_u being known integer values, we can model this map with a finite number of affine pieces.

$$(f_0, f_1) \rightarrow \begin{cases} (f_0 - k_l, k_l s_1 + f_2) & f_1 < -k_l s_1 \\ \vdots & \\ (f_0 - 1, s_1 + f_2) & -s_1 \leqslant f_1 < 0 \\ (f_0, f_1) & 0 \leqslant f_1 < s_1 \\ (f_0 + 1, -s_1 + f_2) & s_1 \leqslant f_1 < 2s_1 \\ \vdots & \\ (f_0 + k_u, -k_u s_1 + f_2) & k_u s_1 \leqslant f_1 \end{cases}$$

Similarly, for d-dimensional accesses we can define set of maps $M_t, t \in [0, d-2]$:

$$M_t = (f_0, \ldots, f_t, f_{t+1}, \ldots, f_{d-1}) \rightarrow \begin{cases} (f_0, \ldots, f_t - k_{t,l}, k_{t,l} s_{t+1} + f_{t+1}, \ldots, f_{d-1}) & f_{t+1} < -k_{t,l} s_{t+1} \\ \vdots & \\ (f_0, \ldots, f_t - 1, s_{t+1} + f_{t+1}, \ldots, f_{d-1}) & -s_{t+1} \leqslant f_{t+1} < 0 \\ (f_0, \ldots, f_t, f_{t+1}, \ldots, f_{d-1}) & 0 \leqslant f_{t+1} < s_{t+1} \\ (f_0, \ldots, f_t + 1, -s_{t+1} f_{t+1}, \ldots, f_{d-1}) & s_{t+1} \leqslant f_{t+1} < 2s_{t+1} \\ \vdots & \\ (f_0, \ldots, f_t + k_{t,u}, -k_{t,u} s_{t+1} + f_{t+1}, \ldots, f_{d-1}) & -k_{t,u} s_{t+1} \leqslant f_{t+1} \end{cases}$$

with $f_i, i \in [0, d-1]$ the subscript expression at depth i and $s_i, i \in [0, d-1]$ the size of the i-th dimension. Starting from the highest t, we apply all maps M_t one by one to the delinearized accesses. For the example in Listing 20 we obtain the following delinearized accesses from the original algorithm:

$$\begin{aligned} S1(i, j, k) &\rightarrow A(i, j, k) \\ S2(\,) &\rightarrow A(1, 1, 1) \\ S3(\,) &\rightarrow A(0, 1, -1) \\ S4(\,) &\rightarrow A(1, -1, 0) \\ S5(\,) &\rightarrow A(1, 0, -1) \end{aligned}$$

After applying a set of maps M_t generated with values $k_{t,k} = 0, k_{t,u} = 0$ chosen to only cover two cases, one with no transformation and one

with a single multiple of the problem size parameter added, we obtain the following delinearized accesses:

$$S1(i,j,k) \to \begin{cases} A(i-1, N+j-1, M+k) : k \leqslant -1 \wedge j \leqslant 0 \\ A(i, j-1, M+k) : k \leqslant -1 \wedge j \geqslant 1 \\ A(i-1, N+j, k) : k \geqslant 0 \wedge j \leqslant -1 \\ A(i,j,k) : k \geqslant 0 \wedge j \geqslant 0 \end{cases}$$

$$S2(\,) \to A(1,1,1)$$
$$S3(\,) \to A(0,0,M-1)$$
$$S4(\,) \to A(0,N-1,0)$$
$$S5(\,) \to A(0,N-1,M-1)$$

S2, S3, S4 and S5 show directly the correct delinearization. The access function for S1 is now slightly more complicated, but the three additional cases only apply under conditions that are removed when simplifying the access under the constraints implied by the iteration domain of S1. After these simplifications we obtain for S1 the mapping $S1(i,j,k) \to A(i,j,k)$. So the piecewise mappings have all been statically reduced to maps with just a single piece.

13.3.3 *Arrays of size* $A[*][\beta_1 P_1][\beta_2 P_2]$

In certain cases (e.g. resizing of images) we may have array sizes of the form $A[*][\beta_1 P_1][\beta_2 P_2]$. Accesses to such arrays would be delinearized to an access $A[\beta_1 f_0][\beta_2 f_1][f_2]$ into an array of size $A[*][P_1][P_2]$. As f_1 can be in the range $0 \leqslant f_1 < \beta_1 P_1$, the expression $\beta_2 f_1$ may not fit into the new range. To address this we can find the gcd of the values in each dimension and use it to adjust the array sizes. Specifically, if all subscript expressions on a certain dimension can be divided by a value x, we can divide all of them by x and multiply the size of the next innermost dimension by x. This transformation is always positive in the sense that it only increases the chance that our delinearization will be correct. As it reduces the range of the subscript expression, the subscript expression is more likely to fit into the ranges implied by the array size. Similarly, as we increase the size of the inner dimension the corresponding subscript expressions on this dimension are also more likely to fit in.

13.4 ARRAYS OF SIZE "PARAMETER + CONSTANT"

As most implementations of multi-dimensional arrays choose to keep the sizes of individual dimensions in per-dimension variables, even multi-dimensional arrays with dimension sizes derived from possibly complex expressions often express the array size as an individual

parameter such that resulting expressions can be delinearized with the approach presented in Section 13.3. However, in certain cases, the most prominent one being C99 arrays, a compiler may have access to the full original array declaration. Even though additional information is in general positive, in this specific case it allows the compiler to mix array size information with index expression information such that delinearizing these expressions becomes a lot more challenging.

We look now at a specific case, where the shape of the array is of the form $A[P_0 + \alpha_0] \dots [P_{n-1} + \alpha_{n-1}]$, $P_i \in \mathcal{P}$, $\alpha \in \mathbb{N}$, with P_i being different for different values of i. As an example we show a simplified 3D stencil computation which computes the average over the elements in a diagonal stencil and which uses a one element border around the actual data elements to avoid the need for special boundary statements.

```
long In[Q+2][R+2][S+2];
long Out[O+2][R+2][S+2];

for (long i = 1; i <= Q; i++)
  for (long i = 1; i <= R; i++) {
    for (long i = 1; i <= S; i++) {
      Out[i][j][k] = 0.33f * (In[i][j][k]
                              + In[i+1][j+1][k+1]
                              + In[i-1][j-1][k-1]);
    }
```

When compiling the access `Out[i][j][k]` and delinearizing the access expression with the approach presented in Section 13.3 we perform the following computation:

$$
\begin{aligned}
& A[i][j][k] && \text{with } A[*][R+2][S+2] && \\
= {} & A[i(R+2)(S+2)+j(S+2)+k] && \text{with } A[*] && \leftarrow \text{linearize} \\
= {} & A[4i+2j+k+2iR+2iS+jS+iRS] && \text{with } A[*] && \leftarrow \text{expand} \\
= {} & A[RS*i+S*(2i+j)+4i+2j+k+2iR] && \text{with } A[*] && \leftarrow \text{sort assuming A[][R][S]} \\
= {} & A[i][2i+j][4i+2j+k+2iR] && \text{with } A[*][R][S] && \leftarrow \text{delinearize}
\end{aligned}
$$

There are two problems. First of all, the previous algorithm fails to guess an array size, as the terms R, S and RS all appear in products that contain induction variables and our previous approach can consequently not define an order on the parameters that allow it to assign parameters to array dimensions. Even if we "guess" the right parameter ordering, the access functions are still derived according to the wrong array sizes. As a result we obtain not only incorrect subscript expressions, but we also obtain subscript expressions that contain non-affine expressions such as $2iR$.

