Polyhedral Compilation for Domain Specific Languages

Chandan Reddy

PSL
UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à **École Normale Supérieure**

Polyhedral Compilation for Domain Specific Languages

Soutenue par
Chanadan Reddy
Le 31 Mars 2019

Ecole doctorale n° 386
SCIENCS MATHÉMATIQUES DE PARIS CENTRE

Spécialité
INFORMATIQUE

Composition du jury :

M. Cédric Bastoul Professor, University of Strasbourg	*Rapporteur*
M. Saman Amarasinghe Professor, Massachusetts Institute of Technology	*Rapporteur*
Mme Corinne Ancourt Directeur de recherche, MINES ParisTech	*Examinateur*
Mme Christine Eisenbeis Directeur de recherche, INRIA Saclay	*Examinateur*
M. Grigori Fursin Ingénieur de recherche, Dividiti	*Examinateur*
M. Albert Cohen Directeur de recherche, Google and ENS	*Directeur de thèse*

ENS | PSL

Abstract

In the recent years, the complexity of optimizing compilers has increased significantly due to increasing diversity of programming languages and heterogeneity of target architectures. Even though there has been a lot of progress with the general purpose compilers, they are not been able to extract peak level performance provided by the specialized libraries. To bridge this performance gap domain specific compilers(DSLs) are proposed, by restricting input to a specialized domain it can perform more aggressive transformations needed to achieve peak performance while being more flexible than standard libraries. One of the major optimization needed to obtain high performance on modern heterogeneous architectures is loop transformations to exploiting locality and automatic parallelization. The polyhedral model has evolved as a highly efficient, reusable generic framework for loop optimizations especially for regular static control affine programs. In this thesis we explore the suitability of polyhedral loop transformation framework in context of compiling Image processing and Deep learning pipelines. We study the challenges of adapting a generic polyhedral scheduler for DSLs. We propose various extensions to the scheduler to find optimal schedule by modeling various hardware and application characteristics.

We present method to handle reductions in polyhedral model. In the state-of-the-art polyhedral compilers there was no support for reductions. The reduction loop was treated as a serial loop and this may be a major bottleneck for several applications especially on GPUs. We propose languages extensions in PENCIL to express arbitrary user-defined reductions. We encode this reduction information in polyhedral model using reduction dependences. We show how to use this dependences in polyhedral scheduler to exploit parallelization of reduction loops. We also propose a template based code generation for emitting highly efficient reduction code for GPUs. We validate our approach through experiments by comparing automatically generated code with the highly tuned library.

Exploiting locality is a key factor in achieving high performance on the complex processors with complex memory/computation hierarchies. The cost function used in the Pluto algorithm optimizes only temporal locality. Exploiting spatial locality is as important as temporal locality and it has implications on vectorization and coalesced memory accesses. we propose a new unified algorithm for optimizing parallelism and locality in loop nests, that is capable of modeling temporal and spatial effects of multiprocessors and accelerators with deep memory hierarchies and multiple levels of parallelism. It orchestrates a collection of parametrizable optimization problems for locality and parallelism objectives over a polyhedral space of semantics-preserving transformations. We discuss the rationale for this unified algorithm, and validate it on a collection of representative computational kernels/benchmarks.

We study the challenges of using polyhedral compilation techniques for a complex, real-world, end-to-end image processing application called SLAMBench. The SLAMBench has several non-affine kernels that not readily amendable for polyhedral compilation.We show the usefulness of summary functions to compile such non-affine parts of the program thus extending the reach of polyhedral compilation. We also present prl runtime library needed to avoid redundant data transfers between device and host. We validate our high-level compilation approach through experiments comparing the performance of the generated code with the highly optimized manual version of the SLAMBench.

We also study the applicability of polyhedral compilation for optimizing deep learning pipelines. Most of the operations in the deep learning pipelines are affine hence are suitability for polyhedral compilation. Our framework is build on TVM an end-to-end deep learning compilation framework with support for multiple front ends such as MXNet, Tensorflow etc. and supports multiple different architectures. We extract the polyhedral representation from TVM IR and use polyhedral scheduler along with performance model based autotuning to automatically find the schedules for TVM operators. In this context we extend the polyhedral scheduler to find optimal schedules for different sizes and shapes of the tensor. We model the amount of data reuse for the case when all the parameter values are known, and formulate the constraints to ILP to maximize data reuse. We also present a performance model based autotuning technique that can cut down the tuning time from hours to minutes. We conduct experiments on the common deep learning benchmarks validating the effectiveness and general applicability of our technique in providing portable performance.

Finally, we summarize our work and present concluding remarks as well as future research directions. We believe with the improvements proposed in this dissertation improves the effectiveness of polyhedral framework as a loop transformation framework for compiling DSLs.

Résumé

Au cours des dernières années, la complexité de l'optimisation du compilateur a considérablement augmenté en raison de la diversité croissante des langages de programmation et de l'hétérogénéité des cibles architectures. Même si les compilateurs à usage général ont beaucoup progressé, ils ne sont pas en mesure d'extraire les performances de pointe fournies par les bibliothèques spécialisées. Pour remédier à cette situation, des compilateurs spécifiques au domaine (DSL) sont proposés, en limitant la saisie à un domaine spécialisé, il peut effectuer des transformations plus agressives nécessaires pour atteindre le pic performances tout en étant plus flexible que les bibliothèques standard. Une des optimisations majeures des transformations de boucle sont nécessaires pour obtenir des performances élevées sur les architectures hétérogènes modernes à exploiter la localité et la parallélisation automatique. Le modèle polyédrique a évolué comme un cadre générique hautement efficace et réutilisable pour l'optimisation des boucles, en particulier pour programmes affines de contrôle statique réguliers. Dans cette thèse, nous explorons la pertinence de cadre de transformation de boucle polyédrique dans le contexte de la compilation Traitement de l'image et Pipelines d'apprentissage en profondeur. Nous étudions les défis de l'adaptation d'un générique ordonnanceur polyédrique pour DSL. Nous proposons diverses extensions à le planificateur pour trouver la planification optimale en modélisant divers matériels et caractéristiques d'application.

Nous présentons une méthode pour gérer les réductions dans un modèle polyédrique. Dans l'état de l'art compilateurs polyédriques, il n'ya pas eu de soutien aux réductions. La réduction boucle a été traitée comme une boucle série et cela peut être un goulot d'étranglement majeur pour plusieurs applications notamment sur les GPU. Nous proposons des extensions de langues dans PENCIL pour exprimer des réductions arbitraires définies par l'utilisateur. Nous encodons ceci informations de réduction dans un modèle polyédrique utilisant des dépendances de réduction. Nous montrons comment utiliser ces dépendances dans un planificateur polyédrique exploiter la parallélisation des boucles de réduction. Nous proposons également un génération de code basée sur des modèles pour une réduction très efficace de l'émission code pour les GPU. Nous validons notre approche par des expériences de comparer le code généré automatiquement avec le très optimisé bibliothèque.

L'exploitation de la localité est un facteur clé pour atteindre de hautes performances sur le processeurs complexes avec des hiérarchies complexes de mémoire / calcul. Le coût fonction utilisée dans l'algorithme de Pluton n'optimise que la localité temporelle. L'exploitation de la localité spatiale est aussi importante que la localité temporelle et a des implications sur la vectorisation et les accès mémoire coalescés. nous proposons un nouvel algorithme unifié pour optimiser le

parallélisme et localité dans des nids de boucles, capable de modéliser temporellement et effets spatiaux des multiprocesseurs et des accélérateurs à mémoire profonde les hiérarchies et les multiples niveaux de parallélisme. Il orchestre un collection de problèmes d'optimisation paramétrables pour la localité et objectifs de parallélisme sur un espace polyédrique de préservation de la sémantique transformations. Nous discutons de la raison de cet algorithme unifié, et valider sur une collection de calcul représentatif noyaux / points de repère.
Nous étudions les défis de l'utilisation de techniques de compilation polyédriques pour SLAMBench, une application de traitement d'image complexe et réaliste, de bout en bout. La SLAMBench a plusieurs noyaux non affines qui ne peuvent pas être facilement modifiés compilation polyédrique.Nous montrons l'utilité des fonctions de synthèse pour compiler de telles parties non affines du programme étendant ainsi la portée de la compilation polyédrique. Nous présentons également la bibliothèque d'exécution prl nécessaire pour éviter les redondances. transferts de données entre l'appareil et l'hôte. Nous validons notre haut niveau approche de compilation par des expériences comparant les performances du code généré avec la version manuelle hautement optimisée de SLAMBench.
Nous étudions également l'applicabilité de la compilation polyédrique à l'optimisation pipelines d'apprentissage en profondeur. La plupart des opérations dans l'apprentissage en profondeur les pipelines sont affines et conviennent donc à la compilation polyédrique. Notre cadre repose sur TVM, une compilation d'apprentissage en profondeur de bout en bout framework avec support de plusieurs interfaces telles que MXNet, Tensorflow etc. et prend en charge plusieurs architectures différentes. Nous extrayons la représentation polyédrique de TVM IR et utilisons planificateur polyédrique avec autotuning basé sur un modèle de performance pour trouver automatiquement les horaires des opérateurs TVM. Dans ce contexte, nous étendons l'ordonnanceur polyédrique pour trouver horaires optimaux pour différentes tailles et formes du tenseur. Nous modélisons la quantité de données réutilisées pour le cas où tous les les valeurs des paramètres sont connues et formulent les contraintes à ILP maximiser la réutilisation des données. Nous présentons également un modèle de performance basé sur technique de réglage automatique qui peut réduire le temps de réglage de quelques heures à quelques minutes. Nous menons des expériences sur les critères communs d'apprentissage en profondeur qui valident la l'efficacité et l'applicabilité générale de notre technique dans fournir des performances portables.
Enfin, nous résumons nos travaux et présentons les conclusions finales ainsi que les recherches futures. directions. Nous croyons aux améliorations proposées dans cette thèse améliore l'efficacité du cadre polyédrique en tant que transformation de boucle cadre pour la compilation de DSL.

Acknowledgements

Undertaking this Ph.D has been a truly challenging experience for me and it would not have been possible to do it without the support and guidance from many people.

Firstly, I would like to express my sincere gratitude to my advisor Albert Cohen for the continuos support my research, for his patience, motivation and immense knowledge. I highly appreciate the freedom he gave me in choosing my topics of research. I am grateful for his patience and encouragement during the difficult phase of my thesis. I would also like to thank him for funding all travel for conferences and workshops. Without his guidance and constant feedback this thesis would not have been achievable.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof Cédric Bastoul, Prof Saman Amarasinghe, Mme Corinne Ancourt, Mme Christine Eisenbeis and M. Grigori Fursin. Special thanks to Prof Cédric Bastoul and Prof Saman Amarasinghe for taking time from their busy schedule to review my thesis.

I thanks my fellow team mates of Parkas lab for all the stimulating discussions and all the fun we had we had in the last four years. Thank you Tobias, Riyad, Ulysee, Alex, Jie, Guillaume, Lelio, Andy, Basile, Tim and all others. It was great sharing lab with all of you during last four years.

I am grateful to my mum, dad and sister Geetha, Gopal and Chaithra, for their moral and emotional support throughout my life. A very special gratitude to Linda for all support and encouragement during my Ph.D, very well appreciated, Thank you.

Chandan

Contents

List of Figures

List of Tables

1 Introduction

Computer architectures continue to grow in complexity, stacking levels of parallelism and deepening their memory hierarchies to mitigate physical bandwidth and latency limitations. Harnessing more than a small fraction of the performance offered by such systems is a task of ever growing difficulty. Optimizing compilers transform a high-level, portable, easy-to-read program into more complex but efficient, target-specific implementation. Achieving performance portability is even more challenging: multiple architectural effects come into play that are not accurately modeled as convex optimization problems, and some may require mutually conflicting program transformations. In this context, systematic exploration of the space of semantics-preserving, parallelizing and optimizing transformations remains a primary challenge in compiler construction.

Loop nest optimization holds a particular place in optimizing compilers as, for computational programs such as those for scientific simulation, image processing, or machine learning, a large part of the execution time is spent inside nested loops. Research on loop nest transformations has a long history [Wol95, KA02]. Much of the past work focused on specific transformations, such as fusion [KM93], interchange [AK84] or tiling [Wol89, IT88], or specific objectives, such as parallelization [Wol86] or vectorization [AK87].

The *polyhedral framework* of compilation introduced a rigorous formalism to represent and operate on the control flow, data flow, and storage mapping of a growing class of loop-based programs [FL11]. It provides a unified approach to loop nest optimization, offering precise relational analyses, formal correctness guarantees and the ability to perform complex sequences of loop transformations in a single optimization step by using powerful code generation/synthesis algorithms. It has been a major force driving research on loop transformations in the past decade thanks to the availability of more generally applicable algorithms, robust and scalable implementations, and embedding into general or domain-specific compiler frameworks [PCB+06, GGL12a]. Loop transformations in polyhedral frameworks are generally abstracted by means of a *schedule*, a multidimensional relation from iterative instances of program statements to logical time. Computing the most profitable legal schedule is the primary goal of a polyhedral optimizer.

Feautrier's algorithm computes minimum delay schedules [Fea92b] for arbitrary nested loops with affine bounds and array subscripts. The Pluto algorithm revisits the method to expose coarse-grain parallelism while improving temporal data locality [BHRS08a, BAC16]. However, modern complex processor architectures have made it imperative to model more diverse sources of performance; deep memory hierarchies that favor consecutive accesses—cache lines on CPUs, memory coalescing on GPUs—are examples of hardware capabilities which must be exploited to match the performance of hand-optimized loop transformations.

1.1 Towards Domain-specific Languages

There exist a large number of mature polyhedral compilation frameworks and loop optimizers, including both research projects [BHRS08b, CCH08, VCJC+13a], and commercial productions [TCE+10, GGL12b, SLLM06]. Such compilers usually take a general-purpose high-level or intermediate language as input, extract the polyhedral representation for static affine parts of it, perform loop optimizations in polyhedral IR and then transform optimizes polyhedral representation to input high-level or intermediate language. Unlike the traditional compiler where there are multiple compilation stages for each individual loop transformations, in polyhedral model all the loop transformations are combined into a single compilation stage. This greatly simplifies the loop transformation stages as we do not have to worry about ordering of individual loop transformation stages. Despite the success over traditional compilers, the optimality of the code generated by such polyhedral compilers still remains elusive, falling behind the performance of heavily hand-tuned codes written by an expert.

One of the reason of performance gap between the generated codes of optimizing compilers and hand-written programs is due to the conservativeness of general combination stages. An expert programmer makes use of domain specific and problem specific information to produce highly optimized library functions for a given architecture whose performance is close to machine peaks. The general purpose compiler does not have all the relevant domain specific information and it cannot perform aggressive transformations needed achieve peak performance. It is very difficult to prove the correctness of such aggressive transformations in a general context hence a very conservative heuristics are chosen. It is not possible for the programmer to convey this domain specific knowledge to the compiler in general purpose languages.

Domain-specific languages (DSLs) are proposed to bridge this performance gap and are becoming prevalent in many application domains. A high-level language that is specific to a given domain improves the programmer productivity since there are high level abstractions specific to a given domain can be used as building blocks. It also allows the DSL compiler to take advantage of the domain specific knowledge while compiling such high level abstractions. It can perform aggressive transformations since input domain is restricted. This makes it much easier to reason about correctness of such transformations and can use aggressive heuristics to obtain high performance. To summaries general purpose compilation is a evolutionary while domain specific compilation is revolutionary. The polyhedral model was successfully integrated with DSLs, such

as those for optimizing DSLs for graphical dataflow language [BB13, SSPS15], stencil computations [HVF+13], image processing applications Halide [RKBA+13] PolyMage [MVB15] etc. Recently, due to the revolution in the field of deep learning there are several DSL compilation frameworks such as XLA in Tensorflow [AIM17], JIT compiler in Pytorch [PCC+], TensorComprehensions [VZT+18] and TVM compiler stack [tvm] are gaining popularity. The loop transformations play very a important role in achieving high performance in these domains. The polyhedral model is highly suited as intermediate representation for performing loop transformations in these DSL compilation framework.

1.2 Handling spatial locality

There has been some past work on incorporating knowledge about consecutive accesses into a polyhedral optimizer, mostly as a part of transforming programs for efficient vectorization [TNC+09, VMBL12, KVS+13]. However, these techniques restrict the space of schedules that can be produced; we show that these restrictions miss potentially profitable opportunities involving schedules with linearly dependent dimensions or decoupling the locality optimization of individual fused clusters of statements. In addition, these techniques model non-convex optimization problems through the introduction of additional discrete (integer, boolean) variables and introducing bounds on coefficients. These ad-hoc bounds do not practically impact the quality of the results, but remain slightly unsatisfying from a mathematical modeling perspective. Finer architectural modeling such as the introduction of spatial effects also pushes for more discrete variables, requiring extra algorithmic effort to keep the dimensional growth under control. A different class of approaches relies on a combination of polyhedral and traditional, syntactic-level loop transformations. A polyhedral optimizer is set up for one objective, while a subsequent syntactic loop transformation addresses another objective. For example, PolyAST uses a polyhedral optimizer to improve locality through affine scheduling and loop tiling. After that, it applies syntactic transformations to expose different forms of parallelism [SPS14]. Prior to PolyAST, the pioneering Pluto compiler itself relied on a heuristic loop permutation to improve spatial locality after the main polyhedral optimization aiming for parallelism and temporal locality [BHRS08a]. Operating in isolation, the two optimization steps may end up undoing each other's work, hitting a classical compiler phase ordering problem.

We propose a polyhedral scheduling algorithm that accounts for multiple levels of parallelism and deep memory hierarchies, and does so without imposing unnecessary limits on the space of possible transformations. Ten years ago, the Pluto algorithm made a significant contribution to the theory and practice of affine scheduling for locality and parallelism. Our work extends this frontier by revisiting the models and objectives in light of concrete architectural and microarchitectural features, leveraging positive memory effects (e.g., locality) and avoiding the negative ones (e.g., false sharing). We fundamentally extend the reach of affine scheduling, seeing it as a collection of parameterizable optimization problems, with configurable constraints and objectives, allowing for schedules with linearly dependent dimensions, and dealing with non-convex spaces. In particular, we contribute a "clustering" technique for loop fusion (an essential locality-enhancing

transformation) which allows to precisely intertwine the iterations of different statements while maintaining the execution order within each loop, and we extend the loop sinking options when aligning imperfectly nested loops to the same depth. We address spatial effects by extending the optimization objective and allowing for linearly dependent dimensions in affine schedules that are out of reach of a typical polyhedral optimizer. We also provide an original approach to non-convex optimization problems where negative schedule coefficients are necessary to tile loop iteration spaces while aligning them with the direction of consecutive memory accesses.

We design our algorithm as a template with multiple configuration dimensions. Its flexibility stems from a parameterizable scheduling problem and a pair of optimization objectives that can be interchanged during the scheduling process. As a result, our approach is able to produce in one polyhedral optimization pass schedules that previously required a combination of polyhedral and syntactic transformations. Since it remains within the polyhedral model, it can benefit from its transformation expressiveness and analysis power, e.g., to apply optimizations that a purely syntactic approach might not consider, or to automatically generate parallelized code for different accelerators from a single source.

1.3 Reductions in polyhedral model

A reduction is an associative and commutative computation that operates on a set of data reducing its dimensionality. Reductions can be found in many scientific application domains such as image processing, liner-algebra, partial differential equations, computational geometry etc. They often found in the codes that test the convergence of iterative algorithms and are executed over and over again. Reductions are also found extensively in Monte Carlo simulations where averages and variances of a vast number of random simulations need to be computed. An unoptimized or poorly optimized reduction will become the bottleneck for whole program performance. In reductions, a binary operator is applied successively to all elements of the input set. This introduces dependences between loop iterations and forces them to execute sequentially. This binary operator is usually associative and commutative. These properties allow us to ignore the sequential dependence and parallelize reductions. Since the reduction operator is associative, we can perform multiple reduction operations in parallel and then combine them to produce the final output. The commutativity of the operator can further be used to optimize the reduction on parallel architectures such as GPUs.

To optimize reductions, we first need to identify them. Automatic detection of simple reductions with commonly used reduction operators such as addition, multiplication, min, max is quite trivial. There are some techniques proposed in the literature to detect them. However, when the reduction is performed on a user-defined data type or with a more complex operator it is very challenging to detect them and none of existing compiler can do this automatically. It is quite clear to the programmer identify reduction operations. Hence, we propose two extensions that allow the programmer to convey reduction information to the compiler. For a compiler to optimize reductions, it requires the identity value of reduction operator, reduction domain and the

actual reduction operator. The programmer can convey all these information using the proposed extensions with very little modifications to input code.