We now present a general approach that allows us to delinearize polynomial expressions to d-dimensional array shapes of the form

$A[P_0 + \alpha_0] \dots [P_{d-1} + \alpha_{d-1}]$, $P_i \in \mathcal{P}$, $\alpha \in \mathbb{N}$, we look at the two and three dimensional special cases.

An access to a two dimensional array $A[f_0(\vec{i})][f_1(\vec{i})]$ with shape $A[*][P_1 + \alpha_1]$ corresponds to the single dimensional access $A[f_0(\vec{i})(P_1 + \alpha_1) + f_1(\vec{i})]$, which after expansion becomes $A[f_0(\vec{i})P_1 + f_0(\vec{i})\alpha_1 + f_1(\vec{i})]$. However, it is unlikely that this structure is preserved. The only structure that can be assumed are different terms, each containing a different set of parameters $g_{\{1\}}(\vec{i})P_1 + g_\emptyset(\vec{i})$. To delinearize this polynomial expression we need to recover expressions $f_0(\vec{i}), f_1(\vec{i}), \alpha_1$ as a function of g_i's. As $f_0(\vec{i})$ is the only coefficient to P_1, recovering the relation $f_0(\vec{i}) = g_{\{1\}}(\vec{i})$ is easy. The second equality we can obtain is $g_\emptyset(\vec{i}) = f_0(\vec{i})\alpha_1 + f_1(\vec{i})$. With $f_0(\vec{i})$ plugged in we obtain $g_\emptyset(\vec{i}) = g_{\{1\}}(\vec{i})\alpha_1 + f_1(\vec{i})$, which allows us to express $f_1(\vec{i})$ as a function of α_1: $f_1(\vec{i}) = g_\emptyset(\vec{i}) - g_{\{1\}}(\vec{i})\alpha_1$. For different values of α_1 we obtain different array sizes and the corresponding delinerizations, which all are lowered to the very same linearized function, perform the same memory accesses and consequently model the program behavior correctly. However, depending on the iteration space boundaries only certain delinearizations ensure the absence of out of bounds accesses. As boundary offsets are commonly small and there is only one value α_1 to verify, it is possible to scan a certain number of α_1 by either statically checking for valid delinearizations or possibly even by generating run-time versioned code for different values of α_1.

Looking at the three dimensional case, we observe that an access $A[f_0(\vec{i})][f_1(\vec{i})][f_2(\vec{i})]$ to an array of shape $A[*][P_1 + \alpha_1][P_2 + \alpha_2]$ is linearized to

$$f_0(\vec{i})(P_1 + \alpha_1)(P_2 + \alpha_2) + f_1(\vec{i})(P_2 + \alpha_2) + f_2(\vec{i})$$

which after expansion yields:

$$f_0(\vec{i})P_1P_2 + f_0(\vec{i})P_1\alpha_2 + f_0(\vec{i})P_2\alpha_1 + f_0(\vec{i})\alpha_1\alpha_2 +$$
$$f_1(\vec{i})P_2 + f_1(\vec{i})\alpha_2 +$$
$$f_2(\vec{i})$$

The corresponding polynomial expression is:

$$g_{\{1,2\}}(\vec{i})P_1P_2 + g_{\{P_1\}}(\vec{i})P_1 + g_{\{P_2\}}(\vec{i})P_2 + g_\emptyset(\vec{i})$$

From the single term that contains P_1P_2, the product of all symbolic parameters defining the array sizes, we recover $f_0(\vec{i}) = g_{\{1,2\}}(\vec{i})$. Assuming P_1 is the outermost parameter, we obtain the value of α_2 from the single term that contains P_1, but not P_2: $g_{\{1\}}(\vec{i}) = f_0(\vec{i})\alpha_2 \Rightarrow \alpha_2 = g_{\{1\}}(\vec{i})/f_0(\vec{i}) = g_{\{1\}}(\vec{i})/g_{\{1,2\}}(\vec{i})$. Looking at the P_2 terms, we obtain the relation $g_{\{2\}}(\vec{i}) = f_0(\vec{i})\alpha_1 + f_1(\vec{i})$. This allows us to derive $f_1(\vec{i}) = g_{\{2\}}(\vec{i}) - f_0(\vec{i})\alpha_1 = g_{\{2\}}(\vec{i}) - g_{\{1,2\}}(\vec{i})\alpha_1$. Again, an expression containing α_1 as free variable. To obtain $f_2(\vec{i})$ we look at the terms without

Algorithm 1: Derive a delinearization

Data: A polynomial expression in function of induction variables and parameters, a list of array size parameters
Result: A set of values $\alpha_k, k \in [1, d-1]$, index expressions $f_k, k \in [0, d-1]$ and set of array size parameters $P_k, k \in [1, d-1]$ or an error if no delinearization found.

```
collect possible array sizes parameters;
foreach permutation of array sizes parameters do
    derive f_0;
    alpha = derive alpha values;
    if alpha ≠ [ ] then
        derive subscript expressions;
        derive run-time condition;
        if run-time condition is a contradiction then
            continue;
        else
            return subscript expressions, run-time-condition,
            array-sizes
return No delinearization found!
```

any parameters. Here we have $g_\emptyset(\vec{i}) = f_0(\vec{i})\alpha_1\alpha_2 + f_1(\vec{i})\alpha_2 + f_2(\vec{i})$ from which we can derive $f_2(\vec{i}) = g_\emptyset(\vec{i}) - f_0(\vec{i})\alpha_1\alpha_2 i - f_1(\vec{i})\alpha_2 = g_\emptyset(\vec{i}) - f_0(\vec{i})\alpha_1\alpha_2 - (g_{\{2\}}(\vec{i}) - f_0(\vec{i})\alpha_1)\alpha_2 = g_\emptyset(\vec{i}) - f_0(\vec{i})\alpha_1\alpha_2 - g_{\{2\}}(\vec{i})\alpha_2 + f_0(\vec{i})\alpha_1\alpha_2 = g_\emptyset(\vec{i}) - g_{\{2\}}(\vec{i})\alpha_2$. As α_1 cancels out, we can unambiguously derive $f_2(\vec{i})$. We can conclude that delinearizing to a three-dimensional array shape does not introduce more freedom. Only α_1 remains unknown and different values may need to be explored.

After having understood the basic approach, we now present a general algorithm to delinearize polynomial expressions to array shapes of arbitrary dimensionality. Our high-level algorithm Algorithm 1 is rather straightforward. We first collect the set of possible array size parameters and then try for each order to find a valid delinearization. To check if a valid delinearization exists, we first compute $f_0(\vec{i})$ and use it to try to derive a set of consistent α values. If we succeed, we derive subscript expressions and run-time conditions. In case the run-time condition is not a contradiction, we assume we found a valid delinearization and finish, otherwise we try the next permutation.

To obtain the set of possible array size parameters, we take the expanded version of the polynomial expression and look again for parameters that are multiplied with a loop induction variable.