Even after the reduction is identified, it is not trivial to optimize them on massively parallel architectures such as GPUs. Reduction operations typically, have very low arithmetic intensity, performing one operation per load. Hence, they are bandwidth bound and require many extensive optimizations to achieve peak performance. There are many libraries like CUB, THRUST that provide a highly efficient implementation of reductions. They can achieve more than 90 percent of peak performance by performing various optimizations including using architecture-specific instruction to perform reduction and tuning parameters for a given architecture. Programmer can customize reduction by providing a reduction operator and an identity value. It is easy to map a single reduction into a library call but, when you have multiple reductions on same input data, or if you want to perform the reduction on selective elements of the input, then it is not efficient to use these libraries as they require multiple reduction calls or preprocessing of input data. For example, consider a program to find min and max elements of an array. We need two library calls, once to find the min and another one to find the max. This inefficient as one could use a single scan of the array to find both max and min. SLAMBench has a reduction kernel that performs 32 different reductions. MG benchmark of the NAS benchmark suite has a reduction kernel to find 10 largest and 10 smallest elements of an array. Using library calls in these cases will be highly inefficient. Although library APIs are highly efficient, they are not flexible enough to adapt to input program requirements.

Reductions are bound by the maximum available bandwidth of the device. We can improve the arithmetic intensity of reductions by performing certain loop transformations such as loop fusion. Performing two reductions at once is twice as fast as two separate reductions. In the latter case, input data is scanned twice, whereas in former input data is scanned only once and two reductions are performed per single load. In order to enable such transformations, we model reductions in polyhedral model. We propose a dependence based abstraction for modeling reductions in polyhedral model. This abstraction fits well into the existing polyhedral compilation toolchain while enabling loop optimizations for reductions.

We propose template based code generation for reductions. We have selected a reduction template that does have all the optimizations required to achieve peak performance. We adapt this template to match the reductions defined by the programmer. Since, we have complete information regarding reductions in the input program, we can precisely modify the template code even after we performed optimizations on reductions. We also auto-tune the template for a given GPU architecture to find optimal values for parameters such as block size, grid size and number reduction elements per threads. Thus, with our approach, we are able to generate highly efficient code for sequences of user-defined reductions that is portable across different architectures.

1.4 SLAMBench compilation

SLAM [NIH+11] is the main algorithm to perform real-time localization and dense mapping using depth information. SLAMBench [NBZ+14] is open-source benchmark for real-time dense SLAM algorithm. This is a large end-to-end benchmark with 14 different kernels that constructs a 3D model from the input depth images. There are several challenges for applying polyhedral computation techniques to such a large benchmark. We port the entire SLAMBench kernel to PENCIL intermediate language and use PPCG to automatically produce CUDA and OpenCL version of kernels. Several of these kernels are non-affine and hence cannot be directly expressed in polyhedral model. We use summary functions to wrap the non-affine core kernel parts while exposing the affine loops to polyhedral model. We also show the need of runtime in avoiding redundant data copies between device and host. We propose prl runtime library that can be used to express the array access information in non pencil regions. This information is used during the program execution to decide the need to transfer data between device and the host. This approach of writing kernels in high level intermediate language (PENCIL) rather than low level device specific languages such as CUDA or OpenCL enhances the programmer productivity while providing portable performance across different GPU devices and architectures. We show the adaptability of polyhedral compilation techniques for large benchmarks.

1.5 Deep learning pipeline compilation

In recent years there is been an exponential growth in deep learning techniques. Deep learning pipeline has all the computations that are affine and are amendable for polyhedral compilation. Typically, in deep learning pipeline consists executing a DAG of tensor computations millions of time during training phase. This computation DAG has many layers of same computation repeated multiple time with different sizes and shapes of tensors. We study advantages and challenges of using polyhedral compilation techniques for deep learning tensor computations. We extract polyhedral intermediate representation from TVM [CMJ+18], an end-to-end compiler stack that has support for many popular deep learning frameworks such as Tensorflow, MXNet, Keras etc. We identify two key challenges with using existing polyhedral compilation techniques with deep learning computations. First, the current state-of-the-art polyhedral scheduler is agnostic to the values of program parameters such as tensor sizes. Hence, it will produce the same schedule for all the different layers with drastically different tensor sizes and shapes. We propose new additions to the Pluto algorithm that models the actual data reuse. We show how to formulate data reuse into linear constraints for the ILP so that we find the schedule that maximizes data reuse. This additional constraints ensure that we find optimal schedule for any given problem sizes. The second problem that we address is improving the autotuning time.

The ILP based polyhedral scheduler automatically finds the best loop transformations. In order to map such computations to GPUs, we need to choose the value of various parameters such as tile sizes, block sizes, grid sizes, unroll factors etc.. Typically, we use an autotuner to find the optimal values of these parameters. The autotuner does an exhaustive search on the entire

exponential search space to find the optimal values. The optimal values of these parameters depends on the particular GPU architecture and problem size. We need to repeat the expensive exhaustive search for different problem sizes and GPU architectures. We propose a performance model based autotuning that drastically reduces time for autotuning from hours to minutes. We propose a performance model based approach to filter the worst candidates from the search space. We automatically build a performance model for each operator on a given architecture. The performance model will predict the execution time of kernel for any given size and parameter values. The performance model a MLP (Multi-layered-perceptron) model that is built from data collected by uniform sampling of the search space. We built an accurate performance model with this approach. We use this model on all the valid candidates on the search space and pick only a few hundred best candidates. Only these candidates are evaluated on the hardware to find the optimal parameter values.

In summary, these are the main contributions of the thesis:

- First class support for reductions in polyhedral model using reduction dependences, extend scheduler to support reductions and template based code generation for GPUs.
- Modeling spatial locality and extending ILP scheduler to handle multiple conflicting objectives.
- Polyhedral compilation of end-to-end real world 3D imaging SLAMBench benchmark.
- Data reuse volume extensions to ILP scheduler to adapt the schedule with problem sizes in the context of compiling deep learning pipeline in polyhedral framework.

2 Background

2.1 Polyhedral framework

The polyhedral framework is a linear algebraic representation of the program parts that are "sufficiently regular". It may represent arithmetic expressions surrounded by loops and branches whose conditions are affine functions of outer loop iterators and runtime constants [FL11]. These constants, referred to as *parameters*, may be unknown at compilation time and are treated symbolically. Expressions may read and write to multidimensional arrays with the same restrictions on the subscripts as on control flow. It has been the main drive for research on loop optimization and parallelization in the last two decades [Fea92b, Bas04, BHRS08a, GGL12a].

The polyhedral framework operates on individual executions of statements inside loops, or *statement instances*, which are identified by a named multidimensional vector, where the name identifies the statement and the coordinates correspond to iteration variables of the surrounding loops. The set of all named vectors is called the *iteration domain* of the statement. Iteration domains can be expressed as multidimensional sets constrained by Presburger formulas [PW94a]. For example, the code fragment in Figure 2.1 contains two statements, `S` and `R` with iteration domains $\mathscr{D}_{\mathtt{S}}(N) = \{\mathtt{S}(i,j) \mid 0 \leq i,j < N\}$ and $\mathscr{D}_{\mathtt{R}}(N) = \{\mathtt{R}(i,j,k) \mid 0 \leq i,j,k < N\}$ respectively. In this paper, we use parametric named relations as proposed in `iscc` [Ver11]; note that set vectors in $\mathscr{D}_{\mathtt{S}}$ and $\mathscr{D}_{\mathtt{R}}$ are prefixed with the statement name. Unless otherwise specified, we assume all values to be integer, $i, j, \cdots \in \mathbb{Z}$.

Polyhedral modeling of the control flow maps every statement instance to a multidimensional logical execution date [Fea92b]. The instances are executed following the lexicographic order of

```
 for (i = 0; i < N; ++i)
   for (j = 0; j < N; ++j)
S:   C[i][j] = 0.0;
 for (i = 0; i < N; ++i)
   for (j = 0; j < N; ++j)
     for (k = 0; k < N; ++k)
R:     C[i][j] += A[i][k] * B[k][j];
```

Figure 2.1 – Naive implementation of matrix multiplication.

their execution dates. This mapping is called a *schedule*, typically defined by piecewise(quasi-) affine functions over the iteration domain $\mathscr{T}_{\mathtt{S}}(\boldsymbol{p}) = \{\boldsymbol{i} \rightarrow \boldsymbol{t} \mid \{t_j = \phi_{\mathtt{S},j}(\boldsymbol{i}, \boldsymbol{p})\} \wedge \boldsymbol{i} \in \mathscr{D}_{\mathtt{S}}\}$, which are disjoint unions of affine functions defined on a finite partition of the iteration domain, allowing integer division by constants. They allow arbitrarily complex loop traversals and interleavings of statement instances. In this paper, $\boldsymbol{x}$ denotes a row vector and $\vec{x}$ denotes a column vector. Code motion transformations may be expressed either by introducing *auxiliary dimensions* [KP95] in the schedule or by using a *schedule tree* structure that directly encodes enclosure and statement-level ordering [VGGC14]. For example, the schedule that preserves the original execution order in Figure 2.1 can be expressed as $\mathscr{T}_{\mathtt{S}}(N) = \{\mathtt{S}(i,j) \rightarrow (t_1,t_2,t_3,t_4) \mid t_1 = 0 \wedge t_2 = i \wedge t_3 = j \wedge t_4 = 0\}$, $\mathscr{T}_{\mathtt{R}}(N) = \{\mathtt{R}(i,j,k) \rightarrow (t_1,t_2,t_3,t_4) \mid t_1 = 1 \wedge t_2 = i \wedge t_3 = j \wedge t_4 = k\}$. The first dimension is independent of the iteration domain and ensures that all instances of `S` are executed before any instance of `R`.

To preserve the program semantics during transformation, it is sufficient to ensure that the order of writes and reads of the same memory cell remains the same [KA02]. First, accesses to array elements (a scalar being a zero-dimensional array) are expressed as multidimensional relations between iteration domain points and named cells. For example, the statement `S` has one write access relation $\mathscr{A}^{\text{write}}_{\mathtt{S}\rightarrow\mathtt{C}} = \{\mathtt{S}(i,j) \rightarrow \mathtt{C}(a_1,a_2) \mid a_1 = i \wedge a_2 = j\}$. Second, pairs of statement instances accessing the same array element where at least one access is a write are combined to define a *dependence relation*. For example, the dependence between statements `S` and `R` is defined by a binary relation $\mathscr{P}_{\mathtt{S}\rightarrow\mathtt{R}} = \{\mathtt{S}(i,j) \rightarrow \mathtt{R}(i',j',k) \mid i = i' \wedge j = j' \wedge (i,j) \in \mathscr{D}_{\mathtt{S}} \wedge (i',j',k) \in \mathscr{D}_{\mathtt{R}}\}$. This approach relates all statement instances accessing the same memory cell and is referred to as *memory-based* dependence analysis. It is possible to compute exact *data flow* given a schedule using the *value-based* dependence analysis [Fea91]. In this case, the exact statement instance that wrote the value before a given read is identified. For example, instances of `R` with $k \neq 0$ no longer depend on `S`: $\mathscr{P}_{\mathtt{S}\rightarrow\mathtt{R}} = \{\mathtt{S}(i,j) \rightarrow \mathtt{R}(i',j',k) \mid i = i' \wedge j = j' \wedge k = 0 \wedge (i,j) \in \mathscr{D}_{\mathtt{S}} \wedge (i',j',k) \in \mathscr{D}_{\mathtt{R}}\}$.

A dependence is *satisfied* by a schedule if all the statement instances in the domain of its relation are scheduled before their counterparts in the range of its relation, i.e., dependence sources are executed before respective sinks. A program transformation is *valid*, i.e., preserves original program semantics, if all dependences are satisfied.

2.1.1 Finding Affine Schedules

Numerous optimization algorithms in the polyhedral framework define a closed form of all valid schedules and solve an optimization problem in that space. As they usually rely on integer programming, objective functions and constraints should be expressed as affine functions of iteration domain dimensions and parameters. Objectives may include: minimum latency [Fea92b], parallelism [BHRS08a], locality [BBK+08] and others.

Multidimensional affine scheduling aims to determine sequences of statement schedule functions of the form $\phi_{S_j} = \boldsymbol{i}\vec{c}_j + \boldsymbol{p}\vec{d}_j + D$ where $\vec{c}_j$, $\vec{d}_j$, D are (vectors of) unknown integer values. Each

such affine function defines one dimension of a multidimensional schedule.

Dependence Distances, Dependence Satisfaction and Violation Consider the affine form $(\phi_{\mathtt{R},j}(\boldsymbol{i},\boldsymbol{p}) - \phi_{\mathtt{S},j}(\boldsymbol{i},\boldsymbol{p}))$, defined for a dependence between `S` and `R`. This form represents the *distance* between dependent statement instances. If the distance is positive, the dependence is *strongly satisfied*, or *carried*, by per-statement scheduling functions ϕ. If it is zero, the dependence is *weakly satisfied*. The dependence with a negative distance that was not *carried* by any previous scheduling function is *violated* and the corresponding program transformation is invalid. For a schedule to be valid, i.e., to preserve the original program semantics, it is sufficient that it carries all dependences [KA02].

Farkas' Lemma Note that the target form of the schedule function contains multiplication between unknown coefficients $\boldsymbol{c}_j$, $\boldsymbol{d}_j, D$ and loop iterator variables $\mathbf{i}$ that may take any value within the *iteration domain*. This relation cannot be encoded in a *linear* programming problem. Polyhedral schedulers usually rely on the affine form of Farkas' lemma, a fundamental result in linear algebra that states that an affine form $\boldsymbol{c}\vec{x} + d$ is nonnegative everywhere in the (non-empty) set defined by $A\vec{x} + \vec{b} \geq 0$ iff it is a linear combination $\boldsymbol{c}\vec{x} + d \equiv \lambda_0 + \boldsymbol{\lambda}(A\vec{x} + \vec{b})$, where $\lambda_0, \boldsymbol{\lambda} \geq 0$. Applying Farkas' lemma to the dependence distance relations and equating coefficients on the left and right hand side of the equivalence gives us constraints on schedule coefficients $\boldsymbol{c}_j$ for the dependence to have non-negative distance, i.e., to be weakly satisfied by the schedule function, in the iteration domains.

Permutable Bands A sequence of schedule functions is referred to as *schedule band*. If all of these functions weakly satisfy the same set of dependences, they can be freely interchanged with each other without violating the original program semantics. Hence the band is *permutable*. Such bands satisfy the sufficient condition for loop tiling [IT88] and are also referred to as *tilable bands*.

2.1.2 Feautrier's Algorithm

Feautrier's algorithm is one of the first to systematically compute a (quasi-)affine schedule if there exists exits one [Fea92a, Fea92b]. It produces *minimal latency* schedules. The general idea of the algorithm is to find the minimal number of affine scheduling functions by ensuring that each of them carries as many dependences as possible. Once all dependence have been carried by the outer loops, the statement instances inside each individual iteration can be computed in any order, including in parallel. Hence Feautrier's algorithm exposes inner, fine-graph parallelism.

Encoding Dependence Satisfaction Let us introduce an extra variable e_k for each dependence in the program. This variable is constrained by $0 \leq e_k \leq 1$ and by $e_k \leq \phi_{\mathtt{R}_k,j}(\boldsymbol{i},\boldsymbol{p}) - \phi_{\mathtt{S}_k,j}(\boldsymbol{i},\boldsymbol{p})$,

where $\mathtt{S}_k$ and $\mathtt{R}_k$ identify the source and the sink of the k^{th} dependence, respectively. $e_k = 1$ iff the dependence is carried by the given schedule function.

Affine Transformations Feautrier's scheduler proceeds by solving linear programming (LP) problems using a special lexmin objective. This objective was introduced in the PIP tool and results in the lexicographically smallest vector of the search space [Fea88]. Intuitively, lexmin first minimizes the foremost component of the vector and only then moves on to the next component. Thus it can optimize multiple criteria and establish preference among them.

The algorithm computes schedule functions that carry as many dependences as possible by introducing a penalty for each non-carried dependence and by minimizing it. The secondary criterion is to generate small schedule coefficients, typically decomposed into minimizing sums of parameter and schedule coefficients separately. These criteria are encoded in the LP problem as

$$\operatorname{lexmin} \sum_k (1 - e_k), \sum_{j=1}^{n_s} \sum_{i=1}^{n_p} d_{j,i}, \sum_{j=1}^{n_s} \sum_{i=1}^{\dim \mathscr{D}_{S_j}} c_{j,i}, e_1, e_2 \dots e_k \dots \tag{2.1}$$

where individual $d_{j,i}$ and $c_{j,i}$ for each statement are included in the trailing positions of the vector in no particular order, $n_p = \dim \vec{p}$ and n_s is the number of statements. The search space is constrained, using the Farkas lemma, to the values $d_{j,i}$, $c_{j,i}$ that weakly satisfy the dependences. Dependences that are carried by the newly computed schedule function are removed from further consideration. The algorithm terminates when all dependences have been carried.

2.1.3 Pluto Algorithm

The Pluto algorithm is one of the core automatic parallelization and optimization algorithms [BHRS08a]. Multiple extensions have been proposed, including different search spaces [VMBL12], specializations and cost functions for GPU [VCJC+13b] and generalizations with guarantees of the existence of a solution [BAC16].

Data Dependence Graph Level On a higher level, Pluto operates on the data dependence graph (DDG), where nodes correspond to statements and edges together with associated relations define dependences between them. Strongly connected components (SCC) of the DDG correspond to the loops that should be preserved in the program after transformation [KA02]. Note that one loop of the original program containing multiple statements may correspond to multiple SCCs, in which case loop distribution is allowed. For each component, Pluto computes a sequence of permutable bands of maximal depth. To form each band, it iteratively computes affine functions linearly independent from the already computed ones. Linear independence ensures the algorithm makes progress towards a complete schedule on each step. Carried dependences are removed only when it is no longer possible to find a new function that weakly satisfies all of them, which

delimits the end of the permutable band. After removing some dependences, Pluto recomputes the SCC on the updated DDG and iterates until at least as many scheduling functions as nested loops are found and all dependences are carried. Components are separated by introducing an *auxiliary* dimension and scheduled by topological sorting.

Affine Transformation Level Affine transformation in Pluto is based on the observation that dependence distance $(\phi_{\mathrm{R},j}(\boldsymbol{i},\boldsymbol{p}) - \phi_{\mathrm{S},j}(\boldsymbol{i},\boldsymbol{p}))$ is equal to the reuse distance, i.e. the number of iterations of the given loop between successive accesses to the same data. Minimizing this distance will improve locality. Furthermore, zero distance implies that the dependence is not carried by the loop (all accesses are made within the same iteration) and thus does not prevent its parallelization. Pluto uses Farkas' lemma to define a parametric upper bound on the distance $(\phi_{\mathrm{R},j}(\boldsymbol{i},\boldsymbol{p}) - \phi_{\mathrm{S},j}(\boldsymbol{i},\boldsymbol{p})) \leq \boldsymbol{u}\vec{p} + w$, which can be minimized in an ILP problem as

$$\text{lexmin}\, u_1, u_2, \ldots, u_{n_p}, w, \ldots, c_{S,1}, \ldots$$

where $n_p = \dim \vec{p}$, and $c_{S,k}$ are the coefficients of $\phi_{\mathrm{S},j}$. The $c_{S,k}$ coefficients are constrained to be represent a *valid* schedule, i.e. not violate dependences, using Farkas' lemma. They are also restricted to have at least one strictly positive component along a basis vector of the null space of the current partial schedule, which guarantees linear independence. Note that it is sufficient to have a non-zero component rather than a strictly positive one, but avoiding a trivial solution with all components being zero may be computationally expensive [BAC16].

Fusion Auxiliary dimensions can be used not only to separate components, but also to group them together by assigning identical constant values to these dimensions. This corresponds to a *loop fusion*. By default, Pluto sets up an integer programming problem to find such constants that optimize the *smart fusion* heuristic. This heuristic tries to maximize fusion between components while keeping the number of required prefetching streams limited [BGDR10]. Pluto also features the *maximum fusion* heuristic, which computes weakly connected components of the DDG and keeps statements together unless it is necessary to respect the dependence.

Tiling For each *permutable band* with at least two members, Pluto performs loop tiling after the full schedule was computed. It is applied by introducing additional schedule dimensions after the band and relating them to those of the band through linear inequalities. New dimensions correspond to *point loops* and original ones correspond to *tile loops*. Various tile shapes are supported through user-selected options. For the sake of simplicity, we hereinafter focus on rectangular tiles.

Differentiating Tile and Point Schedule The default tile construction uses identical schedules for *tile* and *point loops*. Pluto allows to construct different schedules using the following two

post-affine modifications. First, a *wavefront* schedule allows to expose parallelism at the tile loop level. If the outermost schedule function of the band carries dependences, i.e., the corresponding loop is not parallel, then it may be replaced by a sum of itself with the following function, performing a *loop skewing* transformation. It makes the dependences previously carried by the second-outermost function to be carried by the outermost one instead, rendering the second one parallel. Such wavefronts can be constructed for one or all remaining dimensions of the band exposing different degrees of parallelism. Second, loop *sinking* allows to leverage locality and vectorizability of point loops. Pluto chooses the point loop `j` that features the most locality using the heuristic based on scheduled access relations $\mathscr{A} \circ \mathscr{T}$

$$j : L_j = n_S \cdot \sum_i (2s_i + 4l_i + 8v - 16o_i) \rightarrow \max, \tag{2.2}$$

where n_S is the number of statements in the loop, $s_i = 1$ if the scheduled access $\mathscr{A}_i \circ \mathscr{T}$ features spatial locality $a_{n_a} = kt_j + f(\boldsymbol{u}) + w, 1 \leq k \leq 4, n_a = \dim(\mathrm{Dom}\mathscr{A})$ and $s_i = 0$ otherwise; $l_i = 1$ if it yields temporal locality $a_{n_a} = f(\boldsymbol{u}) + w$ and $l_i = 0$ otherwise; $o_i = 1$ if it does not yield either temporal or spatial locality $s_i = l_i = 0$ and $o_i = 0$ otherwise; and $v = 1$ if $o_i = 0 \forall i$. The loop `j` with the largest L_j value is put innermost in the band, corresponding to the *loop permutation*. The validity of skewing and permutation is guaranteed by permutability of the band.

Pluto+ Recent work on Pluto+ extends the Pluto algorithm to prove its completeness and termination as well as to enable negative schedule coefficients [BAC16]. It imposes limits on the absolute values of the coefficients to simplify of the linear independence check and zero solution avoidance.

2.2 Polyhedral Scheduling in isl

Let us now present a variant of the polyhedral scheduling algorithm, inspired by Pluto and implemented in the `isl` library [Ver10]. We occasionally refer to the embedding of the scheduling algorithm in a parallelizing compiler called `ppcg` [VCJC+13b]. We will review the key contributions and differences, highlighting their importance in the construction of a unified model for locality optimization.

The key contributions are: separated specification of relations for semantics preservation, locality and parallelism; schedule search space supporting arbitrarily large positive and negative coefficients; iterative approach simultaneously ensuring that zero solutions are avoided and that non-zero ones are linearly independent; dependence graph clustering mechanism allowing for more flexibility in fusion; and the instantiation of these features for different scheduling scenarios including GPU code generation [VCJC+13b].[1] A technical report is available for the most detailed information about the algorithm and implementation [VJ17].

2.2.1 Scheduling Problem Specification in isl

The scheduler we propose offers more control by through different groups of relations suitable for specific optimization purposes:

- *validity relations* impose a partial execution order on statement instances, i.e., they are dependences sufficient to preserve program semantics;
- *proximity relations* connect statement instances that should be executed as close to each other as possible in time;
- *coincidence relations* connect statement instances that, if not executed at the same time (i.e., not coincident), prevent parallel execution.

In the simplest case, all relations are the same and match exactly the dependence relations of Pluto: pairs of statement instances accessing the same element with at least one write access. Hence they are referred to as *schedule constraints* within `isl`. However, only *validity* relations are directly translated into the ILP *constraints*. *Proximity* relations are used to build the objective function: the distance between related instances is minimized to exploit locality. The scheduler attempts to set the distance between points in the *coincidence* relations to zero, to expose parallelism at a given dimension of the schedule. The latter relations are also useful to inform the scheduler that certain instances may safely commute (atomically), even if dependences exist, removing those dependences from the former relations; it is a basis for *live range reordering* [VC16], which

[1] While many of these features have been available in `isl` since version `isl-0.06-43-g1192654`, the algorithm has seen multiple improvements up until the current version; we present these features as contributions specifically geared towards the construction of better schedules for locality and parallelism, for the first time.

removes *false* dependences induced by the reuse of the same variable for different values, when the live ranges of those values do not overlap.

2.2.2 Affine Transformations

Prefix Dimensions Similarly to Pluto, `isl` iteratively solves integer linear programming (ILP) problems to find permutable bands of linearly independent affine scheduling functions. It uses a lexmin objective, giving priority to initial components of the solution vector. Such behavior may be undesirable when these components express schedule coefficients: a solution with a small component followed by a very large component would be selected over a solution with a slightly larger first component but much smaller second component, while large coefficients tend to yield worse performance [PBB+11]. Therefore, `isl` introduces several leading components as follows:

- sum of all parameter coefficients in the distance bound;
- constant term of the distance bound;
- sum of all parameter coefficients in all per-statement schedule functions;
- sum of all variable coefficients in all per-statement schedule functions.

They allow `isl` to compute schedules independent of the *order of appearance* of coefficients in the lexmin formulation. Without the prefix, it would have also preferred the $(\phi_2 - \phi_1) \leq 0p_1 + 100p_2$ distance bound to $(\phi_2 - \phi_1) \leq p_1 + 0p_2$ bound because $(0, 100) \prec (1, 0)$, while the second should be preferred assuming no prior knowledge on the parameter values.

Negative Coefficients The `isl` scheduler introduces support for negative coefficients by substituting dimension x with its negative and positive part $x = x^+ - x^-$, where $x^+ \geq 0$ and $x^- \geq 0$ in the non-negative lexmin optimization. This decomposition is only performed for schedule coefficients c, where negative coefficients correspond to loop *reversal*, and for parameter coefficients of the bound u, connected to c through Farkas' inequalities. Schedule parameter coefficients and constants d can be kept non-negative because a polyhedral schedule only expresses a relative order. These coefficients delay the start of certain computation *with respect to* another. Thus a negative value for one statement can be replaced by a positive value for all the other statements.

ILP Formulation The `isl` scheduler minimizes the objective

$$\operatorname{lexmin} \sum_{i=1}^{n_p} (u_i^- + u_i^+), w, \sum_{i=1}^{n_p} \sum_{j=1}^{n_s} d_{j,i}, \sum_{j=1}^{n_s} \sum_{i=1}^{\dim \mathscr{D}_{S_j}} (c_{j,i}^- + c_{j,i}^+), \ldots \tag{2.3}$$

in the space constrained by applying Farkas' lemma to *validity* relations. Coefficients u_i and w are obtained from applying Farkas' lemma to *proximity* relations. Distances along *coincidence* relations are required to be zero. If the ILP problem does not admit a solution, zero-distance requirement is relaxed. If the problem remains unsolvable, `isl` performs band splitting as described below.

Individual coefficients are included in the trailing positions and also minimized. In particular, negative parts u_i^- immediately precede respective positive parts u_i^+. Lexicographical minimization will thus prefer a solution with $u_i^- = 0$ when possible, resulting in non-negative coefficients u_i.

Band Splitting If the ILP problem does not admit a solution, `isl` applies a variant of Feautrier's scheduler [Fea92b] using *validity* and *coincidence* relations as constraints [VJ17]. If the problem involved constraints based on *coincidence* relations and outer parallelism is not requested in configuration, it first relaxes such constraints and tries to find a solution. Otherwise, it finishes the current schedule band, removes relations that correspond to fully carried dependences and starts a new band.

2.2.3 Linear Independence

Encoding Just like Pluto, `isl` also computes a subspace that is orthogonal to the rows containing coefficients of the already computed affine schedule functions, but it does so in a slightly different way [VJ17]. Let $\boldsymbol{r}_k$ form a basis of this orthogonal subspace. For a solution vector to be linearly independent from previous ones, it is sufficient to have a non-zero component along at least one of these $\boldsymbol{r}_k$ vectors. This requirement is enforced iteratively as described below.

Optimistic Search `isl` tries to find a solution $\boldsymbol{x}$ directly and only enforces non-triviality if an actual trivial solution (zero) was found. More specifically, it defines *non-triviality regions* in the solution vector $\boldsymbol{x}$ that correspond to schedule coefficients. Each region corresponds to a statement and is associated with a set of vectors $\{\boldsymbol{r}_k\}$ described above. A solution is trivial in the region if $\forall k, \boldsymbol{r}_k\vec{x} = 0$. In this case, the scheduler introduces constraints on the signs of $\boldsymbol{r}_k\vec{x}$, invalidating the current (trivial) solution and requiring the ILP solver to continue looking for a solution. Backtracking is used to handle different cases, in the order $\boldsymbol{r}_1\vec{x} > 0$, then $\boldsymbol{r}_1\vec{x} < 0$, then $\boldsymbol{r}_1\vec{x} = 0 \wedge \boldsymbol{r}_2\vec{x} > 0$, etc. When a non-trivial solution is found, the `isl` scheduler further constrains the prefix of the next solution, $\sum_i u_i, w$, to be lexicographically smaller than the current one before continuing iteration. In particular, it enforces the next solution to have an additional leading zero.

This iterative approach allows `isl` to support negative coefficients in schedules while avoiding the trivial zero solution. Contrary to Pluto+ [BAC16], it does not limit the absolute values of coefficients, but instead requires the `isl` scheduler to interact more closely with the ILP solver.

This hinders the use of an off-the-shelf ILP solver, as is (optionally) done in R-Stream [VMBL12] and Pluto+ [BAC16]. Due to the order in which sign constraints are introduced, `isl` prefers schedules with positive coefficients in case of equal prefix. The order of the coefficients is also reversed, making `isl` prefer a solution with final zero-valued schedule coefficients. This behavior allows to prefer the original loop order in absence of a good transformation.

Although, in the worst case, this iterative approach considers an exponentially large number of sign constraints, it does not often happen in practice. As the validity constraints are commonly derived from an existing loop program, ensuring non-triviality for one region usually makes other validity-related regions non-trivial as well.

Slack for Smaller-Dimensional Statements When computing an n-dimensional schedule for an m-dimensional domain and $m < n$, only m linearly independent schedule dimensions are required. Given a schedule with k linearly independent dimensions, `isl` does not enforce linear independence until the last $(m-k)$ dimensions. Early dimensions may still be linearly independent due to validity constraints. At the same time, `isl` is able to find bands with linearly-dependent dimensions if necessary, contrary to Pluto, which enforces linear independence early.

2.2.4 Clustering

Initially, each strongly-connected component of the DDG is considered as a cluster. First, `isl` computes per-statement schedules inside each component. Then it selects a pair of clusters that have a *proximity* edge between them, preferring pairs where schedule dimensions can be completely aligned. The selection is extended to all the clusters that form a (transitive) validity dependence between these two. Then, the `isl` scheduler tries to compute a global schedule, between clusters, that respects inter-cluster validity dependences using the same ILP problem as inside clusters. If such a schedule exists, `isl` combines clusters after checking several profitability heuristics. Cluster combination is essentially loop fusion, where per-statement schedules are *composed* with schedules between clusters. Otherwise, it marks the edge as *no-cluster* and advances to the next candidate pair. The process continues until a single cluster is formed or until all edges are marked *no-cluster*. Clustering essentially corresponds to loop fusion, except that it allows for rescheduling of individual clusters with respect to each other. The final clusters are topologically sorted using the validity edges.

Clustering Heuristics Clustering provides control over parallelism preservation and locality improvement during fusion. When parallelism is the objective, `isl` checks that the schedule between clusters contains at least as many coincident dimensions on all individual clusters. Furthermore, it estimates whether the clustering is profitable by checking whether it makes the distance along at least one proximity edge constant and sufficiently small.

2.2.5 Additional Transformations

Several transformations are performed on the schedule tree representation outside the `isl` scheduler.

Loop tiling is an affine transformation performed outside the `isl` scheduler. In the `ppcg` parallelizing compiler, it is applied to outermost permutable bands with at least two dimensions and results in two nested bands: *tile loops* and *point loops* [VCJC+13b]. In contrast to Pluto, the `isl` scheduler pushes no other affine transformation to this level.

Parallelization using the `isl` scheduler takes the same approach as Pluto when targeting CPUs. For each permutable band, compiled syntactically into a loop nest, the outermost parallel loop is marked as OpenMP parallel and the deeper parallel loops are ignored (or passed onto an automatic vectorizer).

GPU code generation is performed as follows. First, loop nests with at least one parallel loop are stripmined. At most two outermost parallel *tile loops* are mapped to CUDA blocks. At most three outermost parallel *point loops* are mapped to CUDA threads. Additional placement strategies place temporally local on the GPU shared/local memory and registers, see [VCJC+13b].

3 Reductions in PENCIL

3.1 Overview of PENCIL

PENCILis a platform-neutral compute intermediate language intended as a target language for DSL compilers and as a high level portable implementation language for programming accelerators. A domain expert given high-level knowledge about standard operations in a given domain has a lot of information which could be useful for an optimizing compiler. Information regarding aliasing, parallelization, high level dataflow and other domain specific information will enable compiler to perform more accurate static analysis, additional optimizations and to generate efficient target-specific code. This information is typically difficult for a compiler to extract but that can be easily captured from a DSL, or expressed by an expert programmer. PENCILis a rigorously-defined subset of GNU C99 and provides language constructs, that enable communication of this domain-specific information to the PENCIL compiler.

PENCILwas designed with following main objectives:

Sequential Semantics. We choose sequential semantics in order to simplify DSL-to-PENCIL compiler development and the learning curve of an expert directly developing in PENCIL. Note that even with the sequential semantics the user can still express the parallelization information but will avoid committing to any particular pattern(s) of parallelism.

portability. Any standard non-parallelizing C99 compiler that supports GNU C attributes should be able to compile PENCIL. This ensures portability to platforms without OpenCL/ CUDA support and allows existing tools to be used for debugging sequential PENCILcode.

Ease of analysis. The language should simplify static code analysis, to enable a high degree of optimization. The main restriction of this is that the use of pointers is disallowed, except in specific cases.

Support for domain-specific information. PENCILshould provide facilities that allow a domain

expert or a DSL-to-PENCILcompiler to convey, in PENCIL, domain-specific information that can be exploited by the PENCILcompiler during optimization.

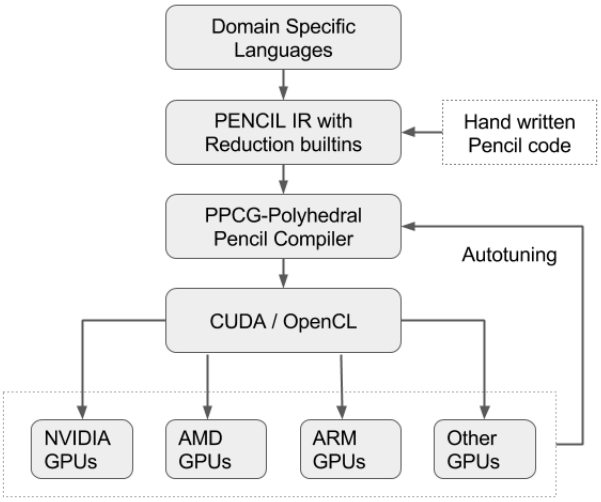

Figure 3.1 – A high level overview of the PENCILcompilation flow

Figure 3.1 shows a high level overview of our PENCILcompiler framework. At the top level a DSL program is translated into PENCILby a DSL-to-PENCIL compiler. We expect all domain-specific optimizations are performed before PENCILtranslation. We use a PPCG as our PENCILcompiler which is modified to handle all PENCILextensions. PPCG is a polyhedral optimization framework that performs loop nest transformations, parallelization, data locality optimization and generates efficient OpenCL or CUDA code. It extracts all the addition information provided through PENCILextensions and uses this information while performing above optimizations. This separation of domain specific optimizations and general loop level optimizations make PENCILa lightweight general-purpose language applicable to wide range of DSLs. Our framework also supports auto-tuning of generated code to find optimal parameter values for tile sizes, block sizes, grid sizes etc.. for a given target architecture.

We detail the most important restrictions imposed by PENCILfrom the point of view of enabling GPU-oriented compiler optimizations. The PENCILspecification [BCG+15] contains the rules in full.

Sized, non-overlapping arrays. Arrays must be declared using the C99 variable-length array syntax [ISO99]; array function arguments must be declared using `pencil_attributes`, a macro expanding to the `restrict` and the `const` C99 type qualifiers and to the `static` C99 keyword. During optimization, the PENCILcompiler thus knows the length of arrays, and that arrays do not overlap.

Pointer restrictions. Pointer declarations and definitions are allowed in PENCIL, but pointer manipulation (including arithmetic) is not, except that C99 array references are allowed as

arguments in function calls. Pointer dereferencing is also not allowed except for accessing C99 arrays. The restricted use of pointers is important for moving data between different address spaces of hardware accelerators, as it essentially eliminates aliasing problems.

No recursion. Recursive function calls are not allowed, because accelerator programming languages such as OpenCL forbid this.

Structured `for` loops. A PENCIL`for` loop must have a single iterator, an invariant start value, an invariant stop value and a constant increment (step). Invariant in this context means that the value does not change in the loop body. By precisely specifying the loop format we avoid the need for a sophisticated induction variable analysis. Such an analysis is not only complex to implement, but more importantly results in compiler analyses succeeding or failing under conditions unpredictable to the user.

The main constructs introduced by PENCILinclude the assume builtin function, the `independent` directive, summary functions and the kill builtin function. They are described here very briefly, for a complete description please refer to [BCG+15, BBC+15].

3.1.1 Summary Functions

Summary functions are used to describe the memory access patterns of (1) library functions called from PENCILcode, for which source code is not available for analysis, and (2) non-PENCILfunctions called from PENCILcode, as they are otherwise difficult to analyze. The use of summary functions enables more precise static analysis. Summary functions enable non static affine programs

Figure 3.2 shows an example use of summary functions. The code calls the function `fft32` (Fast Fourier Transform). This function only reads and modifies (in place) 32 elements of its input array `in`, it does not modify any other parts of the input array. Without a summary function the compiler conservatively assumes that the whole array passed to `fft32` is accessed for reading and writing. Such a conservative assumption prevents parallelization. The effect of function call `fft32` is summarized by the function `summary_fft32`. The pencil compiler derives accurate memory access information (reads and writes of 32 elements) `summary_fft32` enabling parallelization of the loop nest.

Summary function is a powerful construct. It enables polyhedral analysis and transformations for non affine code. Traditional polyhedral compilers can handle static affine code with only affine conditionals. For the codes with non affine code and data dependent conditionals one can create a function encapsulating such code and provide conservative memory accesses information through summary function. A polyhedral compiler can now perform analyses and transformations based on the memory access information in the summary function. We were able to pencilize many large benchmarks with non affine codes and data dependent conditionals using summary functions.

```
__attribute__((pencil_access(summary_fft32)))
void fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);

int ABF(int n, float in[pencil_attributes n][n][n])
{
  // ...
  for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
      fft32(i, j, n, in);
  // ...
}

void summary_fft32(int i, int j, int n,
           float in[pencil_attributes n][n][n]);
{
  for (int k = 0; k < 32; k++)
    __pencil_use(in[i][j][k]);
  for (int k = 0; k < 32; k++)
    __pencil_def(in[i][j][k]);
}
```

Figure 3.2 – Example code illustrating the use of summary functions

3.1.2 Assume Builtin

`__pencil_assume` is an intrinsic function `__pencil_assume(`e`)`, where e is a logical expression, indicates that e is guaranteed to hold whenever the control flow reaches the intrinsic. This knowledge is taken on trust by the PENCILcompiler, and may enable generation of more efficient code. An assume statement allows a programmer to communicate high level facts in the generated code.

A *general 2D convolution* in image processing is a a good example that demonstrates the use of `__pencil_assume`. This image processing kernel calculates the weighted sum of the area around each pixel using a kernel matrix for weights. For a given application if it is sufficient to consider that the size of the convolution matrix (the matrix that hols the convolution kernel) less then 15×15. This can be expressed using the assume builtin as follows:

```
__pencil_assume(kernel_matrix_rows <= 15);
__pencil_assume(kernel_matrix_cols <= 15);
```

3.1.3 Independent Directive

The `independent` directive is used to annotate loops. It indicates that the desired result of the loop execution does not depend in any way upon the execution order of the data accesses from different iterations. In particular, data accesses from different iterations may be executed simultaneously. In practice, the `independent` directive can be used to indicate that the marked loop does not have any loop carried dependence (i.e., it could be run in parallel).

3.1.4 PENCILKill

The `__pencil_kill` builtin function allows the user to refine dataflow information within and across any control flow region in the program. It is a polymorphic function that signifies that its argument (a variable or an array element) is dead at the program point where `__pencil_kill` is inserted, meaning that no data flows from any statement instance executed before the kill to any statement instance executed after.