For the remaining analysis it is necessary to understand the shape of the polynomial expression we analyze. Specifically, that we can group it such that each term is the product between a subset of the

suspected array size parameters and an expression $g_?(\vec{i})$ in loop indexes, non array size parameters and integer constants:

$$
\begin{aligned}
& g_\emptyset(\vec{i}) \\
+\ & g_{\{1\}}(\vec{i})P_1 + g_{\{2\}}(\vec{i})P_2 + \cdots + g_{\{d-1\}}(\vec{i})P_{d-1} \\
+\ & g_{\{1,2\}}(\vec{i})P_1P_2 + g_{\{1,3\}}(\vec{i})P_1P_3 + \cdots + g_{\{2,3\}}(\vec{i})P_2P_3 + \ldots \\
+\ & g_{[1,d-1]}(\vec{i})P_1 \ldots P_{d-1} \\
=\ & \sum_{K \in \mathcal{P}([1,d-1])} \left(g_K(\vec{i}) \prod_{k \in K} P_k \right)
\end{aligned}
$$

We now want to express the previous polynomial as a d-dimensional access $A[f_0(\vec{i})] \ldots [f_{d-1}(\vec{i})]$ to an array of size $A[*][P_1 + \alpha_1] \ldots [P_{d-1} + \alpha_{d-1}]$. To do so, we start by looking at how such an array is linearized:

$$
\begin{aligned}
& f_0(\vec{i})(P_1 + \alpha_1)(P_2 + \alpha_2) \ldots (P_{d-1} + \alpha_{d-1}) \\
+\ & f_1(\vec{i})(P_2 + \alpha_2) \ldots (P_{d-1} + \alpha_{d-1}) \\
+\ & f_2(\vec{i})(P_3 + \alpha_3) \ldots (P_{d-1} + \alpha_{d-1}) \\
& \vdots \\
+\ & f_{d-2}(\vec{i})(P_{d-1} + \alpha_{d-1}) \\
+\ & f_{d-1}(\vec{i}) \\
=\ & \sum_{j \in [0,d-1]} \left(f_j(\vec{i}) \prod_{k \in [j+1,d-1]} (P_k + \alpha_k) \right)
\end{aligned}
$$

and assume this linearized form yields the same access computation as the one-dimensional expression we want to delinearize:

$$
\begin{aligned}
\sum_{K \in \mathcal{P}([1,d-1])} \left(g_K(\vec{i}) \prod_{k \in K} P_k \right) &= \sum_{j \in [0,d-1]} \left(f_j(\vec{i}) \prod_{k \in [j+1,d-1]} (P_k + \alpha_k) \right) \\
&= \sum_{j \in [0,d-1]} \sum_{K \in \mathcal{P}([j+1,d-1])} \left(f_j(\vec{i}) \prod_{k \in K} P_k \prod_{k \in [j+1,d-1] \setminus K} \alpha_k \right)
\end{aligned}
$$

We now match up terms that contain the same set of parameters:

$$
\forall K \in \mathcal{P}([1, d-1]) :
$$

$$
g_K(\vec{i}) \prod_{k \in K} P_k = \sum_{\substack{j \in [0,d-1] \\ \wedge K \subseteq [j+1,d-1]}} \left(f_j(\vec{i}) \prod_{k \in K} P_k \prod_{k \in [j+1,d-1] \setminus K} \alpha_k \right)
$$

Assuming the parameters to be positive we can drop them on both sides of the equation:

$$
\forall K \in \mathcal{P}([1, d-1]) :
$$

$$
g_K(\vec{i}) = \sum_{\substack{j \in [0,d-1] \\ \wedge K \subseteq [j+1,d-1]}} \left(f_j(\vec{i}) \prod_{k \in [j+1,d-1] \setminus K} \alpha_k \right)
$$

Algorithm 2: Derive alpha values

Data: A dimensionality d, a set of expressions $g_s(\vec{i})$
Result: A list of values $\alpha_k, k \in [2, d-1]$ or [] in case of inconsistencies

```
foreach k ∈ [2, d − 1] do
    if g_[1,d−1] not evenly divides g_[1,d−1]\{k}(i⃗) then
        return [];
    α_k = g_[1,d−1]\{k}(i⃗)/g_[1,d−1](i⃗);
    foreach S ∈ P([2, k − 1] \ (∅ ∪ ([1, d − 1] \ {k}))) do
        if g_[1,d−1](i⃗) not evenly divides S then
            return [];
        α′_k = g_S(i⃗)/g_[1,d−1](i⃗);
        if α′_k ≠ α_k then
            return [];
return {k → α_k : k ∈ [2, d − 1]}
```

Having established this set of equalities, we start to relate our terms $g_?(\vec{i})$ to the terms $f_?(\vec{i})$ and $\alpha_?$ that we want to derive. We first derive $f_0(\vec{i}) = g_{[1,d-1]}(\vec{i})$, which can be trivially derived by setting $K = [1, d-1]$ in the previous equation.

Then, we derive the values α_k by looking at the terms $g_{[1,d-1]\setminus\{k\}}(\vec{i})$. In the four-dimensional case such terms have the form:

$$
\begin{aligned}
&g_{\{2,3,4\}}(\vec{i}) = \alpha_1 f_0(\vec{i}) + f_1(\vec{i}) \\
&g_{\{1,3,4\}}(\vec{i}) = \alpha_2 f_0(\vec{i}) && \Rightarrow \alpha_2 = g_{\{1,3,4\}}(\vec{i})/g_{[1,d-1]}(\vec{i}) \\
&g_{\{1,2,4\}}(\vec{i}) = \alpha_3 f_0(\vec{i}) && \Rightarrow \alpha_3 = g_{\{1,2,4\}}(\vec{i})/g_{[1,d-1]}(\vec{i}) \\
&g_{\{1,2,3\}}(\vec{i}) = \alpha_4 f_0(\vec{i}) && \Rightarrow \alpha_4 = g_{\{1,2,3\}}(\vec{i})/g_{[1,d-1]}(\vec{i})
\end{aligned}
$$

In general, we derive $\alpha_k, k \in [2, d-1]$ as $\alpha_k = g_{[1,d-1]\setminus\{k\}}(\vec{i})/g_{[1,d-1]}(\vec{i})$. Similiar to the two and three dimensional case, we can not derive a value for α_1, as we do not know the value of $f_1(\vec{i})$. However, for higher dimensional cases we can make an interesting observation. The values of α_k can not just be obtained by the equalities presented above. In fact, there is a larger set of equalities that all need to return the same values α_k for an array view to be a valid delinearization. Specifically, to derive $\alpha_k, k \in [1, d-1]$ we can choose any pair of sets

$S, \emptyset \subset S \subseteq [1, k-1]$ and $T = S \cap k$ which can be used to compute α_k as follows:

$$
\begin{aligned}
& g_S(\vec{i})/g_T(\vec{i}) \\
&= \sum_{\substack{j \in [0,d-1] \\ \wedge S \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \prod_{x \in [j+1,d-1] \setminus S} \alpha_x \Big) \Big/ \sum_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \prod_{x \in [j+1,d-1] \setminus T} \alpha_x \Big) \\
&= \sum_{\substack{j \in [0,d-1] \\ \wedge S \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \alpha_k \prod_{x \in [j+1,d-1] \setminus T} \alpha_x \Big) \Big/ \sum_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \prod_{x \in [j+1,d-1] \setminus T} \alpha_x \Big) \\
&= \alpha_k \sum_{\substack{j \in [0,d-1] \\ \wedge S \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \prod_{x \in [j+1,d-1] \setminus T} \alpha_x \Big) \Big/ \sum_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \prod_{x \in [j+1,d-1] \setminus T} \alpha_x \Big) \\
&= \alpha_k \sum_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \prod_{x \in [j+1,d-1] \setminus T} \alpha_x \Big) \Big/ \sum_{\substack{j \in [0,d-1] \\ \wedge T \subseteq [j+1,d-1]}} \Big(f_j(\vec{i}) \prod_{x \in [j+1,d-1] \setminus T} \alpha_x \Big) \\
&= \alpha_k
\end{aligned}
$$