3.2 Reduction built-ins

Let us now present the PENCILextensions to express user-defined reductions. Figure 3.3 shows a sample reduction code of complex number multiplications. Automatic techniques for the detection of reductions have been proposed; see [DSHB15] for a survey of the polyhedral ones. However, most of these techniques can only detect a simple reduction with standard operators such as sum, max, min, etc. Real-world applications often involve user-defined reductions with custom reduction operators and user-defined data types. Moreover, reductions might be applied only to a range of array indexes. Automatic detection techniques are expensive and fail to detect such complex user-defined reductions. Often the programmer or the code generator for a high-level, domain-specific language, readily knows all information related to reductions. Hence, we provide PENCILextensions that will allow a programmer to easily express arbitrary reductions.

We analyzed many benchmarks from different suites (SHOC, Rodinia, PolyBench, SLAMBench, etc.) and many DSLs (linear algebra, image processing, signal processing, etc.). Based on these, and considering the semantic constraints of embedding custom reduction information into a statically compiled imperative language with first order functions, we designed the following extensions that are natural and flexible enough to express all the reductions we have encountered. The programmer or DSL compiler needs to convey the following pieces of information regarding reductions to the underlying compiler: *reduction operator*, *identity element* and *reduction domain*. The reduction operator is a commutative and associative function of two arguments that returns one value of the same type. The identity element of the reduction operator is used to initialize the temporary variables that hold the intermediate result. The reduction domain represents the set of iterations for which the reduction operator is called, possibly spanning multiple nested loops.

Figure 3.4 shows a sample reduction kernel taken from the Srand benchmark of the Rodinia suite. Figure 3.5 shows the PENCILversion of Figure 3.4. All the information regarding reductions is expressed using the following two PENCILextensions: **`__pencil_reduction_var_init`** and **`__pencil_reduction`**.

3.2.1 Reduction Initialization Builtin

`__pencil_reduction_var_init` is an intrinsic function that is used to express the identity element of the reduction operator. The first argument of this function is the address of the reduction variable. A reduction variable can be a scalar or an array element of a built-in type or a user-defined data type as in the example in Figure 3.6. The second argument is a function that will take reduction variable as an input and initialize it with the identity element. Having a simple function to initialize reduction variable provides the flexibility to handle initialization of any user-defined data types such as complex numbers. The function provided in **`__pencil_reduction_var_init`** is called by the compiler whenever it wants to initialize the temporary variable required to compute the reductions in parallel. This intrinsic function also marks the beginning of reduction domain.

```
typedef struct COMPLEX {
   int a;
   int b;
} Complex;

Complex multiply(Complex x,Complex y) {
   Complex z;
   z.a = x.a*y.a - x.b*y.b;
   z.b = x.a*y.b + x.b*y.a;
   return z;
}

Complex Reduce(const int N, Complex input[N]){
   int i;
   Complex product = {1.0, 0.0};

   for(i=0;i<N;i++)
      product = multiply(product, input[i]);

   return product;
}
```

Figure 3.3 – Complex number multiplication

3.2.2 Reduction Builtin

__pencil_reduction is another intrinsic function that is used to express a single reduction operation. The first argument for this function must be the address of the reduction variable. The second argument must be the current reduction element. The third argument is a function that implements the reduction operator. This function must accept two variables which are of the same type as the reduction variable and return the result of reduction operator. This function is assumed to be both commutative and associative. Every reduction variable must have an associated **__pencil_reduction_var_init** and **__pencil_reduction**.

The reduction domain is the set of all the instances of **__pencil_reduction** function. **__pencil_reduction** and **__pencil_reduction_var_init** are flexible enough to express most user-defined reductions.

3.2.3 Related work

Many programming languages provide high-level abstractions to express user-defined reductions. Google's Map Reduce [DG08] framework provides parallelization API through which a user can specify custom reduction functions. Intel TBB [Rei07] provides parallel reduction templates that can be specialized through custom reduction methods. Similar to TBB, a user can express reductions in PPL [Mic] using parallel reduction templates. In both cases the user needs to pass explicitly the range of values to be reduced. Cilk++ [FHLLB09] supports a limited number of reduction operators which are used to eliminate the contention of shared reduction variables by performing reductions in a lock-free manner.

MPI [MPI96] includes support for several built-in reduction operators as well as the ability to define custom reduction operators in the context of distributed computing. In MPI, a user needs

```
1  int srand_reduction(int niter, int Nr, int Nc,
2                  float image[Nr][Nc]){
3     for (int iter=0; iter<niter; iter++) {
4        float sum = 0.0; //S1
5        float sum2 = 0.0; //S2
6
7        for (int i = 0; i < Nr; i++) {
8           for (int j = 0; j < Nc; j++) {
9              sum += image[i][j]; //S3
10             sum2 += image[i][j] * image[i][j]; //S4
11          }
12       }
13
14       float meanROI = sum / NeROI; //S5
15       float varROI = (sum2 / NeROI) - meanROI*meanROI;
16       float q0sqr = varROI / (meanROI*meanROI);
17
18       diffusion( Nr, Nc, q0sqr, image, c);
19    }
20 }
```

Figure 3.4 – Reduction from Rodinia's Srand benchmark

to create a custom function with fixed syntax which will be called by runtime while reduction is performed across multiple nodes. ZPL [DCS02] relies on overloading for the specification of user defined reductions. The user needs to create two functions with specific signatures, one to return the identity element and another function that implements reduction operator.

OpenMP 3.0 [Opea] and OpenACC [WSTaM12] support built-in reduction operators through declarative pragma syntax. OpenMP 4.0 [Opeb] now supports user-defined reductions as well, specifying the custom reduction function and identity element through `#pragma omp declare reduction`. One may use this user-defined reduction in a `#pragma omp parallel for reduction` directive. The pragma is associated with a specific for loop which forms the domain of the reduction. If multiple loop nests form the reduction domain, it is not directly possible to express it through OpenMP pragmas. The programmer needs to modify the loop structure to handle such cases. In our proposed approach the nesting depth of the reduction and its domain are inferred from reduction builtin locations, and such that the programmer does not need to modify the loop structure. Also, in OpenACC the reduction variable has to be scalar, while in our approach the reduction variable can be an array element with arbitrary subscript expression (e.g., a histogram computation).

```
void initialize(float *val) {
   *val = 0.0;
}
float reduction_sum(float v1, float v2){
   return v1 + v2;
}

void srand_reduction(int niter, int Nr, int Nc,
                float image[Nr][Nc]){
   for (int iter=0; iter<niter; iter++) {
      float sum;
      float sum2;
      __pencil_reduction_var_init(&sum2, initialize);
      __pencil_reduction_var_init(&sum, initialize);

      for (int i = 0; i < Nr; i++) {
         for (int j = 0; j < Nc; j++) {
            __pencil_reduction(&sum, image[i][j],
                               reduction_sum);
            __pencil_reduction(&sum2, image[i][j] * image[i][j],
                               reduction_sum);
         }
      }

      float meanROI = sum / NeROI;
      float varROI = (sum2 / NeROI) - meanROI*meanROI;
      float q0sqr = varROI / (meanROI*meanROI);

      diffusion( Nr, Nc, q0sqr, image, c);
   }
}
```

Figure 3.5 – Example from Rodinia's Srand reduction in PENCIL

```
void initialize(float *val){
   *val = 0.0;
}
float reduction_sum(float v1, float v2){
   return v1 + v2;
}

void kernel_correlation(int M, int N,
                  float data[M][N], float mean[M]){
   for (int j = 0; j < M; j++) {
      __pencil_reduction_var_init(&mean[j], initialize);
      for (int i = 0; i < N; i++)
         __pencil_reduction(&mean[j], data[i][j],
                        reduction_sum, NULL);
      mean[j] /= N;
   }
}
```

Figure 3.6 – Example from PolyBench's correlation benchmark

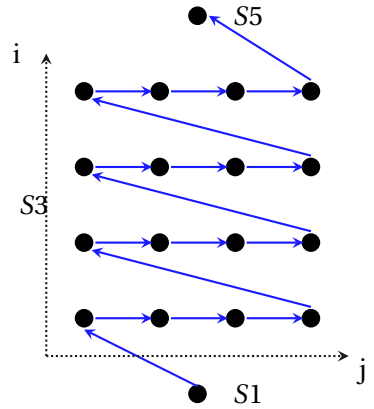

Figure 3.7 – Original reduction domain and dependences

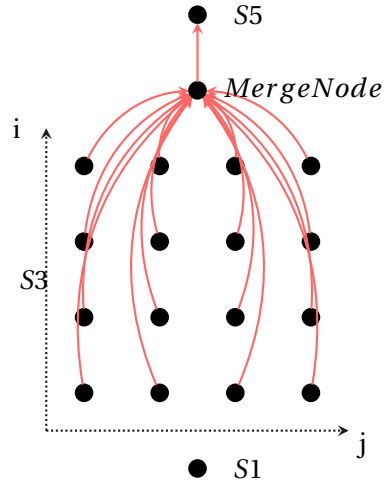

Figure 3.8 – Modified reduction dependences

3.3 Modeling Reductions in Polyhedral Model

3.3.1 Reduction Domain

The reduction domain is defined as the set of all iterations on which the reduction operator is applied. In pragma-based approaches such as OpenMP or OpenACC, the reduction domain is simply the loop that is annotated with a reduction pragma. In some benchmarks such as the Rodinia's Srand benchmark shown in Figure 3.4, the reduction domain is not a single for-loop. The programmer needs to coalesce (flatten) multiple loops into a single one before annotating it with pragmas. In PENCIL, since the programmer uses builtin functions that are not associated with a specific loop, no modifications of the control flow is required. Note that the reduction domain not necessary the same as the iteration domain of `__pencil_reduction` statement. For example, the iteration domain of the statement S3 in Figure 3.4 is $\{S3(iter, i, j) : 0 \leq iter \leq niter; 0 \leq i < nr; 0 \leq j < nc\}$ whereas, the reduction

domain for the reduction in S3 is $\{S3(i,j) : 0 \leq i < nr; 0 \leq j < nc\}$. The reduction operation always starts with the initialization of the reduction variable. The programmer explicitly marks the reduction variable initialization through `__pencil_reduction_var_init`. the reduction domain is derived from the iteration domains of `__pencil_reduction` and `__pencil_reduction_var_init` statements. For each `__pencil_reduction` statement, the reduction variable—the first argument of the function—is looked up find the dominating `__pencil_reduction_var_init` defining the same reduction variable. The reduction domain is simply the subset of the iteration domain of `__pencil_reduction` parametrized on the loop iterators of `__pencil_reduction_var_init`.

3.3.2 Reduction Dependences

Figure 3.7 depicts the Read-after-Write (RAW) dependences of a single reduction for the program in Figure 3.4. The arrows between iterations indicate the flow of data, i.e., a value produced by the source iterator is read at the destination iteration. Because the reduction variable is read and written in all iterations of the reduction domain, there is a serial dependence between them. This dependence between iterations of the reduction domain is called a reduction dependence. This dependence usually forces the serial execution of the reduction. Since the reduction operator is associative, we can perform the reductions in parallel and then combine the partial results to obtain a final reduction value. The associativity of the reduction operation is abstracted by relaxing the reduction dependences. We can precisely compute these reduction dependences from the reduction domain. The reduction of partial results and synchronization needed to produce the end result is modeled by introducing an additional node in the dependence graph, called a *merge node*, and adding a new dependence from all iterations in reduction domain to the merge node as shown in Figure 3.8. These are called *reduction isolation dependences* are necessary to prevent the use of the reduction variable before the reduction operation is complete when the reductions are performed in parallel. The merge node is inserted right after the reduction loops in the input AST. The relaxation of the serial reduction dependences into parallelism-friendly reduction isolation dependences will accurately model the flow of data for the parallel execution of reductions.

3.3.3 Reduction-Enabled Scheduling

A schedule represents the execution order of statement instances in a program. Various transformations are performed by changing the schedule. Dependences are used to find the valid set of schedules. A schedule is said to be valid if does not violate dependences in the input program, i.e., all the source iterations of dependences are executed before the destination iterations. State-of-the-art polyhedral compilers are seriously limited in the application of affine transformations in the presence of reductions, because of the serial reduction dependence. They do not exploit the associativity of the reduction operator. The explicit dependence manipulation as explained in the previous section enables transformations such as tiling and parallelization of reduction loops.

Because the serial reduction dependences are relaxed, the polyhedral scheduler can now safely reorder the reduction iterations.

A practical and automatic scheduling algorithm like Pluto takes a dependence graph as input and recursively constructs schedule. At each level of the recursion, the algorithm first checks for the components that do not depend on each other and hence can be scheduled independently. Within each component, the algorithm uses an ILP solver to construct a sequence of one-dimensional affine functions (hyperplanes), such that each of these functions independently respect all dependences and are optimal based on heuristics such as induced communication. After the construction of a band is completed, the dependence graph is updated to only contain dependences that are mapped to the same values by the current hyperplane, and the process is repeated until the number of hyperplanes found is equal to the dimensionality of the loop. Our dependence based abstraction of reductions fits well with the automatic scheduling algorithms enabling affine transformations for user defined reductions. Because the relaxation of serial reduction dependences, the reduction iterations can now be reordered and parallelized. The new merge dependences represent the required data-flow of combining the partial reduction results to produce the final reduction value. It is essential for the scheduler to know about the cost of performing reductions in parallel and merge dependences accurately represent this cost. For example, consider a loop nest with a parallel loop and a reduction loop. The scheduler should choose the parallel loop with no loop-carried dependences as the outermost loop rather than the reduction loop because of the cost of merge dependences. Hence, with this approach reduction parallelism is exploited and yields communication-free parallelism. Many of the previous approaches that just relax the reduction dependence have the problem of treating both types of parallelism equally which could be lead to poor schedules and additional communication.

3.3.4 Related work

Modeling reductions was commonly done implicitly, e.g., by ignoring the reduction dependences during a post parallelization step [Jou86, JD89, PP94, RF93, PE95, RP99, XKH04, VSHS14]. The first one to introduce reduction dependences, were Pugh and Wonnacott [PW94b]. Similar to most other approaches [RF94, SKN96, RF00, GR06] the detection and modeling of reductions was performed on imperative statements and utilizing a precise but costly access-wise dependence analysis. In the works of Redon and Feautrier [RF00, RF94] the reductions are modeled as Systems of Affine Recurrence Equations (SAREs). Array expansion allows the elimination of memory-based dependences induced by reductions and facilitate the recognition of induction patterns. They also propose a polyhedral scheduling algorithm that optimally schedules reductions together with other statements, assuming reductions are computable in single time step. Such atomic reduction computation simplifies scheduling choices while preventing schedules that reorder or interleave reduction statement instances with other statement instances. Zou et al. [ZR12] extended the work of Redon and Feautrier [RF94] by removing the restriction of an atomic reduction computation. Their scheduling algorithm tries to minimize the latency while scheduling reductions. Compared to these works our dependence based abstraction works on

existing practical polyhedral scheduling algorithms such as Pluto [BHRS08b]. Our approach does not require any preprocessing such as array expansion while dependences accurately model the specific atomicity constraints of reductions.

In contrast to polyhedral optimizations for reductions, Gupta et al [GR06] propose techniques to exploit the associativity and commutativity of reductions to decrease the complexity of a computation in the context of dynamic programming. Their method reuses intermediate results of reduction computations.

Stock et al. [SKG+14] describe how reduction properties can be used to reorder stencil computations, to better exploit register reuse and to eliminate loads and stores. However neither do they describe the detection method nor does their method enable scheduling for generic reductions.

Doerfert et al. [DSHB15] propose compiler techniques to automatically detect reductions on LLVM IR. They also propose a model to relax memory-based dependences to enable polyhedral scheduling in the presence of reductions. However, their techniques cannot detect reductions on arbitrary data types such as complex number multiplication and their reduction modeling do not account for the cost of reductions while scheduling them.

3.4 Code Generation

Parallelizing reductions on GPUs is a challenging task. Reductions typically have low arithmetic intensity performing just one operation per memory load and hence are bound by the device's maximum memory bandwidth. The performance of reductions is often measured by the effective bandwidth achieved by a given implementation. Vendor and highly tuned libraries such as CUB and Thrust provide an optimized implementation of reductions achieving performance above 90% percent of the peak bandwidth.

3.4.1 Optimizations for Reduction code on GPUs

The problem of Optimizing reductions in CUDA is well studied and the list of all the optimizations and their impact on performance for a single reduction is explained in [Mar]. The important optimizations are listed below.

Kernel Decomposition. The reduction is parallelized at two levels, matching the thread hierarchy of the GPUs. At the first level, each thread block is allocated a portion of the reduction domain. Within each thread block, a tree-based decomposition is used to perform local reductions and to produce partial results. These partial results are stored in global memory. At the second level, another kernel is launched to reduce the partial results and produce the final value.

Shared memory utilization. Within each thread block, multiple threads use shared memory to keep the intermediate reduction values. The tree-based reduction is performed on these values while avoiding divergent branches. Shared memory bank conflicts are avoided by reordering memory loads making use of the reduction operator's commutativity.

Complete unrolling. The kernel is specialized by completely unrolling the reduction tree. This is done using templates to generate a specialized kernel for different block sizes. This eliminates unnecessary control flow and synchronization between the threads in a thread block as the reduction proceeds.

Using shuffle instructions. Shuffle instructions [Nvic] can be used to accelerate fine-grained reductions within a warp. These special instructions control the exchange of data between threads, eliminating data accesses to shared memory and the associated synchronizations.

Multiple reductions per thread. Each thread loads and reduces multiple elements into shared memory before the tree-based reduction in shared memory. More work per thread will help to hide the memory latency. The optimal number of elements per thread depends on the architecture and is determined by tuning this parameter along with the number of blocks and threads per block.

3.4.2 Template Specialization for User-defined Reductions

A generic polyhedral optimizer such as PPCG can perform various optimizations such as shared memory allocation and register promotion, memory coalescing, etc.for generic programs [VJC+13]. It uses heuristics to determine the profitability of such optimizations. However, it cannot perform all the specialized optimizations required for highly efficient reductions. Some of these optimizations are specific to reductions, and applicable only to specific hardware or device characteristics. Hence it is difficult to come up generic heuristics for these optimizations in order to make them applicable to generic programs. These optimizations are crucial to achieve close-to-peak performance on GPUs. Hence, we follow a template-based approach for generating efficient code for user-defined reductions. We precisely identify the reductions through programmer-provided constructs, hence can generate efficient reduction code capturing all the optimizations above, even on user-defined reductions.

User-defined reduction operator. A user-defined reduction is characterized by reduction operator, the identity element of the operator and reduction domain. The new reduction operator is expressed in a separate PENCILfunction. It is easy to adapt the parallel template to user-defined reductions by replacing the reduction operation with the user-defined function, and similarly for the partial value initialization. All these function calls are inlined to eliminate overhead. The reduction domain is equally distributed among multiple thread blocks.

Shared memory allocation. In the parallel reduction template, each thread will be allocated temporary storage in the shared memory to store the partial results. The size of shared memory required per thread is equal to the size of the reduction variable. It is important to compute the total required size for all threads. The total shared memory available varies from device to device. The available shared memory limits certain transformations and thread block sizes. The maximum amount of shared memory on the device is an input to our framework. We then compute the shared memory required to implement the reduction and use it to calculate the maximum thread block size.

Fusing multiple reductions. The template can also be adapted to generate code for multiple fused reductions. Fusion can help increase the arithmetic intensity of the reduction. We use the following heuristic to determine when to fuse. Two reductions are fused if there is an overlap with global data accessed between them. For example, if two separate reductions operate on same input array, then these two reductions are fused and a single reduction template is generated. This optimization is always profitable since by fusion we are enabling data reuse in shared memory. Existing polyhedral techniques can be used to assess the correctness of such a fusion transformation, taking into account all side-effects involved in associated computations involved in the reductions (inductively or not). Code generation for fused reductions is a straightforward extension of the above-mentioned technique: the shared memory required is the sum of shared memory requirement for individual reductions and the two reduction functions are called one after another in the

template.

Auto-tuning. The performance of the template depends on the target GPU. The following two parameters are tuned to obtain optimal performance on a given architecture: number of threads, and number of thread blocks. The reduction template supports powers of two only for the number of threads, so there are only few values considered starting from 2 and up to the maximum allowed by the device. The maximum is also limited by the required shared memory for the reduction. The thread block size varies over the 2 to 256 interval.

3.5 Experimental Evaluation

We implemented PENCILreduction support in a developmental branch of PPCG. Our framework takes a C program with PENCILfunctions as input, with reduction kernels according to the specification. It generates CUDA code automatically, which is then auto-tuned to a particular GPU architecture. Since the search space for auto-tuning is relatively small for our reduction template, we could conduct an exhaustive search of the optimization space within minutes on all examples.

We compare the performance of the generated code with highly tuned libraries such as CUB v1.5.1 [Nvia] and Thrust v1.8.1 [Nvib]. Both of these libraries provide APIs for performing reductions. The user can customize them by specifying a reduction operator and identity value. Note that these libraries are optimized for a particular GPU architecture. CUB, for example, has an internal database of the optimal values for thread block size, number of threads and number of items per thread for all known architectures. We also compare the performance with the OpenACC PGI 2015 compiler [The], which provides high-level directives, including reduction pragmas, to program accelerators. Note that the PGI OpenACC compiler currently only supports a limited set of pre-defined reduction operators.

We evaluate performance on the following three GPU architectures: a desktop NVIDIA GTX 470 with peak memory bandwidth of 133.9 GB/s, the more advanced NVIDIA Quadro K4000 with peak bandwidth of 134.9 GB/s, and the low power embedded GPU NVIDIA Jetson TK1 with peak bandwidth of 14.78 GB/s. We do not present OpenACC numbers on TK1 because the PGI compiler does not support this this architecture. All the benchmarks are compiled with the NVIDIA CUDA 7.5 toolkit, with the `-O3` optimization flag. Reduction performance is measured as the effective bandwidth utilized by the benchmark, and is computed using

$$\text{Bandwidth} = \text{InputArraySize}/\text{ReductionTime}.$$

Each benchmark is run 100 times and the average of these times is taken as the reduction time. We also perform a dry-run before benchmarking to hide device configuration and kernel compilation time. In the graphs below, performance numbers are presented as a percentage of peak bandwidth of the device.

3.5.1 Single Reduction Kernel

The first benchmark is a single sum reduction over an array of 2^{22} elements. This corresponds to a single call in CUB and Thrust, and to a single for-loop marked with a reduction pragma in OpenACC. The performance of the sum benchmark for three different data types is shown in Figures 3.9, 3.10 and 3.11, abbreviated as SumInt, SumFloat and SumDouble. CUB achieves an impressive 92.4% of the peak device bandwidth for int and 81.5% for double on the GTX 470. The performance of Thrust is slightly lower at 73.8% for int and float. OpenACC is only able to achieve 23.7% for int and 45.2 for double, whereas the PENCILreduction code generated by our

framework achieves 90.2% of the peak.

The performance of CUB on the embedded TK1 is only 37.5% of peak for int and 65.5% for double, whereas PENCILcode achieves 78.9% and 79.91%, respectively. We suspect that both CUB and Thrust libraries are not tuned for the TK1 platform, whereas our generated code benefited from auto-tuning. This shows the importance and effectiveness of auto-tuning even after all optimizations have been applied and a suitable template is being used. Table 3.1 collects the optimal values for the number of thread blocks and the number of threads. Note that the exhaustive search does not take much time because of the limited search space for reduction templates.

Benchmark	GTX 470	Quadro K4000	TK1
SumInt	84,128	128,128	222,128
SumFloat	84,128	128,128	103,512
SumDouble	120,256	124,64	253,256

Table 3.1 – Optimal parameter values for the number of thread blocks and for the the thread block size

3.5.2 Srand Reduction kernel

The Srand reduction kernel shown in Figure 3.4 consists of two reductions on the same array. It is quite straightforward to express such reductions in OpenACC and PENCIL, whereas CUB or Thrust involve two library calls along with some additional processing. One reduction call to compute sum, an intermediate step to compute the square of the input, and then another reduction call to compute the sum of squares. This is clearly inefficient as input array is traversed twice to perform reductions separately. This kernel illustrates the limitations of library-based approaches in real world applications: Figures 3.9, 3.10 and 3.11 show the performance of the Srand kernel abbreviated as SrandInt, SrandFloat and SrandDouble on different GPU architectures. The performance of PENCILand OpenACC is almost identical to the case of a single reduction kernel. Because the array is scanned only once, reduction time does not vary much. On the other hand, for both CUB and Thrust, the time taken is twice that of a single reduction because of the two array traversals. This kernel illustrates the benefits of fusing multiple reductions, as evaluating multiple reduction operators per memory access made much more effective use of the bandwidth. The OpenACC compiler was also able to fuse the two reductions because both reductions appeared in the same loop. This restriction does not apply to our framework, which supports the fusion of different reduction kernels.

3.5.3 SLAMBench reduction kernel

SLAMBench [NBZ+14] is a computer vision benchmark for 3D modeling and tracking. The benchmark contains a reduction kernel with 32 different reductions. It is straightforward to

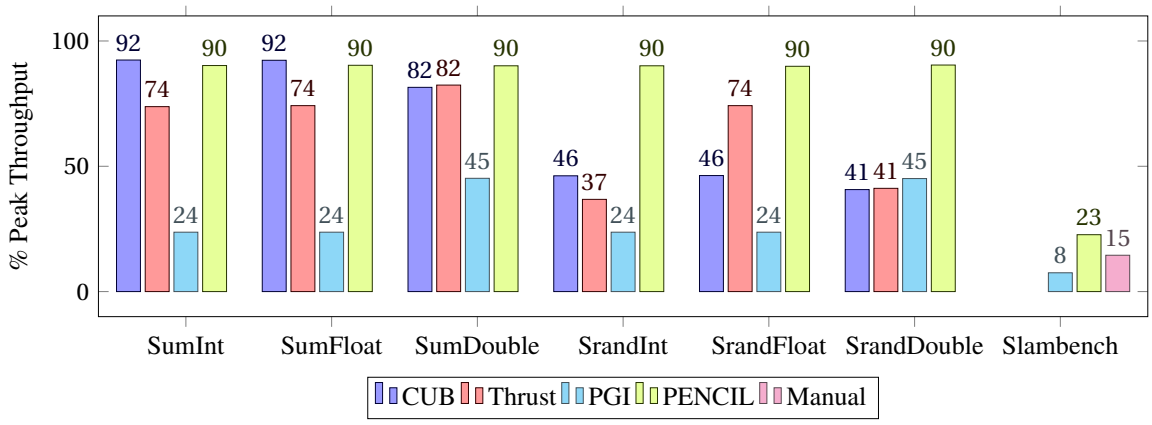

Figure 3.9 – Performance on NVIDIA GeForce GTX 470

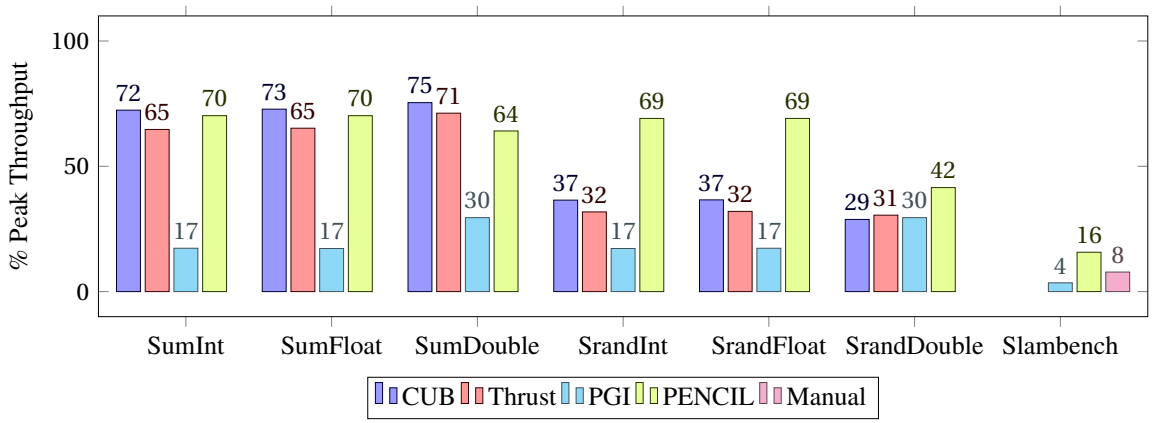

Figure 3.10 – Performance on NVIDIA Quadro K4000

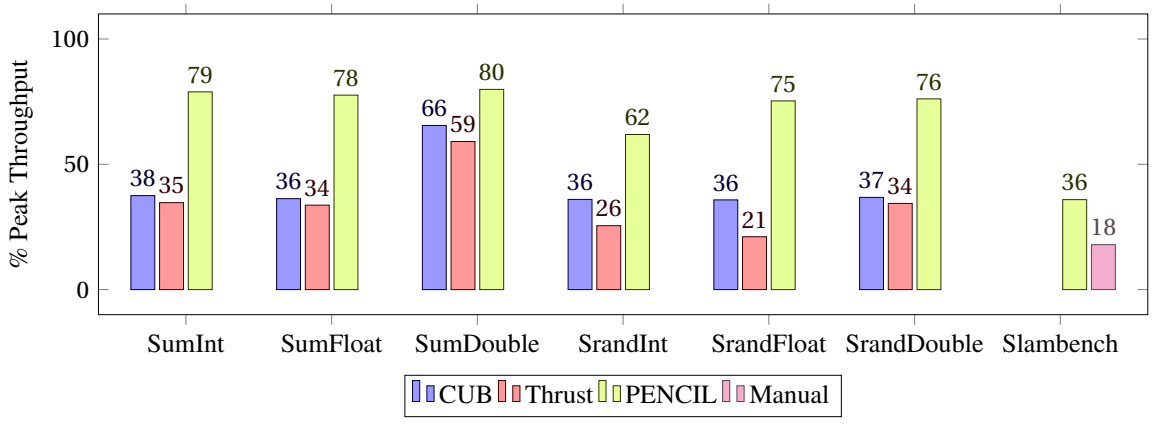

Figure 3.11 – Performance on NVIDIA TK1

port these reductions in PENCILand OpenACC whereas it is tedious and inefficient to do so in CUB and Thrust, as they involve multiple calls to the reduction API. The comparison of our framework with OpenACC and with SLAMBench's manual CUDA implementation is shown in Figures 3.9, 3.10 and 3.11. Because there are 32 reductions performed for a single element,

the memory bandwidth is no longer the bottleneck. PENCILoutperforms both OpenACC and the manual implementation. The latter is almost twice as fast as OpenACC. The SLAMBench implementation uses shared memory to store intermediate reduction values while reductions are performed in parallel.

Benchmark	OpenACC	PENCIL
GTX 470	0.51×	1.53×
Quadro K4000	0.45×	2.00×
TK1	–	1.97×

Table 3.2 – Speedup of the SLAMBench reduction kernel relative to the manual implementation

4 SLAMBench

SLAM systems aim to perform real-time localisation and mapping "simultaneously" for a sensor moving through an unknown environment. SLAM could be part of a mobile robot, enabling the robot to update its estimated position in an environment, or an augmented reality (AR) system where a camera is moving through the environment, allowing the system to render graphics at appropriate locations in the scene (e.g. to illustrate repair procedures in-situ or to animate a walking cartoon character on top of a table). Localisation typically estimates the location and pose of the sensor (e.g. a camera) with regard to a map which is extended as the sensor explores the environment.

The KinectFusion [NIH+11] algorithm utilises a depth camera to perform real-time localisation and dense mapping. A single raw depth frame from these devices has both noise and holes. KinectFusion registers and fuses the stream of measured depth frame obtained as the scene is viewed from different viewpoints into a clean 3D geometric map. kinectFusion normalizes each incoming depth frame and applies a bilateral filter (Preprocess); before computing a point cloud (with normals) for each pixel in the camera frame of reference. Next, KinectFusion estimates (Track) the new 3D pose of the moving camera by registering this point cloud with the current global map using a variant of iterative closest point (ICP). Once the new camera pose has been estimated, the corresponding depth map is fused into the current 3D reconstruction (Integrate). KinectFusion utilises a voxel grid as the data structure to represent the map, employing a truncated signed distance function (TSDF) to represent 3D surfaces. The 3D surfaces are present at the zero crossings of the TSDF and can be recovered by a raycasting step, which is also useful for visualising the reconstruction. The key advantage of the TSDF representation is that it simplifies fusion of new data into existing data to the calculation of a running average over the current TSDF volume and the new depth image. KinectFusion has been adopted as a major building block in lot of recent SLAM systems.

SLAMBench, is a benchmark that provides portable, but untuned, KinectFusion [NIH+11] implementations in C++ (sequential), OpenMP, CUDA and OpenCL for a range of target platforms. SLAMBench includes techniques and tools to validate an implementation's accuracy using the

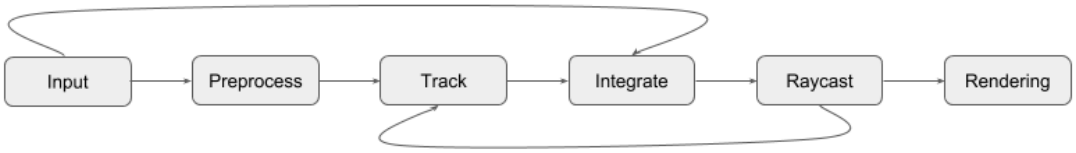

Figure 4.1 – A high level overview of the SLAMBench pipeline

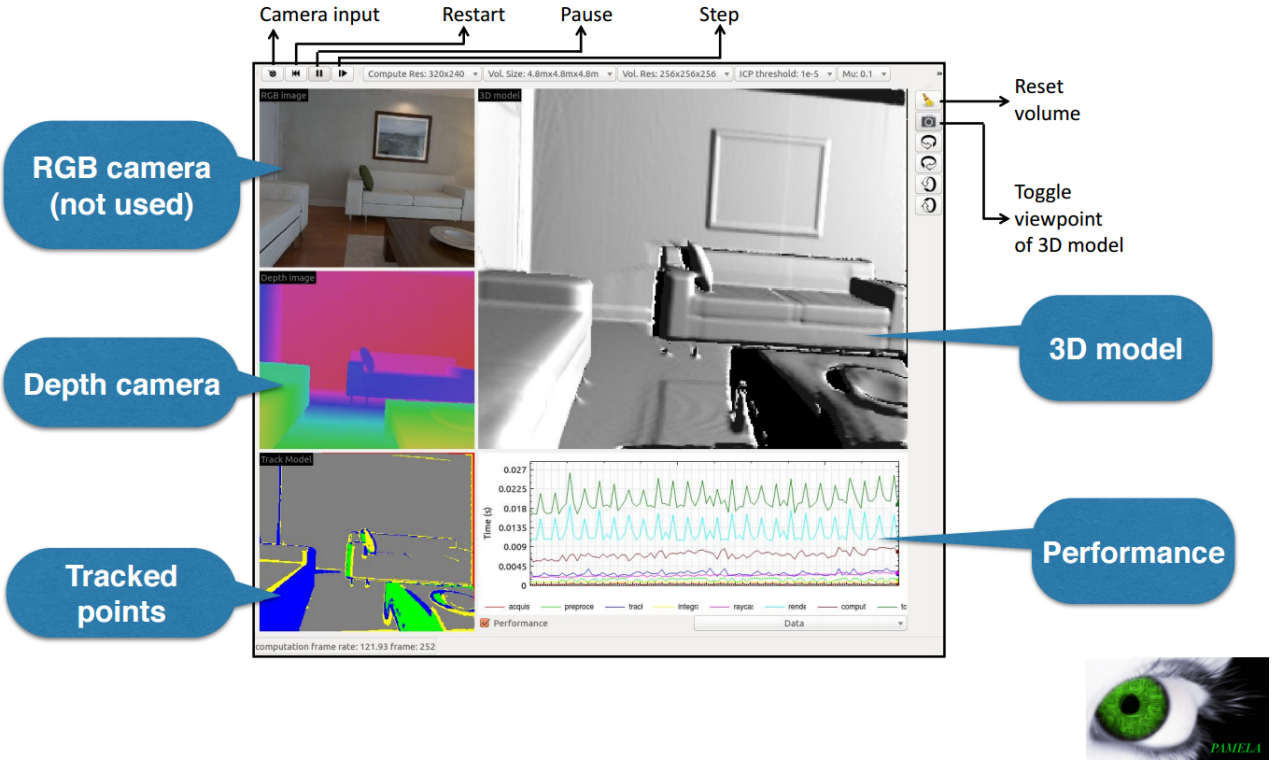

Figure 4.2 – SLAMBench GUI

ICL-NUIM dataset [HWMD14] along with algorithmic modifications to explore accuracy and performance tradeoffs. ICL-NUIM [HWMD14] is a high-quality synthetic dataset providing RGB-D sequences for 4 different camera trajectories through a living room model. An absolute trajectory error (ATE) is calculated as the difference between the ground truth and the estimated trajectory of a camera produced by a SLAM implementation, this enables system accuracy to be measured at each frame.The ICL-NUIM benchmark provides not only a number of realistic pre-rendered sequences, but also open source code that can be used by researchers in order to generate their own test data as required. As the unmodified ICL-NUIM dataset is the input of the SLAMBench benchmark any user can run SLAMBench in a straightforward way on other datasets.

Kernels	Pipeline	Pattern	In	Out	%
acquire	Acquire	n/a	pointer	2D	0.03
mm2meters	Preprocess	Gather	2D	2D	0.06
bilateralFilter	Preprocess	Stencil	2D	2D	33.68
halfSample	Track	Stencil	2D	2D	0.05
depth2vertex	Track	Map	2D	2D	0.11
vertex2normal	Track	Stencil	2D	2D	0.27
track	Track	Map/Gather	2D	2D	4.72
reduce	Track	Reduction	2D	6x6	2.99
solve	Track	Sequential	6x6	6x1	0.02
integrate	Integrate	Map/Gather	2D/3D	3D	12.85
raycast	Raycast	Search/Stencil	2D/3D	2D	35.87
renderDepth	Rendering	Map	2D	2D	0.12
renderTrack	Rendering	Map	2D	2D	0.06
renderVolume	Rendering	Search/Stencil	3D	2D	9.18

Table 4.1 – SLAMBench Kernels

4.1 SLAMBench Kernels

Table 4.1 summarises the 14 main computationally significant computer vision kernels contained in SLAMBench The following is a description of each kernel:

- **acquire**: acquires a new RGB-D frame. This input step is included explicitly in order to account for I/O costs during benchmarking, and for real applications.
- **mm2meters**: transforms a 2D depth image from millimeters to meters. If the input image size is not the standard 640x480, only a part of the image is converted and mapped into the output.
- **bilateralFilter**: is an edge-preserving blurring filter applied to the depth image. It reduces the effects of noise and invalid depth values.
- **halfSample**: is used to create a three-level image pyramid by sub-sampling the filtered depth image. Tracking solutions from low resolution images in the pyramid are used as guesses to higher resolutions.
- **depth2vertex**: transforms each pixel of a new depth image into a 3D point (vertex). As a result, this kernel generates a point cloud.
- **vertex2normal**: computes the normal vectors for each vertex of a point cloud. Normals are used in the projective data association step of the ICP algorithm to calculate the point-plane distances between two corresponding vertices of the synthetic point cloud and a new point cloud.
- **track**: establishes correspondence between vertices in the synthetic and new point cloud.

- **reduce**: adds up all the distances (errors) of corresponding vertices of two point clouds for the minimisation process. On GPUs, the final sum is obtained using a parallel tree-based reduction.
- **solve**: performs a singular value decomposition on the CPU that solves a 6x6 linear system. A 6-dimensional vector is produced to correct the new estimate of camera pose.
- **integrate**: integrates the new point cloud into the 3D volume. It computes the running average used in the fusion.
- **raycast**: computes the point cloud and normals corresponding to the current estimate of the camera position.
- **renderDepth**: visualises the depth map acquired from the sensor using a colour coding.
- **renderTrack**: visualises the result of the tracking. For each pixel different colours are associated with one of the possible outcomes of the tracking pass
- **renderVolume**: visualises the 3D reconstruction from a fixed viewpoint or a user specified viewpoint.

In addition to these kernels, the SLAMBench pipeline also contains two initialisation kernels not shown in Table 4.1 namely generateGaussian and InitVolume. These kernels are part of an initialisation step that is performed only at startup. generateGaussian generates a Gaussian bell curve and stores it in a 1D array; initVolume initialises the 3D volume.

4.2 Pencilizing SLAMBench Kernels

We were able to port seven out of eleven kernels directly to pencil without using any pencil directives, as shown in table 4.2. All of these kernels had only affine accesses and are readily handled by PPCG compiler. We only needed to surround with $scop$ and $endscop$ pragmas. Additionally, we used `__pencil_assume` builtins to specify additional information related to size of loop bounds and image sizes, as shown in 4.3. This information is used by the PPCG compiler to simplify generated code. With the information provided through `__pencil_assume` PPCG was able to generate the kernel without any unnecessary conditionals that is very similar to manually written kernel is SLAMBench. We are able to achieve only 0.1 FPS(Frames per Second) compared to 68 FPS with manual SLAMBench kernels. This is because PPCG was not able to generate kernels for $track$, $reduce$ and $integrate$. These main computational kernels were executed on CPU resulting in bad performance.