The following lists the closed form expressions that compute α_2 and α_3 for the 4D case:

$$
\begin{aligned}
\alpha_2 &= g_{\{1,3\}}(\vec{i})/g_{\{1,2,3\}}(\vec{i}) \\
\alpha_3 &= g_{\{1\}}(\vec{i})/g_{\{1,3\}}(\vec{i}) \\
\alpha_3 &= g_{\{2\}}(\vec{i})/g_{\{2,3\}}(\vec{i}) \\
\alpha_3 &= g_{\{1,2\}}(\vec{i})/g_{\{1,2,3\}}(\vec{i})
\end{aligned}
$$

As the different values of α_k are overdefined, we can use this information to cross check our delinearization. Specifically, we can use it to validate the order of the array size parameters. Algorithm 2 gives the full algorithm we use to obtain the different alpha values.

After having derived the different values of α_k, we can now derive the terms $f_j, j \in [2, d-1]$ by looking at the terms $g_{[j+1,d-1]}(\vec{i})$:

$$
\begin{aligned}
g_{\{1,\ldots,d-1\}}(\vec{i}) &= f_0(\vec{i}) \\
g_{\{2,\ldots,d-1\}}(\vec{i}) &= \alpha_1 f_0(\vec{i}) + f_1(\vec{i}) \\
g_{\{3,\ldots,d-1\}}(\vec{i}) &= \alpha_1 \alpha_2 f_0(\vec{i}) + \alpha_2 f_1(\vec{i}) + f_2(\vec{i}) \\
&= \alpha_2 (\alpha_1 f_0(\vec{i}) + f_1(\vec{i})) + f_2(\vec{i}) \\
&= \alpha_2 g_{\{2,\ldots,d-1\}}(\vec{i}) + f_2(\vec{i}) \\
&\vdots \\
g_{\{j,\ldots,d-1\}}(\vec{i}) &= \alpha_{j-1} g_{\{j-1(\vec{i}),\ldots,d-1\}}(\vec{i}) + f_{j-1}(\vec{i}) \\
&\vdots \\
g_{\emptyset}(\vec{i}) &= \alpha_{d-1} g_{\{d-1\}}(\vec{i}) + f_{d-1}(\vec{i})
\end{aligned}
$$

Algorithm 3: Derive subscript expressions

Data: A dimensionality d, a set of expressions
$g_s(\vec{i}), s \in \mathcal{P}([0, d-1])$, a set of values $\alpha_k, k \in [2, d-1]$
Result: A set of expressions $f_k(\vec{i}), k \in [0, d-1]$
$f_0(\vec{i}) = g_{[1,d-1]}(\vec{i})$;
`/* The next line assumes` $\alpha_1 = 0$. `*/`
$f_1(\vec{i}) = g_{2,d-1]}(\vec{i})$;
foreach $j \in [2, d-1]$ **do**
  $f_j(\vec{i}) = g_{[j+1,d-1]}(\vec{i}) - \alpha_j g_{[j,d-1]}(\vec{i})$
return $\{j \to f_j(\vec{i}) : j \in [0, d-1]\}$

from which we derive $f_j(\vec{i}) = g_{[j+1,d-1]}(\vec{i}) - \alpha_j g_{[j,d-1]}(\vec{i})$. Which for a 4D array yields:

$$f_0(\vec{i}) = g_{\{1,2,3\}}(\vec{i})$$
$$f_1(\vec{i}) = g_{\{2,3\}}(\vec{i}) - \alpha_1 g_{\{1,2,3\}}(\vec{i})$$
$$f_2(\vec{i}) = g_{\{3\}}(\vec{i}) - \alpha_2 g_{\{2,3\}}(\vec{i})$$
$$f_3(\vec{i}) = g_{\emptyset}(\vec{i}) - \alpha_3 g_{\{3\}}(\vec{i})$$

The general algorithm (Algorithm 3) is straightforward, as it mainly uses the equalities just given to derive the relevant values.

As a last step, we obtain the set of necessary run-time conditions. This step is unchanged from Section 13.3 - we use again isl to compute the set of parameter values for which we can ensure the absence of out-of-bounds array accesses.

13.5 IMPLEMENTATION

We implemented Section 13.3 within LLVM and Polly [3]. In our implementation, LLVM's scalar evolution framework [99] has been used to perform the transformation of index expressions necessary for example to extract the array size parameter candidates or to perform the division and remainder computations we use to obtain the subscript expressions. Besides the basic delinearization support, we also implemented support for array size parameters in the index expressions (Section 13.3.2) as well as support for deriving a unique array shape for a set of accesses (Section 13.3.1). We also have full support for the generation of run-time conditions that validate our delinearization. For the generation of run-time conditions, we exploited the ability of our new AST generator to generate AST expressions from user-provided integer sets (Section 10.4.5). This feature allowed us to use isl to compute the set of run-time constraints that need to be checked, the AST generator to generate optimal code for them and Polly's code generation back end to translate the resulting AST expressions to LLVM IR. One optimization that has shown useful for

reducing the complexity of run-time conditions is to ask isl to remove any constraints that are only valid for parameter values for which no memory access is executed. This is obviously valid. In case no data access is executed, we can not possibly model this access incorrectly.

The algorithm presented in Section 13.4 has been prototyped and tested in mathematica [72] to ensure we can extend our implementation in LLVM in the future to cover even more important cases.

13.6 EXPERIMENTAL EVALUATION

```
#include <boost/numeric/ublas/matrix.hpp>
using namespace boost::numeric::ublas;

void gemm(matrix<float> &C, matrix<float> &A,
          matrix<float> &B) {
  int n = A.size1();
  int m = B.size2();
  int p = A.size2();

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

Listing 21: A gemm kernel written in C++ using boost::ublas