4.3 Summary functions

The $integrate$ and $track$ kernels have non-affine accesses and PPCG was not able to analyze the dependences. We encapsulate the core computation in another function and use summary

```
int mm2meters_pencil(uint outSize_x, uint outSize_y,
        float out[restrict const static outSize_y][outSize_x],
        uint inSize_x, uint inSize_y,
        const ushort in[restrict const static inSize_y][inSize_x],
        int ratio)
{
#pragma scop
      {
            __pencil_assume(outSize_y < 960);
            __pencil_assume(outSize_x < 1280);
            __pencil_assume(outSize_y % 120 == 0);
            __pencil_assume(outSize_x % 160 == 0);
            __pencil_assume(outSize_x > 0);
            __pencil_assume(outSize_y > 0);
            __pencil_assume(inSize_x > 0);
            __pencil_assume(inSize_y > 0);
            for (uint y = 0; y < outSize_y; y++) {
                  for (uint x = 0; x < outSize_x; x++) {
                        int xr = x * ratio;
                        int yr = y * ratio;
                        out[y][x] = in[yr][xr] / 1000.0f;
                  }
            }
      }
#pragma endscop
      return 0;
}
```

Figure 4.3 – Pencilized mm2meters kernel

Kernels	Building Block	Pattern	PPCG kernel
mm2meters	Preprocess	Gather	yes
bilateralFilter	Preprocess	Stencil	yes
halfSample	Track	Stencil	yes
depth2vertex	Track	Map	yes
vertex2normal	Track	Stencil	yes
track	Track	Map/Gather	no
reduce	Track	Reduction	no
integrate	Integrate	Map/Gather	no
renderDepth	Rendering	Map	yes
renderTrack	Rendering	Map	yes
renderVolume	Rendering	Search/Stencil	no

Table 4.2 – Pencilizing First attempt, Manual : 68 FPS, PPCG : 0.1 FPS

```
void integrateKernel_core_summary( ... )
{
      for (int z = 0; z <= vol_size_z; ++z) {
            const float depthVal = depth[y][x];
            const short2 volVal = vol_data[z][y][x];
            vol_data[z][y][x] = volVal;
      }
      for (int i = 0; i < depthSize_y; i++)
      {
            for (int j = 0; j < depthSize_x; ++j)
            {
                  __pencil_use(depth[i][j]);
            }
      }
}

void integrateKernel_core( ... )
    __attribute__((pencil_access(integrateKernel_core_summary)));


#pragma scop
      {
            for (unsigned int y = 0; y < vol_size_y; y++) {
                  for (unsigned int x = 0; x < vol_size_x; x++) {
                        integrateKernel_core( ... );
                  }
            }

      }
#pragma endscop
```

Figure 4.4 – Integrate kernel with Summary function

functions to provide conservative affine accesses information. The pencilized $integrate$ kernel with summary function is shown in Figure 4.4. The summary function is a very powerful abstraction that can be used to hide complex data flow and expose only the outer relevant loops to PPCG. We can use affine loops in the summary functions which provides flexibility to express arbitrary array accesses to the polyhedral compiler. The summary functions also help reducing the complexity for the ILP solver inside the polyhedral auto scheduler. The entire summarized function is treated as a single statement with reads and writes as defined in the summary function. Hence, with the help of summary function PPCG was able to generate kernels for both $integrate$ and $track$. Table 4.3 shows the performance PPCG generated kernels. We were able to achieve around one FPS which is much less than the 68 FPS obtained with the manual SLAMBench kernels.

Kernels	Building Block	Pattern	PPCG kernel
mm2meters	Preprocess	Gather	yes
bilateralFilter	Preprocess	Stencil	yes
halfSample	Track	Stencil	yes
depth2vertex	Track	Map	yes
vertex2normal	Track	Stencil	yes
track	Track	Map/Gather	yes
reduce	Track	Reduction	no
integrate	Integrate	Map/Gather	yes
renderDepth	Rendering	Map	yes
renderTrack	Rendering	Map	yes
renderVolume	Rendering	Search/Stencil	yes

Table 4.3 – With Summary functions, Manual : 68 FPS, PPCG : 1 FPS

Kernels	Building Block	Pattern	PPCG kernel
mm2meters	Preprocess	Gather	yes
bilateralFilter	Preprocess	Stencil	yes
halfSample	Track	Stencil	yes
depth2vertex	Track	Map	yes
vertex2normal	Track	Stencil	yes
track	Track	Map/Gather	yes
reduce	Track	Reduction	yes
integrate	Integrate	Map/Gather	yes
renderDepth	Rendering	Map	yes
renderTrack	Rendering	Map	yes
renderVolume	Rendering	Search/Stencil	yes

Table 4.4 – With Parallel reductions, Manual : 68 FPS, PPCG : 50 FPS

4.4 Handling Reductions

After using summary functions PPCG was able to generate GPU kernels for all one $reduce$ function. The earlier version of PPCG was not able to parallelize reduction loops. It used to generate a kernel in which all the reductions were executed by single GPU thread. This was a become bottleneck for SLAMBench as the $reduce$ was in the critical path and is executed many time per single frame. Hence PPCG generated kernels has low performance of around 1 FPS compared to 68 FPS achieved with manual SLAMBench kernels. This was main motivation for handling reductions in chapter 3. We used the reduction built-ins as defined in chapter 3 to express the $reduce$ kernel. PPCG was able to generate efficient parallel reduction kernel for $reduce$ and performance improved to 50 FPS.

```
1   enum prl_mem_flags {
2     // defaults, not necessary to state explicitly, but may make code
        more readable what is meant.
3     prl_mem_readable_writable = 0,
4     prl_mem_readable = 0,
5     prl_mem_writable = 0,
6     prl_mem_host_readable = 0,
7     prl_mem_host_writable = 0,
8     prl_mem_dev_readable = 0,
9     prl_mem_dev_writable = 0,
10
11    // Take ownership; i.e. free resource on prl_mem_free
12    prl_mem_host_take = 1 << 0,
13    prl_mem_dev_take = 1 << 1,
14
15    // Whether the buffers are read/written in SCoPs (dev) or outside
        (host)
16    prl_mem_host_noread = 1 << 2,
17    prl_mem_host_nowrite = 1 << 3,
18    prl_mem_host_noaccess = prl_mem_host_noread |
        prl_mem_host_nowrite,
19    prl_mem_dev_noread = 1 << 4,
20    prl_mem_dev_nowrite = 1 << 5,
21    prl_mem_dev_noaccess = prl_mem_dev_noread | prl_mem_dev_nowrite,
22  };
```

Figure 4.5 – PRL memory flags

4.5 PRL runtime

We cannot include the entire SLAMBench into one single scop because of the presence of data dependent outer loop. This loop repeatedly calls track and reduce kernel until the achieved error is below a threshold value. Hence we only add pargma scops across individual kernels. Before the kernel execution the PPCG compiler generates data transfer calls from host to device memory for all the array accessed within a kernel. It also generates device to host transfer calls for all the arrays that are updated within device after the kernel execution. In SLAMBench we have sequences of separate kernels generated by PPCG with the data transfer call at the beginning and the end of each kernel. This results in many redundant data transfers across these kernel launches. These redundant data transfers have significant impact on the overall performance. In order to eliminate these we use PRL runtime.

The PRL runtime provides wrapper around memory allocation functions (malloc and free functions named `prl_mem_alloc` and `prl_mem_free`), data-transfer functions and kernel launches. The user can specify additional information through `prl_mem_flags` enum in `prl_mem_alloc`. The various options of `prl_mem_flags` is listed in Figure 4.5. These option enables programmer to specify data flow information in the non-pencil regions. The runtime uses these information to eliminate redundant data transfers. For e.g if an array is only used to store intermediate result

```
1   reductionoutput = (float*) prl_mem_get_host_mem(prl_mem_alloc(
      reductionoutput_size, prl_mem_host_nowrite));
2
3
4   floatDepth = (float*) prl_mem_get_host_mem(prl_mem_alloc(sizeof(
      float)* size, prl_mem_host_noaccess));
```

Figure 4.6 – PRL allocation calls

between two kernels, then we can use `prl_mem_host_nowrite` option in `prl_mem_alloc`. Now the runtime knows that this particular array is not modified by host outside of scops. All the device-to-host data transfer calls of this array will be no-ops. Thus eliminating all redundant data transfer. We also extended PRL to dump useful profiling and debugging information for the entire application. We are able to achieve 65 FPS which is very close to 66 FPS obtained by manually optimized SLAMBench kernel. The main advantage to using PENCIL for SLAMBench is it provides portable performance across different architectures. Once the kernels are written in PENCIL we can use PPCG to generate CUDA or OpenCL or OpenMP kernels automatically. Hence unlike manual written SLAMBench we don‘t have to maintain three different versions of the same kernels. Also we can easily tune the kernel parameters such as tile size, block size, grid size etc. for any given GPU architectures using autotuning.

5 Unified Model for Spatial Locality and Coalescing

Modern architectures feature deep memory hierarchies that may affect performance in both positive and negative ways. CPUs typically have multiple levels of cache memory that speed up repeated accesses to the same memory cells—*temporal locality*. Because loads into caches are performed with cache-line granularity, accesses to subsequent memory cells are also sped up—*spatial locality*. At the same time, parallel accesses to *adjacent* memory addresses may cause *false sharing*: caches are invalidated and data is re-read from more distant memory even if parallel threads access *different* addresses that belong to the same line. GPUs feature *memory coalescing* that group simultaneous accesses from parallel threads to adjacent locations into a single memory request in order to compensate for very long global memory access times. They also feature a small amount of fast shared memory into which the data may be copied in advance when memory coalescing is unattainable. Current polyhedral scheduling algorithms mostly account for the *temporal proximity* and leave out other aspects of the memory hierarchy.

We propose to manage all these aspects in a *unified* way by introducing new *spatial proximity* relations into the `isl` scheduler. They connect pairs of statement instances that access adjacent array elements. We treat spatial proximity relations as dependences for the sake of reuse distance computation. Unlike dependences, however, spatial proximity relations do not constrain the execution order and admit negative distances. We loosely refer to a spatial proximity relation as *carried* when the distance along it is not zero. If a schedule function carries a spatial proximity relation, it results in subsequent statement instances accessing subsequent array elements, and the distance along relation characterizes the access stride.

Spatial proximity relations can be used to set up two different ILP problems. The first problem, designed as a variant of the *Pluto* problem, attempts to carry as little spatial proximity as possible. The second problem, a variation of Feautrier's algorithm, carries as many spatial proximity relations as possible while discouraging skewed schedules. Choosing one or another problem to find a sequence of schedule functions allows `isl` to produce schedules accounting for memory effects. In particular, *false sharing* is minimized by carrying as little spatial proximity relations as possible in coincident dimensions. Spatial locality is leveraged by carrying as many spatial

proximity relations as possible in the last schedule function. This in turn requires previous dimensions to carry as little as possible. GPU memory coalescing is achieved by carrying as many spatial proximity as possible in the coincident schedule function that will get mapped to the block that features coalesced accesses. Additionally, this may decrease the number of arrays that will compete for the limited place in the shared memory as only those that feature non-coalesced accesses are considered.

5.1 Modeling Line-Based Access

The general feature of the memory hierarchies we model is that *groups* of subsequent memory cells rather than individual elements can be accessed. Although the number of array elements that form such groups varies depending on the target device and on the size of an element, it is possible to capture the general trend as follows. We modify the access relations to express that the statement instance accesses C subsequent elements. The constant C is used to chose the maximum stride for which spatial locality is considered, for example if $C = 4$, different instances of `A[5*i]` are not spatially related, and neither are statements accessing `A[i+5]` and `A[i+10]`.

Conventionally for polyhedral compilation, we assume not to have any information on the internal array structure, in particular whether a multidimensional array was allocated as a single block. Therefore, we can limit modifications to the last dimension of the access relation. Line-based access relations are defined as $\mathscr{A}' = \mathscr{A} \circ \mathscr{C}$ where $\mathscr{C} = \{\boldsymbol{a} \to \boldsymbol{a}' \mid a'_{1..(n-1)} = a_{1..(n-1)} \wedge a'_n = \lfloor \frac{a_n}{C} \rfloor\}$, and $n = \dim \vec{a} = \dim(\mathrm{Dom}\,\mathscr{A})$. This operation replaces the last array index with a virtual number that identifies groups of memory accesses that will be mapped to the same cache line. We use integer division with rounding to zero to compute the desired value. An individual memory reference may now accesses a set of array elements and multiple memory references that originally accessed distinct array elements may now access the same set.

The actual cache lines, dependent on the dynamic memory allocation, are not necessarily aligned with the ones we model statically. We use the over-approximative nature of the scheduler to mitigate this issue. Before constraining the space of schedule coefficients using Farkas' lemma, both our algorithms eliminate existentially-quantified variables necessary to express integer division. Combined with transitively-covered dependence elimination, it results in a relations between pairs of (adjacent in time) statement instances potentially accessing the same line. The over-approximation is that the line may start at *any* element and is *arbitrarily large*. While this can be encoded directly, our approach has two benefits. First, if C is chosen to be large enough, the division-based approach will cover strided accesses. For example, adding vectors of complex numbers represented in memory as a single array with imaginary and real part of a complex number placed immediately after each other.Second, it limits the distance at which *fusion* may be considered beneficial to exploit spatial locality between accesses to *disjoint* sets of array elements.

Out-of-bounds accesses are avoided by intersecting the ranges of the line-based access relations with sets of all elements of the same array $\mathrm{Im}\mathscr{A}'_{\mathtt{S}_i\to\mathtt{A}_j} \leftarrow \mathrm{Im}\mathscr{A}'_{\mathtt{S}_i\to\mathtt{A}_j} \cap \mathrm{Im}\bigcup_k \mathscr{A}_{\mathtt{S}_k\to\mathtt{A}_j}$.

Accesses to scalars, treated as zero-dimensional arrays, are excluded from line-based access relation transformation since we cannot know in advance their position in memory, or even whether they will remain in memory or will be promoted.

5.2 Spatial Proximity Relations

Computing Spatial Proximity Relations Given unions of line-based read and write access relations, we compute the *spatial proximity* relations using the exact dataflow-based procedure that eliminates transitively-covered dependences [Fea91]. That is, only statement instances adjacent in time in the original program are considered to depend on each other. Note that we also consider spatial Read-After-Read (RAR) "dependence" relations as they are an important source of spatial reuse. Thanks to the separation of validity, proximity and coincidence relations in the scheduling algorithm, this does not unnecessarily limit parallelism extraction (which is controlled by the coincidence relations and does not include RAR relations).

Access Pattern Separation Consider the code fragment in Figure 5.1. Statement `S1` features a spatial RAR relation on B characterized by $\mathscr{P}_{S1\to S1,B} = \{(i,j) \to (i',j') \mid (i' = i+1 \wedge \lfloor j'/C \rfloor = \lfloor j/C \rfloor) \vee (i' = i \wedge \lfloor j'/C \rfloor = \lfloor j/C \rfloor)\}$. In this case, the first disjunct connects two references to `B` that access different parts of the array. Therefore, spatial locality effects are unlikely to appear.

Statement `S2` features a spatial proximity relation on D: $\mathscr{P}_{S2\to S2,D} = \{(i,j,k) \to (i',j',k') \mid (i' = i \wedge \lfloor k'/C \rfloor = \lfloor j/C \rfloor) \vee (i' = i \wedge \lfloor j'/C \rfloor = \lfloor k/C \rfloor)\}$. Yet the spatial reuse only holds when $|k-j| \leq C$, a significantly smaller number of instances than the iteration domain. The schedule would need to handle this case separately, resulting in an inefficient branching control flow.

```
for (i = 1; i < 42; ++i)
  for (j = 0; j < 42; ++j) {
S1: A[i][j] += B[i][j] + B[i-1][j];
    for (k = 0; k < 42; ++k)
S2:   C[i][j] += D[i][k] * D[i][j];
  }
```

Figure 5.1 – Non-identical (`S1`) and non-uniform (`S2`) accesses to an array.

Both of these cases express group-spatial locality that is difficult to exploit in an affine schedule. Generalizing, the spatial locality between accesses with different access *pattern* is hard to exploit in an affine schedule. Two access relations are considered to have different patterns if there is at least one access function, excluding the last one, that differs between them. The last function is also considered but without the constant factor, that is `D[i][j]` has the same pattern as `D[i][j+2]`, but not as `D[i][j+N]`. Note that we only transform the access relations for the sake of dependence analysis, the actual array subscripts remain the same. The analysis itself is then performed for each group of relations with *identical patterns*.

Access Completion Consider now the statement `R` in Figure 2.1. There exists, among others, a spatial RAR relation between different instances of `R` induced by reuse on `B`:

$$\mathscr{P}_{R\to R,B} = \{(i,j,k) \to (i',j',k') \mid \\ ((i' = i \wedge j' = j+1 \wedge \lfloor j'/C \rfloor = \lfloor j/C \rfloor \wedge k' = k) \vee \\ (\exists \ell \in \mathbb{Z} : i' = i+1 \wedge j' = C\ell \wedge j = C\ell + C - 1 \wedge k' = k))\}.$$

While both disjuncts do express spatial reuse, the second one connects statement instances from different iterations of the outer loop, `t`. Similarly to the previous cases, spatial locality exists for a small number of statement instances, given that loop trip count is larger than C. In practice, an affine scheduler may generate a schedule with inner loop *skewed* by $(C-1)$ times the outer loop, resulting in inefficient control flow.

Pattern separation is useless in this case since the relation characterizes self-spatial locality, and `B[k][j]` is the only reference with the same pattern. However, we can prepend an access function `i` to simulate that different iterations of the loop `i` access disjoint parts of `B`.

Note that the array reference `B[k][j]` only uses two iterators out of three available. Collecting the coefficients of affine access functions as rows of matrix A, we observe that such problematic accesses do not have full column rank. Therefore, we *complete* this matrix by prepending *linearly independent* rows until it reaches full column rank. We proceed by computing the Hermite Normal Form $H = A \cdot Q$ where Q is $n \times n$ unimodular matrix and H is an $m \times n$ lower triangular matrix, i.e. $h_{ij} = 0$ for $j > i$. Any row-vector v with at least one non-zero element $v_k \neq 0, k > m$ is linearly independent from all rows of H. We pick $(n-m)$ standard unit vectors $\hat{e}_k = (0 \ldots 0, 1, 0, \ldots 0), m < k \leq n$ to complete the triangular matrix to an n-dimensional basis. Transforming the basis with unimodular Q preserves its completeness. In our example, it transforms `B[k][j]` into `B[i][k][j]`, so that different iterations of surrounding loops access different parts of the array. This transformation is only performed for defining *spatial proximity* relations without affecting the accesses themselves.

Combining *access pattern separation* and *access completion*, we are able to keep a reasonable subset of self-spatial and group-spatial relations that can be profitably exploited in an affine schedule. Additionally, this decreases the number of constraints the ILP solver needs to handle, which for our test cases helps to reduce the compilation time.

5.3 Temporal Proximity Relations

Temporal proximity relations are computed similarly to dependences, with the addition of RAR relations. Furthermore, we filter out non-uniform relations whose source and sink of belong to the same statement as we cannot find a profitable affine schedule for these.

5.4 Carrying as Few Spatial Proximity Relations as Possible

Our goal is to minimize the number of spatial proximity relations that are carried by the affine schedule resulting from the ILP. The distances along these relations should be made zero. Contrary to *coincidence* relations, some *may* be carried. Those are unlikely to yield profitable memory effects in subsequent schedule dimensions and should removed from further consideration. Contrary to *proximity* relations, small non-zero distances are seldom beneficial. Therefore, minimizing the *sum* of distance bounds or making it zero as exposed earlier is unsuitable for *spatial* proximity. We have to consider bounds for separate *groups* of spatial proximity relations, each of which may be carried independently of the others. These groups will be described in section 5.5 below. Attempting to force zero distances for the largest possible number of groups with relaxation on failure is combinatorically complex. Instead, we minimize the distances and only keep the relations for which the distance is zero. Intuitively, this removes the first group that must be carried if the previous groups are not. This encoding does not guarantee a *minimal number* of groups is carried. For example, $(0,0,1,1) \prec (0,1,0,0)$ so $(0,0,1,1)$ will be preferred by lexmin even though it carries more constraints. On the other hand, we can leverage the lexicographical order to prioritize certain groups over others by putting them early in the lexmin formulation.