```
function gemm(A,B,C)
        n,p = size(A)
        m,p = size(B)
        @inbounds for i=1:n, j=1:m, k=1:p
                C[i,j] += A[i,k]*B[k,j]
        end
end
```

Listing 22: A gemm kernel written in Julia

We tested the implementation of our delinearization on all 30 polybench 3.2 kernels [101] with the use of C99 variable length arrays enabled (`-DPOLYBENCH_USE_C99_PROTO`). From the 29 kernels currently detected by Polly (Polly skips floyd-warshall due to zero extends in the loop bounds introduced by the use of 32 bit induction variables), 27 can be correctly delinearized and only two kernels (ludcmp, fdtd-apml) are currently skipped, due to the array size itself being of the form $N + 1$. It is interesting to note that the Polybench code is written in a way that all delinearizations should be statically provable.

Nevertheless our delinearization concluded that run-time checks are necessary for five benchmarks (correlation, covariance, 2mm, doitgen, symm). Looking closer into why run-time checks are still generated we understand that in the original polybench source code certain parameters have been accidentally swapped in the array declarations and loop bounds, which unintentionally changed the semantics of the loop kernels in a way that only if a certain relation between the different parameter holds (e.g. the matrices are quadratic) the execution does not inhibit out-of-bound accesses. The run-time conditions computed directly reflect those conditions and ensure that only in such cases our optimized loop is used. Even though not foreseen, this example nicely shows the benefits of our optimistic delinearization. Not only did it prevent a possible mis-compilation of this code, but it also ensured that we could still optimize it even though there exists a set of parameter values under which this optimization is not correct.

We also verified our delinearization in two other environments. Listing 21 shows a simple gemm kernel implemented in boost::ublas, a blas library that uses C++ expression templates to generate efficient code. After extensive inlining, LLVM can remove the noise of the template instantiations and the matrix multiply kernel is exposed to Polly. After some trivial loop invariant code motion performed manually, our Polly combined with delinearization successfully detects this kernel. In Listing 22 we implemented the same kernel in Julia [32], a dynamic high-level language for scientific computing. Similarly to the ublas example, after some simple loop invariant code motion performed manually, delinearizing the array accesses yields the expected results and the Julia kernel can be optimized with Polly. For matrices of size 1024×1024 and type `float`, already Polly's default optimization speeds up the computation from 15.3 to 2.6 seconds.

13.7 RELATED WORK

There have been previous approaches for delinearization starting with Maslov [91] who introduced delinearization to speed up dependence analysis by reducing the complexity of linear dependence analysis problems that result from multi-dimensional arrays of known size. In his work Maslov also briefly discussed how to handle non-linear array references as they arise from arrays with symbolic sizes. Maslov's work requires the iteration space boundaries to be known, the iteration space itself being rectangular (and starting from zero) and the boundaries to be integer constants. He lifts the last restriction when extending his work to arrays with symbolic sizes, but the iteration space is still required to be rectangular and of a size that allows the delinearization to be proven statically. Furthermore, Maslov requires that: "each resulting dimension must contain at least one variable

and no variable can appear in more than one dimension". Maslov does not explore delinearization outside of the context of dependence analysis and does not address the problem of finding a consistent delinearization for a set of array references. Maslov contributed a second approach in his work on polynomial constraint simplification [93] where delinerization in the context of triangular iteration spaces and possibly non-rectangular (triangular) arrays is discussed. Even though the restrictions on the shape of the iteration space have been lifted, the iteration space is still required to statically prove the delinearization. Also, delinearization is again only applied in the context of dependence analysis. No approach for a consistent delinearization of a group of array references has been shown. His polynomial constraint simplification may suffer from an explosion of the number of constraints, in case of large fixed-size offsets between arrays. Also, the example shown in the paper is exploiting a special case where an induction variable is only used in a single array dimension. It is unclear how general his approach is. Simbürger and Größlinger [116] recently discussed delinearization within Polly using quantifier elimination, but again they focused on solving dependence analysis issues. Cierniak [43] presents a solution independent of dependence analysis, discusses delinearization for non-rectangular arrays and also provides ideas how to unify the delinearization of multiple subscripts. However, he does not discuss a symbolic solution and he requires that each loop index appears in at most one array dimension. In our approach we aim for a purely symbolic solution to delinearize access functions not limited to the context of dependence analysis. Our solution provides a delinearization and the relevant run-time checks even for cases where delinearization can not be proven correct statically. We do not impose constraints on the shape of the iteration space and we allow, except for the presence of array size parameters, arbitrary affine access functions in the delinearized arrays.

13.8 SUMMARY

We have shown an optimistic approach to delinearization that can express polynomial array accesses as multi-dimensional array shapes with sizes given as individual parameters, parameters times a constant or parameters plus a constant, with the first two cases even supporting the use of identical parameters in multiple dimensions. Thanks to our optimistic approach we are able to delinearize array accesses even though the delinearization can not be proven statically. Instead, we provide a set of conditions that can be used to verify our delinearization at run-time. Our approach has been prototyped in mathematica and significant parts have also been implemented in LLVM and Polly. We used this implementation to evaluate our approach on kernels from Julia, blast::ublas and polybench and have

seen promising results. The new support for parametric sized arrays, enables the use of Polly to optimize parametric compute problems expressed in a wide range of programming languages. Or, the other way around, a wide range of programming languages who use LLVM as their compiler is now able to benefit from high-level loop optimizations.

Delinearization as such is highly valuable to enable the exploitation of our tiling techniques in the context of low-level compilers. Parts of the ideas and techniques presented here have been evolved in discussions on the LLVM mailing list with contributions from Hal Finkel, Sebastian Pop and Armin Größlinger. The actual techniques described have been developed in collaboration with P. Sadayappan, J. Ramanujam and Sebastian Pop, with Sebastian Pop also contributing important parts of the delinearization implementation in LLVM.

Part VI

CONCLUSION

14 CONCLUSION

This thesis is concluded with an overview over my personal contributions and an outlook on further work.

14.1 PERSONAL CONTRIBUTIONS

Tilings and Optimizations for Stencils

- The design, modeling and prototyping of a new split tiling algorithm, its integration into a general purpose polyhedral GPU compiler and its evaluation on a set of stencil kernels. This also includes the parametrization of AST generation strategies to ensure sufficient specialization and the separation of full and partial tiles in the generated code.
- The evolution of our split-tiling strategy to a one dimensional hexagonal tiling and its combination with parallelogram tiling to obtain a hybrid tiling that enables us to address GPU specific concerns. Besides the polyhedral formulations of the schedule transformation as well as the implementation of a prototype, this includes the presentations of optimizations that ensure inter-tile reuse and the use of aligned loads to reduce global memory bandwidth.
- Investigation of the relation between hexagonal and diamond tiling. From a detailed analysis of diamond tiling in respect of the constraints it imposes on tile sizes and wavefront coefficients and the formulation of conditions that ensure uniform integer offsets across all tiles, a formulation of hexagonal tiling for two dimensional schedules (1 time dimension, 1 space dimension) applicable in the context of the general purpose optimizer Pluto is derived. An analysis of the tile sizes that maximize the compute-to-communication and compute-to-synchronization ratios is provided for both diamond and hexagonal tiling.