Combining Temporal and Spatial Proximity Generally, we expect *temporal locality* to be more beneficial to performance than *spatial locality*. Therefore, we want to prioritize the former. This can be achieved by grouping temporal proximity relations in the ILP similarly to spatial proximity ones and placing the temporal proximity distance bound *immediately before* the spatial proximity distance bound. Thus lexmin will attempt to exploit temporal locality first. If it is impossible, it will further attempt to exploit spatial locality. Proximity relations carried by the current partial schedule are also removed iteratively. Note that they would have been removed anyway after the tilable band can no longer be extended. The new ILP minimization objective is

$$\text{lexmin} \sum_{i=1}^{n_p} (u_{1,i}^{T+} + u_{1,i}^{T-}), w_1^T, \sum_{i=1}^{n_p} (u_{1,i}^{S+} + u_{1,i}^{S-}) \ldots \sum_{i=1}^{n_p} (u_{n_g,i}^{T+} + u_{n_g,i}^{T-}), w_{n_g}^T, \sum_{i=1}^{n_p} (u_{n_g,i}^{S+} + u_{n_g,i}^{S-}), w_{n_g}^S, \ldots \tag{5.1}$$

where $u_{j,i}^T$ are coefficients of the parameters and w_j^T is the constant factor in the distance bound for the j^{th} group of proximity relations, $1 \leq j \leq n_g$, and $u_{j,i}^S$, w_j^S are their counterparts for temporal proximity relations. The remaining non-bound variables are similar to those of (2.3), namely the sum of schedule coefficients and parameters and individual coefficient values.

5.5 Grouping and Prioritizing Spatial Proximity Constraints

Grouping spatial proximity relations reduces the number of spatially-related variables in the ILP problem and thus the number of iterative removals. However, one must avoid grouping relations

when, at some minimization step, one of them must be carried while the other should not.

Initial Groups Consider the statement `R` in Figure 2.1. There exists a *spatial proximity* relation $R \rightarrow R$ carried by the loop `j` due to accesses to `C` and `B`, and another one carried by the loop `k` and due to the access to `A`. If these relations are grouped, their distance bound will be the same for choosing j or k as the new schedule function. This effectively prevents the scheduler from taking any reasonable decision and makes it choose dimensions in order of appearance, (i, j, k). Yet the schedule (i, k, j) improves spatial locality because *both* `C` and `B` will benefit from the last loop carrying the spatial proximity relation. This is also the case for multiple accesses to the same array, e.g., `C` is both read and written. Therefore, we initially introduce a group for each *array reference*.

After introducing per-reference bounds, we order groups in the lexmin formulation to prioritize carrying groups that are potentially less profitable in case of conflict. We want to avoid carrying groups that offer the most scheduling choices given the current partial schedule as well as those accesses that appear multiple times. This is achieved by lexicographically sorting them following the decreasing access *rank* and *multiplicity*, which are defined below. Descending order makes the lexmin objective carry groups with the smallest *rank* and *multiplicity* first.

Access Rank This sorting criterion is used to prioritize array references that, given the current partial schedule, have the most subscripts that the remaining schedule functions can affect. Conversely, if all subscripts correspond to already scheduled dimensions, the priority is the lowest. Each array reference is associated with an access relation $\mathscr{A} \subseteq (\vec{i} \rightarrow \vec{a})$. Its *rank* is calculated as the number of not yet fixed dimensions. In particular, given the current partial schedule $\mathscr{T} \subseteq (\vec{i} \rightarrow \vec{o})$, we compute the relation between schedule dimensions and access subscripts through composition $\mathscr{T}^{-1} \circ \mathscr{A} \subseteq (\vec{o} \rightarrow \vec{a})$. The number of equations in $\mathscr{T}^{-1} \circ \mathscr{A}$ corresponds to the number of fixed subscripts. Therefore the rank is computed as the difference between the number of subscripts $\dim \vec{a}$ and the number of equations in $\mathscr{T}^{-1} \circ \mathscr{A}$.

Access Multiplicity In cases of identical ranks, our model prioritizes repeated accesses to the same cell of the same array. Access *multiplicity* is defined as the number of access relations to the same array that have the same affine hull *after removing the constant term*. The multiplicity is computed across groups. For example, two references `A[i][j]` and `A[i][j+42]` both have *multiplicity* = 2. Read and write accesses using the same occurrence of the array in the code, caused by compound assignment operators, are considered as two distinct accesses.

Combining Groups The definition of *access multiplicity* naturally leads to the criterion for group combination: groups that contribute to each others' *multiplicity* are combined, and the *multiplicity* of the new group is the sum of those of each group.

5.6 ILP Problem to Carry Many Spatial Proximity Relations

Our goal is to find a schedule function that carries as many spatial proximity relations as possible with small (reuse) distance as this corresponds to spatial reuse. However, skewing often leads to loss of locality by introducing additional iterators in the array subscripts. The idea of Feautrier's scheduler is to carry as many dependences as possible in each schedule function, which is often achieved by skewing. We modify Feautrier's ILP to discourage skewing by swapping the first two objectives: first, minimize the sum of schedule coefficients thus discouraging skewing without avoiding it completely; second, minimize the number of *non-carried* dependence groups. Yet the minimal sum of schedule coefficients is zero and appears in case of a trivial (zero) schedule function. Therefore, we slightly modify the linear independence method of Section 2.2.3 to remain in effect even if "dimension slack" is available. This favors non-trivial schedule functions that may carry spatial proximity against a trivial one that never does. The minimization objective is

$$\text{lexmin} \sum_{i=1}^{\max\dim\mathscr{D}_S} \sum_{j=1}^{n_s} (c^-_{j,i} + c^+_{j,i}), \sum_{k=1}^{n_g} (1 - e_k), \sum_{i=1}^{n_p} \sum_{j=1}^{n_s} d_{j,i}, \ldots \tag{5.2}$$

where n_s is the number of statements, n_p is the number of parameters, e_k are defined similarly to Feautrier's LP problem for each of n_g groups of spatial proximity relations. Validity constraints must be respected, distances along coincidence relations are to be made zero if requested.

5.7 Scheduling for CPU Targets

On CPUs, spatial locality is likely to be exploited if the innermost loop accesses subsequent array elements. False sharing may be avoided if parallel loops do not access adjacent elements. Therefore, a good CPU schedule requires outer dimensions to carry as few spatial proximity relations as possible and the innermost dimension to carry as many as possible. Hence we minimize (5.1) for all dimensions. On the last dimension we apply (5.2).

Single Degree of Parallelism For CPU targets, `ppcg` exploits only one coarse-grained degree of parallelism with OpenMP pragmas. Therefore, we completely relax *coincidence* relations for each statement that already has one coincident dimension in its schedule, giving the scheduler more freedom to exploit spatial locality. Furthermore, the *clustering* mechanism now tolerates loss of parallelism as long as one coincident dimension is left.

Wavefront Parallelism Generally, we attempt to extract coarse-grained parallelism, i.e., render outer schedule dimensions coincident. When coincidence cannot be enforced in the outermost dimension of a band, we continue building the band without enforcing coincidence to exploit tilability. Instead, we leverage *wavefront* parallelism by skewing the outermost dimension by the innermost after the band is completed. Thus the outermost dimension carries all dependences

previously carried by the following one, which becomes parallel.

Unprofitable Inner Parallelism Marking inner loops as OpenMP parallel often results in inefficient execution due to barrier synchronization. Therefore, we relax *coincidence* relations when two or less dimensions remain, even if no coincident dimension was found. As a result, the scheduler will avoid exposing such inner parallelism and still benefit from improved spatial locality.

Carrying Dependences to Avoid Fusion The *band splitting* in the `isl` makes each dimension computed by Feautrier's algorithm belong to a separate band. Therefore, dependences and (spatial) proximity relations carried by this dimension are removed from further consideration. Without these dependences and proximity relations, some fusion is deemed unprofitable by the *clustering* heuristic. We leverage this side effect to control the increase of register pressure caused by excessive fusion. We define the following heuristic $h = \sum_{i,k\,:\,\mathrm{aff}\mathscr{A}_{S_i\to k}\mathrm{unique}} \dim(\mathrm{Dom}\,\mathscr{A}_{S_i\to k})$ where $\mathscr{A}_{S_i\to k}$ have unique affine hulls across the SCC: $\forall i, j, \forall k \neq l, \mathrm{aff}\mathscr{A}_{S_i\to k} \neq \mathrm{aff}\mathscr{A}_{S_j\to l}$. The uniqueness condition is required to consider repeated accesses to the same array, usually promoted to a register, with the same subscripts once. This heuristic is based on the assumption that each supplementary array access uses a register. It further penalizes deeply nested accesses by taking into account the input dimension of the access relation.

As we still prefer to exploit outer parallelism whenever possible, this heuristic is only applied when the scheduler fails to find an outer parallel dimension in a band. When the h value is large $h > h_{\mathrm{lim}}$, we use Feautrier's algorithm to compute the next schedule function. This may prevent *some* further fusion and thus decreases parallelism in the current dimension while exposing parallelism in the subsequent dimensions. Otherwise, we continue computing the band and rely on *wavefront* parallelism as explained above. The values of h_{lim} can be tuned to a particular system.

Parallelism/Locality Trade-off If a schedule dimension is coincident and carries spatial proximity relations, its optimal location within a band is not obvious: if placed outermost, it will provide coarser-grained parallelism, if placed innermost, it may exploit spatial locality. The current implementation prefers parallelism as it usually yields better performance gains. However, in case of tiling, both characteristics can be exploited: in the *tile* loop band, this dimension should be put outermost to exploit parallelism; in the *point* loop band, this dimension should be put innermost to exploit spatial locality. To leverage the additional scheduling freedom offered by stripmining/tiling, note that `ppcg` optionally performs *post-tile loop reordering* using the Pluto heuristic (2.2).

5.8 Scheduling for GPU Targets

High-end GPUs typically exploit three degrees of parallelism or more. Memory coalescing can be exploited along the parallel dimension mapped to the `x` threads. One should strive to coalesce as many accesses as possible; `ppcg` will try to copy arrays with uncoalesced accesses into the limited shared memory. Therefore, we first minimize (5.2) while enforcing zero distance along *coincidence* constraints. If no coincidence solution can be found, we apply Feautrier's scheduler for this dimension in an attempt to expose *multiple* levels of inner parallelism. If a coincident solution does not carry any *spatial proximity*, we discard it and minimize (2.3) instead. Because the band members must carry the same dependences and proximity relations, it does not make sense to keep looking for another dimension that carries spatial proximity if the first could not exploit spatial proximity: if spatial proximity could have been exploited, it would have already been found. It also does not make sense to keep looking for such dimension in the following band since only the outermost band with coincident dimensions is mapped to GPU blocks. Therefore, we relax *spatial proximity* constraints. They are also relaxed once one dimension that carries them is found as memory coalescing is applied along only one dimension. After relaxation, we continue with the regular `isl` scheduler applying (2.3) or Feautrier's ILP.

Mapping The outermost coincident dimension that carries spatial proximity relations in each band is mapped to the `x` block by `ppcg`. All other coincident dimensions, including the outermost if it does not carry spatial proximity, are mapped to blocks in inverse order, i.e., `z`, `y`, `x`.

5.9 Experimental Evaluation

The evaluation takes two parts. We first compare speedups obtained by our approach with those of other polyhedral schedulers; the following section highlights the differences in affine schedules produced with and without considering memory effects.

5.9.1 Implementation Details

Our proposed algorithm is implemented as an extension to `isl`. Dependence analysis and filtering is implemented as an extension to `ppcg`. Our modifications apply on top of the development versions of both tools (precise commits will be released and indicated in the published version) available from git://repo.or.cz/ppcg.git and git://repo.or.cz/isl.git.

Additional Modifications to the `isl` Scheduler Miscellaneous improvements were introduced to `isl` alongside the design and implementation of the new scheduler. Solving an integer LP inside Feautrier's scheduler instead of a rational LP if the latter gives rational solutions; this avoids large schedule coefficients. Using original loop iterators in the order of appearance in case of cost function ties; similarly to Pluto. In Pluto-style ILP, minimize the sum of coefficients for loop iterators $\vec{i}$ rather than for already computed schedule dimensions ϕ_j; for example, after computing $\phi_1 = i + j$ and $\phi_2 = j + k$ prefer $\phi_3 = \phi_1 - \phi_2 = i - k$ over $\phi'_3 = \phi_1 + \phi_2 = i + 2j + k$. For reproducibility and finer characterization of the scheduler, we compare both the stable and the development version of `isl` with our implementation in cases where they produce different schedules.

Iterative Removal of Proximity Relations As described in Subsection 2.2.3, `isl` uses an iterative approach to explore the schedule space with negative coefficients. To reduce the number of cases to consider, it iterates over subspaces more aggressively by only looking for a solution that is *significantly better* than the existing one, i.e., the one where the leading non-zero component of the new solution is less than the same component of the original solution. For example, if it found a solution $(0, 0, x, 1, 1, 1), x > 0$ in one subspace, it may ignore a lexicographically smaller solution $(0, 0, x' = x, 0, 0, 0)$ in another subspace as it constrains $x' < x$. This behavior may be detrimental when attempting to carry a small number of spatial proximity relations. Therefore, we have to relax the constraints that led to the first $x \neq 0$. Instead of solving multiple ILP problems from scratch, it is possible to leverage the incremental solver available in `isl`.

GPU Mapping We extended the schedule tree to communicate information about the ILP problem that produced each of the dimensions. If the first coincident schedule dimension was produced by carrying many spatial proximity relations, we map it to the `x` block. For the remaining dimensions, and if no spatial proximity was carried, we apply the regular `z,y,x` mapping order. Arrays required by GPU kernels and mapped to shared memory are copied in

row-major order without re-scheduling before the first and after the last kernel call. Further exploration of mapping algorithms and heuristics is of high interest but out of the scope of this paper.

5.9.2 Experimental Protocol

Systems We experimentally evaluated our unified model on different platforms by executing the transformed programs on both CPUs and GPUs—with the same input source code, demonstrating performance portability. Our testbed included the following systems:

- **ivy/kepler**: 4× Intel Xeon E5-2630v2 (Ivy Bridge, 6 cores, 15MB L3 cache), NVidia Quadro K4000 (Kepler, 768 CUDA cores) running CentOS Linux 7.2.1511. We used gcc 4.9 compiler with options `-O3 -march=native` for CPU, and nvcc 8.0.61 with option `-O3` for GPU.
- **skylake**, Intel Core i7-6600u running Ubuntu Linux 17.04. We used the GCC 6.3.0 compiler with `-O3 -march=native` options.
- **westmere**, 2× Intel Xeon X5660 (Westmere, 6 cores, 12MB L3 cache) running Red Hat Enterprise Linux Server release 6.5. We used icc 15.0.2 with option `-O3`.

Benchmarks We evaluate our tools on PolyBench/C 4.2.1, a benchmark suite representing computations in a wide range of application domains and commonly used to evaluate the quality of polyhedral optimizers. We removed a `typedef` from `nussinov` benchmark to enable GPU code generation by `ppcg`. Additionally, we introduced variants of `symm`, `deriche`, `doitgen` and `ludcmp` benchmarks, in which we manually performed scalar or array expansion to expose more parallelism. On CPUs, all benchmarks are executed with `LARGE` data sizes to represent more realistic workloads. On GPUs, we observed that, even with `EXTRALARGE` size, some benchmarks use too little data and run too fast to obtain stable performance measurements. On the other hand, different benchmarks (those that require skewing to express inner parallelism), did not terminate in 20 minutes with this size. Therefore, we used modified program sizes for GPUs reported in Figure 5.4, which are powers of two so as to simplify the generated code after tiling.

Tools Since the Pluto+ implementation cannot handle several of the Polybench 4.2.1 benchmarks, we compare against Pluto. Note that [BAC16] reports that Pluto+ and Pluto generate identical schedules for PolyBench, which is consistent with our observations.

The polyhedral compilers we compared are the following:

- `ppcg` *public*: latest `ppcg` release (`ppcg-0.07` with `isl-0.18`)

- `ppcg` *trunk*: see implementation details;
- `ppcg` *spatial*: chapter 5, with and without *post-tile* reordering;[1]
- *Pluto*: Pluto 0.11.4 with `-parallel -tile` options when appropriate;
- *PolyAST*: with reductions and DOACROSS parallelism support disabled.[2]

All versions of `ppcg` and Pluto were instructed to perform loop tiling with size 32 on CPUs and 16 on GPUs. Smaller sizes on GPUs help fit as many arrays as possible into the shared memory. Pluto is unable to generate CUDA code and was not used in GPU evaluations.

Measurements We collected execution times using the default PolyBench timing facility on CPU, and using the NVidia CUDA profiler on GPUs (we summed all kernel execution times reported by the profiler in cases where multiple kernels were generated).

For each condition, we performed all measurements 5 times and picked the *median* value.

5.9.3 Sequential Code Performance

Polyhedral optimizers can be used to improve the performance of *sequential* programs (with exploitable vector parallelism) on modern CPUs with deep memory hierarchies and advanced vectorization features. We measured run times of the benchmarks on the **skylake** system, which features the AVX2 instruction set. For Pluto, we used `-tile -intratileopt` flags. For baseline `ppcg`, we used `-target=c -tile` flags. For our variant of `ppcg`, we additionally used the `-isl-schedule-spatial-fusion` flag (consider spatial fusion relations in fusion heuristic). The speedup of the transformed code over the original code is shown in Figure 5.2(top).

Spatial locality-aware scheduling resulted in significant improvements for the two PPCG spatial versions relative to Pluto for `2mm, 3mm, gemver, mvt` and `symm` and benchmarks. For `2mm, 3mm`, the speedup grows from 2.9× to 4.6×. Pluto was unable to transform `symm` while our flow achieves 2.4× speedup, part of which is attributed to changes in `isl` alone. For several benchmarks, including `atax, deriche, jacobi-1d, ludcmp`, all variants of `ppcg` generate faster codes. This is due to (1) a different loop fusion structure thanks to the clustering technique and (2) live-range reordering support. Small differences in performance between Pluto and `ppcg`-spatial, like those observed in `covariance`, `correlation` or `trmm` are due to the differences in code generation algorithms between the tools: `ppcg` tends to generate simpler and thus faster control flow than CLooG, used in Pluto. For `gemm`, Pluto code is slightly more efficient because `ppcg` decided to fuse the initialization and the computation statements,

[1] Available at [redacted for double-blind review]

[2] Reduction is ignored at the parallelization phase and DOACROSS is converted into wavefront DOALL.

improving (temporal and spatial) locality but resulting in more complex control flow. Finally, Pluto versions outperform those of ppcg for adi, gesummv and gramschmidt. This is due to the difference in tiling strategies: contrary to ppcg, Pluto may tile imperfectly nested loops. This strategy is in practice equivalent to performing loop fusion after tiling, which would hinder ppcg's clustering approach as the entire schedule is considered to be fixed when loop tiling is performed. At the same time, we observed large variance of speedups between different runs of gesummv and gramschmidt, some runs of ppcg-spatial approaching Pluto results. Further experimentation on different systems and with different tile strategies and sizes is required to draw conclusions for these cases. Post-tile reordering in ppcg had only a marginal effect for the sequential code.

5.9.4 Parallel CPU Code Performance

While Section 5.3 showed robust performance for sequential execution, we would expect the impact of our unified approach to be even stronger when optimizing for both parallelization and memory locality. We measured run times of the benchmarks on the **ivy** system with 24 threads. For Pluto, we used `-parallel -tile -intratileopt` flags. For baseline ppcg, we used `-target=c -openmp -tile` flags. For our variant of ppcg, we used a set of flags that enables all heuristics described in this paper. The speedup of the transformed and parallelized code over the original sequential version is shown in Figure 5.2(middle).

Similarly to sequential versions, our approach results in significant speedup over pluto for 2mm and 3mm, growing from 6.8× to 14.4× and from 6.5× to 16.7×, respectively. Even without memory effects modeling, ppcg outperforms Pluto because of differences in the selection of loop fusion transformations. ppcg also outperforms Pluto in multiple other cases, including larger benchmarks correlation and covariance. Memory effects modeling corrects numerous cases in which standard ppcg was counterproductive. Furthermore, it is able to achieve up to 1.4× for stencil-like codes heat-3d and jacobi-1d where Pluto yields a 2× slowdown. This is due to a simpler schedule structure exposing inner parallelism *via* Feautrier's scheduler and our heuristic for register pressure reduction. Minor differences in performance, for example in the seidel-2d case, are again caused by differences in code generation whereas the schedules produced by Pluto and ppcg-spatial are identical. Live-range reordering enables ppcg to parallelize ludcmp and symm. In these cases, memory effects play only a small role in performance improvement. For example, the speedup for symm grows from 2.3 with ppcg-trunk to 2.7 with ppcg-spatial-posttile. The difference is more visible after scalar expansion (symm_ex): Pluto is able to parallelize this version and achieves 20× speedup while ppcg-spatial-posttile reaches 25.8× speedup. For the reasons discussed earlier, Pluto still significantly outperforms ppcg on gramschmidt (8.8× and 2.9× speedup, respectively) benchmark as well as on nussinov. As for sequential versions, this difference is caused by ppcg's inability to perform loop fusion after tiling.