Polyhedral Building Blocks

- Contributions to the design of a polyhedral extractor to ensure it correctly reflects C99 semantics even in parts of the standard

that are not widely understood such as e.g., the wrapping behavior of unsigned integers in the presence of overflow. To evaluate this polyhedral extractor an analysis of the extraction support in existing tools as well as an in-depth comparison with Polly and clan was performed.

- Analysis of existing code generation approaches and definition of new AST generation requirements. Contributions to defining, designing and testing a new AST generation concept that expands AST generation beyond the generation of control flow by allowing the creation of user-defined AST expressions. This in particular includes work to ensure the generation of efficient AST expressions in the presence of existentially quantified variable as well as the design and testing of a fine-grained option system to guide AST generation decisions. An in-depth analysis of polyhedral unrolling was performed as well as an analysis of the capabilities of existing code generation approaches.

- An analysis of schedule uses in existing polyhedral compilers and contributions to the design and formulating of a new tree-based schedule representation. The evaluation of this schedule representation by reformulating our hexagonal tiling strategy in terms of schedule trees.

Low-level Compilers

- A brief overlook over our involvement in the development of gcc/Graphite and LLVM/Polly, a long standing engineering and platform effort to enable the use of our techniques at the level of compiler. IRs.

- Contributions to the requirement analysis and the design of a new approach for the delinearization of arrays with parametric size. This also includes the review and discussion when upstreaming changes to LLVM as well as the implementation of necessary changes in Polly. Finally, the evaluation of this approach on programs using multi-dimensional arrays as provided by C99, Julia and boost::ublast.

14.2 FUTURE WORK

There are several topics that are interesting to address in future work.

Tiling and Optimizations for Stencils

In the context of stencil optimizations we can proceed in different directions. Investigating the benefits of our work on a wider range of

platforms, e.g., different GPU designs or many-core processors such as the Xeon PHI, as well as automatic platform-specific tuning could increase the set of targets that benefit from our work. Orthogonal stencil optimizations such as associative reordering [9] or stencil specific data-layout transformations [68] may enable further optimization opportunities. Similarly, exploiting the large amount of registers available on GPUs to perform more in-register computations or to cache temporary values may be beneficial.

We are also interested in the optimization of non-iterative computations such as sequences of heterogeneous stencils as they arise from image processing pipelines. In such kernels the height of the time dimension is commonly short, such that tiling schemes that can scale their tile size independently of the time tile height seem to be a good choice. For heterogeneous stencils it also seems interesting to not use an over approximation of the dependences to derive tile shapes, but to generate tile shapes that follow precisely the dependences. Such tile shapes are more complicated to describe and code generate, but the integer sets available in isl are generic enough to model them and with sufficient specialization in the AST generator the generated code might be very efficient. One interesting approach to implement this optimization would be to write an LLVM/Polly based optimization pass that uses our work on delinearization to analyze the stencil code Halide [108] generates when given a schedule without any optimizations and to subsequently apply the correct tiling on LLVM-IR. First tests have already shown that parsing Halide code in Polly requires only little changes.

In the context of tile shape analysis it would be interesting to consider a wider range and more complex tile shapes as well as to better incorporate target information. This might allow us to answer questions such as "what is the optimal tile shape for a kernel with two degrees of parallelism on a platform with 256 threads, 128 byte cache line size and 4 data elements per vector?". Making tile shape analysis more efficient is another area worth investigating. This could possibly be done by simplifying the expressions generated by barvinok [132] through the reduction of redundancy, over approximation or some other changes, possibly guaranteeing that the resulting expressions are posynomials [109]. Extending our new AST generator to code generate quasi-polynomials may be another interesting future contribution, as it would enable the run-time evaluation of such posynomials.

Polyhedral Building Blocks

One feature we only drafted but did not test yet in our AST generator is to derive for each (sub)expression the minimal data type necessary to compute it correctly. Even though at the moment we can use large 64 bit data types to avoid integer overflows, being able to

go to smaller data types may be beneficial for both 32 bit architectures and especially targets such as FPGAs. We also hope to further exploit the ability to automatically generate user provided expressions to generate more sophisticated run-time checks. One idea interesting to explore is to allow optimizations that assume run-time behavior that is difficult to model, but well defined, does not occur. For C, this is for example the wrapping of unsigned integers. On LLVM IR such wrapping behavior often needs to be modeled for any type of integer computation, because after some optimizations have been applied LLVM is not always able to preserve enough information to ensure the absence of wrapping. Hence, modeling it correctly is very important to ensure full correctness.

Even though our experience with schedule trees was until now positive, it is essential to get more experience and more community feedback on their usability. We also would like to explore new use cases, e.g., their use for the description of combined manual and automatic transformations. Another interesting use of schedule trees is the possibility to leave the overall schedule intact and only locally optimize it in cases where we are certain that the transformation will be purely positive. Such kinds of uses seem to be helpful in production compilers where it is often more important to ensure that performance is never degraded rather than to increase the average performance of all optimized code.

Low-level Compilers

After having increased the kinds of codes we can handle to multi-dimensional arrays of parametric size, we would like to further widen the scope of where Polly can be used. This time we would like to look into dynamic language features as they appear in C++ or in languages such as Julia [31]. Those languages pose interesting new research problems such as for example the elimination of run-time bound checks. In dynamic languages compute kernels often raise out-of-bounds exceptions in their core compute loops. Being able to create run-time checks that guard a version of the code where we can be sure no run-time checks are needed will allow us to perform aggressive optimizations while at the same time preserving the safety guarantees given by the language. As a result there may be no need for the user to decide between a safe version of their code or a fast version of their code. The ability of our AST generator to generate run-time checks from possibly complex conditions should be very helpful for this work.

Part VII

APPENDIX

[12] Walid Abu-Sufah, David Kuck, and Duncan Lawrie. Automatic program transformations for virtual memory computers. In *Proceedings of the 1979 National Computer Conference*, pages 969–969. IEEE Computer Society, 1979.

[13] Christophe Alias, Fabrice Baray, and Alain Darte. Bee+ cl@ k: An implementation of lattice-based array contraction in the source-to-source translator rose. In *ACM SIGPLAN Notices*, volume 42, pages 73–82. ACM, 2007.

[14] Frances Allen and John Cocke. A catalogue of optimizing transformations. 1971.

[15] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(4):491–542, 1987.

[16] Saman P Amarasinghe, Jennifer-Ann M Anderson, Monica S Lam, and Chau-Wen Tseng. An overview of the suif compiler for scalable parallel machines. In *PPSC*, pages 662–667, 1995.

[17] Mehdi Amini, Corinne Ancourt, Fabien Coelho, Béatrice Creusillet, Serge Guelton, FranÃ§ois Irigoin, Pierre Jouvelot, Ronan Keryell, and Pierre Villalon. PIPS is not (just) polyhedral software. In C. Alias and C. Bastoul, editors, *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Chamonix, France, 2011.

[18] Mehdi Amini, Béatrice Creusillet, Stéphanie Even, Ronan Keryell, Onig Goubier, Serge Guelton, Janice Onanian McMahon, François-Xavier Pasquier, Grégoire Péan, Pierre Villalon, et al. Par4All: From convex array regions to heterogeneous computing. In *IMPACT*, 2012.

[19] Corinne Ancourt and François Irigoin. Scanning polyhedra with do loops. In *ACM Sigplan Notices*, volume 26, pages 39–50. ACM, 1991.

[20] 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. *Science of Computer Programming*, 72(1):3–21, 2008.

[21] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Supercomputing*, page 40. IEEE Computer Society Press, 2012.

[22] Utpal Banerjee. *Data dependence in ordinary programs*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1976.

[23] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors. *SIGPLAN Notices*, 44(4):219–228, February 2009. ISSN 0362-1340. doi: 10.1145/1594835.1504209.

[24] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. Automatic c-to-cuda code generation for affine programs. In *Compiler Construction*, pages 244–263. Springer, 2010.

[25] Cédric Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC*, volume 2, pages 23–30, 2003.

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

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

[28] Cedric Bastoul. Clan - a polyhedral representation extractor for high level programs, 2008.

[29] Marouane Belaoucha, Denis Barthou, Adrien Eliche, and Sid-Ahmed-Ali Touati. Fadalib: an open source c++ library for fuzzy array dataflow analysis. *Procedia Computer Science*, 1(1): 2075–2084, 2010.

[30] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *Compiler Construction*, pages 283–303. Springer, 2010.

[31] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *arXiv preprint arXiv:1209.5145*, 2012.

[32] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[33] Somashekaracharya G Bhaskaracharya and Uday Bondhugula. Polyglot: a polyhedral loop transformation framework for a graphical dataflow language. In *Compiler Construction*, pages 123–143. Springer, 2013.

[34] William Blume, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, et al. Parallel programming with polaris. *Computer*, 29(12):78–82, 1996.

[35] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Notices*, 43(6):101–113, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375595.

[36] Uday Bondhugula, Oktay Günlük, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 343–352, 2010.

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

[38] Francky Catthoor, Eddy de Greef, and Sven Suytack. *Custom memory management methodology: Exploration of memory organisation for embedded multimedia system design*. Kluwer Academic Publishers, 1998.

[39] Chun Chen. Polyhedra scanning revisited. In *Conference on Programming Language Design and Implementation*, pages 499–508, New York, NY, USA, 2012. ACM.

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

[41] Matthias Christen, Olaf Schenk, and Helmar Burkhart. Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In *IPDPS*, 2011.

[42] Matthias-Michael Christen. *Generating and auto-tuning parallel stencil codes*. PhD thesis, University of Basel, 2011.

[43] Michal Cierniak and Wei Li. Recovering logical data and code structures. Technical report, 1995.

[44] 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*, pages 151–160. ACM, 2005.

[45] NVIDIA Corporation. Tegra k1 technical white paper, v1.0, 2013.

[46] Béatrice Creusillet and Francois Irigoin. Interprocedural array region analyses. *International Journal of Parallel Programming*, 24, December 1996. ISSN 0885-7458.

[47] Alain Darte, Yves Robert, and Frédéric Vivien. Loop parallelization algorithms. In *Compiler Optimizations for Scalable Parallel Systems: Languages, Compilation Techniques and Run Time Systems*, volume 1808 of *LNCS*, pages 141–171. Springer Verlag, 2001.

[48] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of SC '08*, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press. ISBN 978-1-4244-2835-9.

[49] Kaushik Datta, Shoaib Kamil, Samuel Williams, Leonid Oliker, John Shalf, and Katherine A. Yelick. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM Review*, 51(1):129–159, 2009.

[50] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.

[51] Peng Di and Jingling Xue. Model-driven tile size selection for DOACROSS loops on GPUs. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 401–412, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23396-8.

[52] Jack J Dongarra, Hans W Meuer, and Erich Strohmaier. Top500 supercomputer sites. 1994.

[53] ACE Associated Compiler Experts. Parallelization using polyhedral analysis, 2008.

[54] Paul Feautrier. Array expansion. In *2nd International Conference on Supercomputing (ICS'88)*, pages 429–441. ACM, 1988.

[55] Paul Feautrier. Parametric integer programming. *RAIRO Recherche opérationnelle*, 22(3):243–268, 1988.

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

[57] Paul Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. *International Journal of Parallel Programming*, 21:389–420, 1992. ISSN 0885-7458. 10.1007/BF01379404.

[58] Paul Feautrier. Some efficient solutions to the affine scheduling problem: Part i. one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992. ISSN 0885-7458. doi: 10.1007/BF01407835.

[59] Max Geigl. Parallelization of loop nests with general bounds in the polyhedron model. *Master's thesis, Universit at Passau*, 1997.

[60] 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. ISSN 0885-7458. doi: 10.1007/s10766-006-0012-3.

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

[62] Martin Griebl and Christian Lengauer. The loop parallelizer loopo. In *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21, pages 311–320. Citeseer, 1996.

[63] Martin Griebl, Paul Feautrier, and Christian Lengauer. Index set splitting. *International Journal of Parallel Programming*, 28(6): 607–631, 2000.

[64] OpenACC Working Group et al. The openacc application programming interface, 2011.

[65] Gaël Guennebaud. Eigen: a c++ linear algebra library. 2011.

[66] Dongni Han, Shixiong Xu, Li Chen, and Lei Huang. Pads: A pattern-driven stencil compiler-based tool for reuse of optimizations on gpgpus. In *ICPADS*, pages 308–315, 2011.

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

[68] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, J. Ramanujam, and P. Sadayappan. A stencil compiler for short-vector SIMD architectures. In *International Conference on Supercomputing (ICS)*. ACM, 2013.

[69] Ryutaro Himeno. Himeno benchmark, 2011.

[70] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *International Conference on Supercomputing (ICS)*, 2012.

[71] Song-You Hong, Jimy Dudhia, and Shu-Hua Chen. A revised approach to ice microphysical processes for the bulk parameterization of clouds and precipitation. *Monthly Weather Review*, 132(1), 2004.

[72] Wolfram Research Inc. Mathematica 9.0, 2012.

[73] Guillaume Iooss, Sanjay Rajopadhye, Christophe Alias, and Yun Zou. Cart: Constant aspect ratio tiling. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, January 2014.

[74] François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *Proceedings of the 5th international conference on Supercomputing*, pages 244–251. ACM, 1991.

[75] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[76] Marta Jiménez, José M Llabería, and Agustín Fernández. Register tiling in nonrectangular iteration spaces. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(4):409–453, 2002.

[77] W Kelly, V Maslov, W Pugh, E Rosser, T Shpeisman, and D Wonnacott. New user interface for petit and other interfaces: user guide. *University of Maryland*, 1995.

[78] Wayne Kelly. *Optimization within a Unified Transformation Framework*. PhD thesis, 1996.

[79] Wayne Kelly and William Pugh. A unifying framework for iteration reordering transformations. In *IEEE First Int. Conf. on Algorithms and Architectures for Parallel Processing (ICAPP 95)*, volume 1, April 1995.

[80] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, 1995.

[81] Wayne Kelly, Vadim Maslov, William Pugh, Evan Rosser, Tatiana Shpeisman, and Dave Wonnacott. The Omega calculator and library, version 1.1.0. 1996.

[82] 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, 2007. ISBN 978-1-59593-764-3. doi: 10.1145/1362622.1362691.

[83] 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*, pages 127–138. ACM, 2013.

[84] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective automatic parallelization of stencil computations. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 235–244, 2007.

[85] David J Kuck, Robert H Kuhn, David A Padua, Bruce Leasure, and Michael Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–218. ACM, 1981.

[86] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[87] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated computers from a portable high level programming abstraction. In *3rd Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU)*, GPGPU '10, New York, NY, USA, 2010. ACM.

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

[89] Vincent Loechner and Doran K Wilde. Parameterized polyhedra and their vertices. *International Journal of Parallel Programming*, 25(6):525–549, 1997.

[90] W. A. Maniatty, B.K. Szymanski, and T. Caraco. Parallel computing with generalized cellular automata. Technical report, Department of Computer Science, Rensselaer Polytenchic Institute, 1998.

[91] Vadim Maslov. Delinearization: An efficient way to break multiloop dependence equations. *SIGPLAN Not.*, 27(7):152–161, July 1992. ISSN 0362-1340. doi: 10.1145/143103.143130.

[92] Vadim Maslov. Lazy array data-flow dependence analysis. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *POPL*. ACM Press, 1994. ISBN 0-89791-636-0.

[93] Vadim Maslov and William Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. Technical report, In CONPAR 94 - VAPP VI, Int. Conf. on Parallel and Vector Processing, 1994.

[94] Jiayuan Meng and Kevin Skadron. A performance study for iterative stencil loops on gpus with ghost zone optimizations. *International Journal of Parallel Programming*, 39(1):115–142, 2011.

[95] Paulius Micikevicius. 3d finite difference computation on gpus using cuda. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, pages 79–84, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-517-8. doi: 10.1145/1513895.1513905.

[96] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-d blocking optimization for stencil computations on modern cpus and gpus. In *Proceedings of SC '10*, pages 1–13, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-7559-9. doi: 10.1109/SC.2010.2.

[97] Daniel Orozco, Elkin Garcia, and Guang Gao. Locality optimization of stencil applications using data dependency graphs. In Keith Cooper, John Mellor-Crummey, and Vivek Sarkar, editors, *Languages and Compilers for Parallel Computing*, volume 6548 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-19594-5. doi: 10.1007/978-3-642-19595-2_6.

[98] Constantine D Polychronopoulos, Milind B Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1):45–72, 1989.

[99] Sebastian Pop, Albert Cohen, and Georges-André Silber. Induction variable analysis with delayed abstractions. In *High Performance Embedded Architectures and Compilers*, pages 218–232. Springer, 2005.

[100] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006, 2006.

[101] L.-N. Pouchet. PolyBench/C 3.2. `http://www.cs.ucla.edu/~pouchet/software/polybench/`.

[102] Louis-Noël Pouchet. *Interative Optimization in the Polyhedral Model*. PhD thesis, University of Paris-Sud 11, Orsay, France, January 2010.

[103] Louis-Noël Pouchet. Polyopt, a polyhedral optimizer for the rose compiler, 2011.

[104] William Pugh. Uniform techniques for loop optimization. In *5th International Conference on Supercomputing (ICS'91)*, pages 341–352. ACM, 1991.

[105] William Pugh and David Wonnacott. *An exact method for analysis of value-based array data dependences*. Springer, 1994.

[106] William Pugh and David Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4): 1248–1278, 1994.

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

[108] 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 *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seattle, WA, June 2013.

[109] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. Positivity, posynomials and tile size selection. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pages 1–12. IEEE, 2008.

[110] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *ACM SIGPLAN Notices*, volume 42, pages 405–414. ACM, 2007.

[111] Jon Peddie Research. Qualcomm single largest proprietary gpu supplier, imagination technologies the leader in gpu ip, arm and vivante growing rapidly, according to latest report from jon peddie research.

[112] Jason Sams. Blog post: Renderscript part 2, 2011.

[113] Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical report, 1990.

[114] Eric Schweitz, Richard Lethin, Allen Leung, and Benoit Meister. R-stream: A parametric high level compiler. *Proceedings of HPEC*, 2006.

[115] Jun Shirako and Vivek Sarkar. Oil and water can mix! experiences with integrating polyhedral and ast-based transformations. 2013.

[116] Andreas Simbürger and Armin Größliger. On the variety of static control parts in real-world programs: from affine via multi-dimensional to polynomial and just-in-time. In Sanjay Rajopadhye and Sven Verdoolaege, editors, *Proceedings of the 4th International Workshop on Polyhedral Compilation Techniques*, Vienna, Austria, January 2014.

[117] G. Smith. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 2004.

[118] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache oblivious parallelograms in iterative stencil computations. In *ICS*, pages 49–59, 2010.

[119] Robert Strzodka, Mohammed Shaheen, Dawid Pajak, and Hans-Peter Seidel. Cache accurate time skewing in iterative stencil computations. *2012 41st International Conference on Parallel Processing*, 0:571–581, 2011. ISSN 0190-3918. doi: http://doi.ieeecomputersociety.org/10.1109/ICPP.2011.47.

[120] A. Taflove. *Computational electrodynamics: The Finite-difference time-domain method*. Artech House, 1995.

[121] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The Pochoir stencil compiler. In *SPAA*, pages 117–128. ACM, 2011.

[122] Konrad Trifunović, 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 eal-world polyhedral compilation. In *2nd GCC Research Opportunities Workshop (GROW)*, 2010.

[123] Robert A Van Engelen. Efficient symbolic analysis for optimizing compilers. In *Compiler Construction*, pages 118–132. Springer, 2001.

[124] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. Polyhedral code generation in the real world. In *International Conference on Compiler Construction (CC)*, volume 3923 of *LNCS*, pages 185–201, Vienna, 2006. Springer.

[125] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT*, Paris, France, January 2012.

[126] Nicolas T. Vasilache. *Scalable Program Optimization Techniques in the Polyhedral Model*. PhD thesis, Université Paris Sud XI, Orsay, September 2007.

[127] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Strout. Non-affine Extensions to Polyhedral Code Generation. In *International Symposium on Code Generation and Optimization (CGO)*, Orlando, FL, United States, 2014.

[128] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software (ICMS'10)*, LNCS 6327, pages 299–302. Springer-Verlag, 2010.

[129] Sven Verdoolaege. Counting affine calculator and applications. In C. Alias and C. Bastoul, editors, *1st International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Chamonix, France, 2011.

[130] Sven Verdoolaege. Integer sets and relations: from high-level modeling to low-level implementation, 2013. Spring School on Polyhedral Code Analysis and Optimizations.

[131] Sven Verdoolaege. Integer set library: Manual - version 0.12.1, 2014.

[132] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner, and Maurice Bruynooghe. Counting integer points in parametric polytopes using Barvinok's rational functions. *Algorithmica*, 48(1):37–66, June 2007.

[133] Sven Verdoolaege, Gerda Janssens, and Maurice Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. In *Computer Aided Verification 21*, pages 599–613. Springer, June 2009.

[134] Sven Verdoolaege, Martin Palkovič, Maurice Bruynooghe, Gerda Janssens, and Francky Catthoor. Experience with widening based equivalence checking in realistic multimedia systems. *Journal of Electronic Testing*, 26(2):279–292, 2010.

[135] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4):54:1–54:23, January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400713.

[136] Robert P Wilson, Robert S French, Christopher S Wilson, Saman P Amarasinghe, Jennifer M Anderson, Steve WK Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W Hall, Monica S Lam, et al. Suif: An infrastructure for research on parallelizing and optimizing compilers. *ACM Sigplan Notices*, 29(12):31–37, 1994.

[137] Michael E Wolf and Monica S Lam. A data locality optimizing algorithm. *ACM Sigplan Notices*, 26(6):30–44, 1991.

[138] Michael E Wolf and Monica S Lam. A loop transformation theory and an algorithm to maximize parallelism. *Parallel and Distributed Systems, IEEE Transactions on*, 2(4):452–471, 1991.

[139] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361. Society for Industrial and Applied Mathematics, 1987.

[140] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. AlphaZ: A system for design space exploration in the polyhedral model. In *Proceedings of the 25th International Workshop on Languages and Compilers for Parallel Computing*, 2012.

[141] Xing Zhou, Jean-Pierre Giacalone, María Jesús Garzarán, Robert H. Kuhn, Yang Ni, and David Padua. Hierarchical overlapped tiling. In *Proceedings of the 10th Intl. Symp. Code Gen. and Opt.*, CGO '12, pages 207–218, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1206-6. doi: 10.1145/2259016.2259044.

[142] Wei Zuo, Peng Li, Deming Chen, Louis-Noël Pouchet, Shunan Zhong, and Jason Cong. Improving polyhedral code generation for high-level synthesis. In *International Conference on Hardware/Software Codesign and System Synthesis*, 2013.