Note that syntactic post-tile reordering transformation is not always beneficial when our algorithm

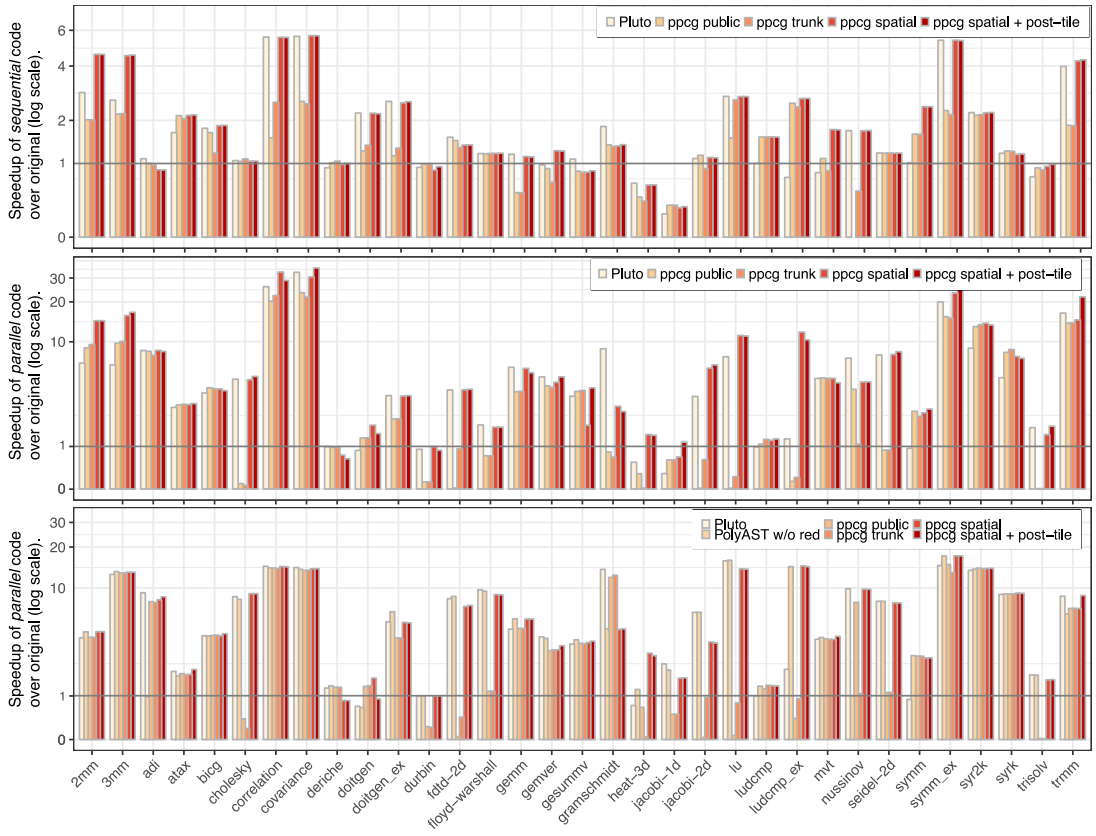

Figure 5.2 – Speedup of the optimized tiled code over the original code with different scheduling algorithms; top: sequential code on **skylake**, middle: parallel code on **ivy**; bottom: parallel code on **westmere**.

is used to exploit spatial locality. For example, it increases speedup for `covariance` from 30.5× to 32.4× and *decreases* it from 33× to 28.7× for `correlation`. Post-tile reordering is mainly beneficial when *different* schedules are required for tile loops and point loops.

5.9.5 Comparison with Hybrid Affine/Syntactic Approach

We compared the results our approach can achieve with those of PolyAST (disabling reductions and doacross supports), a state-of-the-art hybrid scheduling tool that uses an affine scheduler to improve locality and then relies on syntactic AST-based transformations to exploit parallelism. We also compared with Pluto-optimized codes. The speedups from parallel execution on **westmere** are shown in Figure 5.2(bottom). As PolyAST was designed to yield efficient code for `icc`, we used this compiler across all tools. Pluto was run with `-noprevector` flag to disable `icc`-specific pragmas it is able to generate contrary, since that feature is not supported by the other tools.

Overall, the observed performances for PolyAST and `ppcg` are very close. PolyAST could not

fully transform `adi` and `nussinov` into the polyhedral model, hence obtained no speedup. Both PolyAST and `ppcg`-spatial computed identical schedules for `2mm` and `3mm`, resulting in 4.5× and 13.2× speedups for the respective benchmarks. Similar schedules and hence close performance characteristics are observed for multiple other benchmarks, including `symm`, `trisolv` and `lu`. Such observations confirm our intuition that a unified polyhedral approach can obtain comparable schedules to a hybrid approach. Minor performance differences should be attributed to differences in code generation tools, which may enable and hinder different `icc` optimizations. For example, schedules for `floyd-warshall` are identical for PolyAST, Pluto and `ppcg`, yet they achieve 9.7×, 9.4× and 8.9× speedups, respectively. We did not tune the register pressure reduction heuristic to **westmere**, which resulted in performance difference on stencil-like benchmarks: on `heat-3d`, `ppcg` obtains 2.9× speedup while PolyAST reaches only 1.2×; on `jacobi-2d`, the situation is the inverse, `ppcg` obtains only 3.7× speedup while PolyAST reaches 6.5×. In both cases, our approach chose to create two inner parallel loops with simple schedules rather than a single parallel loop with more complex schedules due to skewing and shifting. It does so by applying Feautrier's scheduler for the outer dimension and splitting bands. Yet for the smaller `jacobi-2d` this is not profitable. Setting $h_{\text{lim}} = 32$ for this system would produce the same schedule as Pluto. The live-range reordering support in PPCG enables additional loop tiling for benchmarks including `doitgen` and `ludcmp`, and results in better performance than PolyAST and Pluto. Finally, for `atax` and `trmm` benchmarks, both Pluto and `ppcg`-spatial outperform PolyAST. Based on the decoupled optimization policy, PolyAST's affine scheduling phase focuses on improving data locality and consequently locates non-doall loops at the outermost for `atax` and `trmm`. In contrast, Pluto and `ppcg`-spatial enable outermost doall parallelism while enhancing per-tile data locality via the *post-tile* reordering.

5.9.6 Parallel GPU Code Performance

GPU performance was evaluated on the **kepler** system. We only compared different variants of `ppcg` as Pluto cannot produce GPU code and PolyAST-GPU relies on a drastically different code generation tool.

We selected six PolyBench benchmarks where the spatial effects scheduler had an impact, as presented in Figure 5.3. For all cases except `lu`, `ppcg` discovers no outer parallelism and resorts to repeated kernel calls from the generated host code. Figure 5.4 summarizes the number of different kernels and the cumulative number of kernel invocations. In such cases, an important benefit of our approach lies reducing the number of kernel calls, each of which introduces overhead, as well as optimization of the kernel itself. The spatial version of `ppcg` reduced the number of kernels for `adi` and `lu` due to different scheduling decisions and spatial effects-aware fusion. As a result, speedup for `lu` grows from 4.6× to 19.1×. For `adi`, it slightly decreases from 0.74× to 0.7×, but this kernel seems unsuitable for GPU processing anyway. For `gramschmidt` and `trisolv`, our algorithm manages to reduce the number of kernel invocations. Note that the kernel execution time for `trisolv` is marginal in the total execution time, therefore mapping this kernel alone to GPU is counterproductive. However, the impact of our optimization could

be increased if `trisolv` was part of a larger application that was mapped on to the GPU. For `symm`, our algorithm took a better decision for memory coalescing, resulting in small additional speedup. Finally, for `seidel-2d`, both spatial effects and trunk `ppcg` show slowdowns relative to the public version. A detailed analysis shows that this was because the code generation in public `ppcg` happened to interchange the two innermost loop, whereas the trunk and spatial versions always strive to preserve the original loop order for the innermost loops. Thus, the superior performance of public ppcg was accidental, and not a result of a scheduling decision. Correction of this regression requires a scheduling algorithm that can jointly optimize for the different memory spaces in the CPU and GPU, which is an excellent candidate for future research.

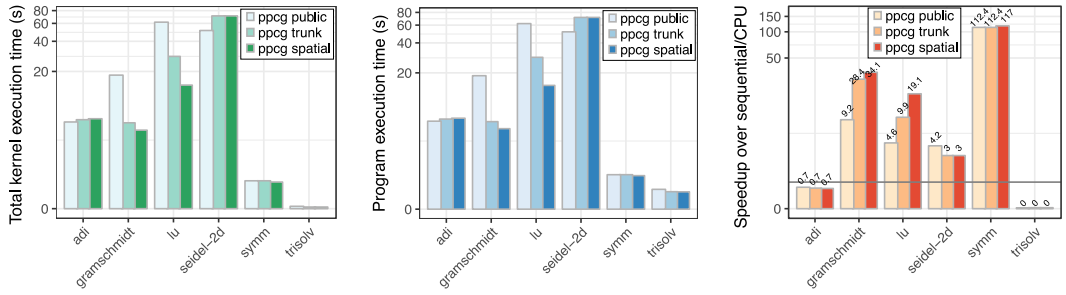

Figure 5.3 – Left and center graphs show total kernel execution time and program execution time (lower is better). Right graph shows speedup over CPU sequential version (higher is better).

	adi	gramschmidt	lu	seidel-2d	symm	trisolv
parameter value	512	2048	4096	1024 × 4096	2048	4096
# kernels (public)	14	7	3	1	2	3
# invocations (public)	7168	28643	20471	16372	2	12286
# kernels (spatial)	6	7	2	1	2	3
# invocations (spatial)	3072	12287	8190	16372	2	8192

Figure 5.4 – Parameter values, number of kernels generated by public and spatial versions of `ppcg` and cumulative number of invocations of those kernels for each benchmark (lower is better).

Beyond these 6 cases, spatial locality modeling did not affect the generated schedule since `ppcg` prioritizes parallelism over any sort of locality effects for GPU targets. On the other hand, a mechanism to avoid large schedule coefficients in Feautrier's scheduler was introduced in `isl`-trunk and significantly improved performance on `deriche`, `doitgen`, `cholesky` and `ludcmp` (most spectacular on `deriche`'s 3256× speedup). However, compared to the sequential CPU code, it results in a modest 1.25× improvement.

Overall, our approach can indeed exploit additional memory coalescing. As Polybench includes only a small number of benchmarks with sufficient amount of parallelism for GPU mapping, spatial effects-related changes the schedule only in a small number of cases. In other cases, it prefers parallelism as the main source of performance. Evaluating on larger benchmarks with longer execution time would be necessary to fully estimate the benefits of our model on GPUs.

5.10 Differences in Schedules: Case Study Discussions

5.10.1 Two Matrix Multiplications

`2mm` is a linear algebra kernel that computes a $I \times L$ matrix $D = \beta C \cdot \alpha(A \cdot B)$ where A is a $I \times K$, B is a $K \times J$, and C is a $J \times L$ matrix; α, β are scalars as shown in Fig 5.5(left).

The schedule computed by our algorithm for the OpenMP target is $\{\texttt{S1}(i,j) \rightarrow (0,i,0,j)\} \cup \{\texttt{S2}(i,j,k) \rightarrow (0,i,k,j)\} \cup \{\texttt{S3}(i,j) \rightarrow (1,i,0,j)\} \cup \{\texttt{S4}(i,j,k) \rightarrow (1,i,k,j)\}$ whereas Pluto proposes $\{\texttt{S1}(i,j) \rightarrow (0,i,j,1,0)\} \cup \{\texttt{S2}(i,j,k) \rightarrow (1,i,j,0,k)\} \cup \{\texttt{S3}(i,j) \rightarrow (0,i,j,0,0)\} \cup \{\texttt{S4}(i,j,k) \rightarrow (1,i,k,1,j)\}$, `ppcg`-trunk proposes $\{\texttt{S1}(i,j) \rightarrow (0,i,j,0)\} \cup \{\texttt{S2}(i,j,k) \rightarrow (0,i,j,k)\} \cup \{\texttt{S3}(i,j) \rightarrow (1,i,j,0)\} \cup \{\texttt{S4}(i,j,k) \rightarrow (1,i,j,k)$ and PolyAST proposes $\{\texttt{S1}(i,j) \rightarrow (i,0,j)\} \cup \{\texttt{S2}(i,j,k) \rightarrow (i,1,k,j)\} \cup \{\texttt{S3}(i,j) \rightarrow (i,2,j)\} \cup \{\texttt{S4}(i,j,k) \rightarrow (i,3,k,j)$.

Pluto exploits locality between `tmp[i][j]` in `S2` and `tmp[i][k]` in `S4` by fusing the surrounding loops. By doing so, it loses the possibility to improve spatial locality on `B[k][j]` in `S2`. It also introduces extra control flow due to different schedules in the three last dimensions. Finally, it loses proximity between `S1` and `S2` and between `S3` and `S4`. `ppcg` essentially preserves the original code structure because two outer dimensions are parallel, which is unnecessary for CPUs. PolyAST first computes the most profitable loop order for each statement by the DL memory cost model [Sar97] and finds schedules based on the profitable orders, e.g., `i-k-j` is chosen for `S2` and `S4` as with our approach; and all `i` loops are fused to improve locality.

Our approach maintains the original fusion structure, trading off locality between "initialization" and "computation" statements for locality between `S2` and `S4`. The reasoning inside each of the two new loop nests is identical, so we only consider the first one. Our algorithm replaces temporal locality on `tmp[i][j]` with spatial locality, spatial locality on `A[i][k]` with temporal locality and additionally leverages spatial locality on `B[k][j]`. The first dimension is chosen as `i` because access ranking prioritizes `tmp[i][j]` due to write access. In this reference, `i` carries neither proximity nor spatial proximity. For the second dimension, access ranking prioritizes `B[k][j]` because it uses two yet unscheduled iterators. Thus the scheduler chooses `k` as it does not carry spatial proximity, `i` being linearly dependent on the previous dimension. The remaining dimension is chosen as `j` because it carries spatial proximity on both `tmp[i][j]` and `B[k][j]`.

On GPUs, both `ppcg`-trunk and `ppcg`-spatial produced the same schedule with respect to mapping: `i` and `j` loops are outer parallel and are mapped to `y` and `x` blocks, respectively, original fusion structure is maintained. However, without spatial effects modeling, `ppcg` cost function cannot distinguish between `i` and `j` loops. It maintains their original order, which is profitable in this particular case. Had these loops been nested in the opposite order, the public `ppcg` would have mapped `j` to `y` and `i` to `x`, failing to exploit memory coalescing. Such schedule results in 1.5× slowdown in kernel execution time and 1.3× slowdown overall, compared to the profitable schedule for our test sizes. Our spatial effects-aware approach, on the other hand,

would have still computed the same profitable schedule. More detailed evaluation is required to fully demonstrate the stability of our algorithm to isomorphic loop permutations in the input. However, we expect this behavior to appear across multiple benchmarks.

5.10.2 LU Decomposition

LU decomposition is a linear algebra kernel that, given an $N \times N$ matrix A computes lower and upper triangular matrices L and U such that $L \cdot U = A$. It may be implemented in-place as shown in Figure 5.5(right), which is challenging for analysis due to a large number of non-uniform dependences.

```
void 2mm(double alpha, double beta,
   double A[NI][NK], double B[NK][NJ],
   double C[NJ][NL], double D[NI][NL]) {
 double tmp[NI][NJ];
 for (i = 0; i < NI; i++)
   for (j = 0; j < NJ; j++) {
S1:  tmp[i][j] = 0.0;
     for (k = 0; k < NK; ++k)
S2:    tmp[i][j] += alpha * A[i][k] * B[k][j];
   }
 for (i = 0; i < NI; i++)
   for (j = 0; j < NL; j++) {
S3:  D[i][j] *= beta;
     for (k = 0; k < NJ; ++k)
S4:    D[i][j] += tmp[i][k] * C[k][j];
   }
}
```

```
void lu(double A[N][N]) {
 for (i = 0; i < N; i++) {
   for (j = 0; j < i; j++) {
     for (k = 0; k < j; k++)
S1:    A[i][j] -= A[i][k] * A[k][j];
S2:  A[i][j] /= A[j][j];
   }
   for (j = i; j < N; j++)
     for (k = 0; k < i; k++)
S3:    A[i][j] -= A[i][k] * A[k][j];
 }
}
```

Figure 5.5 – Code of the `2mm` (left) and `lu` (right) benchmarks with labeled statements.

Both Pluto and our algorithm resort to *wavefront* parallelism after tiling. For the OpenMP version, Pluto proposes the schedule $\{\texttt{S1}(i,j,k) \rightarrow (i,j,k)\} \cup \{\texttt{S2}(i,j) \rightarrow (i,j,j)\} \cup \{\texttt{S3}(i,j,k) \rightarrow (i,j,k)\}$ that essentially embeds `S2` in the innermost loop so as to respect dependences. After tiling, it interchanges two innermost *point* loops to leverage spatial locality but keeps tile loop order unchanged. Our algorithm computes directly the schedule $\{\texttt{S1}(i,j,k) \rightarrow (i,k,j)\} \cup \{\texttt{S2}(i,j) \rightarrow (i,j,j)\} \cup \{\texttt{S3}(i,j,k) \rightarrow (i,k,j)\}$ including this interchange. Using this schedule for *both tile* and *point* loops improves sequential performance thanks to avoiding false sharing effect. The public `ppcg` does not have support for wavefront parallelism and does not parallelize this kernel. PolyAST proposes the schedule identical to ours because the DL model selected these loop orders as the most profitable in terms of memory cost reduction. The data locality modeling in our affine scheduler enables the same loop orders as the mixed affine/syntactic approach, while our unified scheduling approach be in no danger of missing doall parallelism discussed in Section 5.9.5.

For GPU targets, `ppcg`-trunk proposes the schedule $\{\texttt{S1}(i,j,k) \rightarrow (k,0,i,j)\} \cup \{\texttt{S2}(i,j) \rightarrow (j,1,i,j)\} \cup \{\texttt{S3}(i,j,k) \rightarrow (k,2,i,j)\}$ where j gets mapped to the x block and accesses `A[*][j]` feature memory coalescing. Note that `ppcg`-trunk respects the original order of loops and does not explicitly optimize for coalescing. Our algorithm produces the schedule $\{\texttt{S1}(i,j,k) \rightarrow (k,1,j,i)\} \cup \{\texttt{S2}(i,j) \rightarrow (j,0,j,i)\} \cup \{\texttt{S3}(i,j,k) \rightarrow (k,1,j,i)\}$ where j is also mapped to the x block

because it is known to feature coalescable accesses. Furthermore, our algorithm fuses two loops resulting in fewer kernels reducing kernel launch overhead and thus decreasing the total computation time.

5.10.3 Triangular Matrix Multiplication

Triangular matrix multiplication is a linear algebra kernel that takes $N \times N$ lower triangular matrix A and square matrix B; and computes a $N \times N$ matrix $Bout = A \cdot B$. Note that the output $Bout$ is to be stored in place of the input array B as shown in Fig. 5.5(right).

Both PPCG and Pluto select outermost doall parallelism for `S1` and `S2`, and proposes the schedule $\{\texttt{S1}(i,j,k) \rightarrow (0,j,i,k)\} \cup \{\texttt{S2}(i,j) \rightarrow (1,i,j)\}$ while schedule by PolyAST is $\{\texttt{S1}(i,j,k) \rightarrow (0,k,i,j)\} \cup \{\texttt{S2}(i,j) \rightarrow (1,i,j)\}$ where the outermost `k` loop of `S1` carries loop dependence and thereby requires doacross parallelization.

Based on the decoupled optimization policy, PolyAST's affine scheduling phase focuses on improving data locality and selects `k-i-j` loop order for `S1`; and the parallelization phase in the latter syntactic stage detects the doacross parallelism located outermost. On the other hand, our unified modeling of locality and parallelism enables outermost doall and post-tile permutation locates `j` loop at the innermost position for better locality within a tile. Our approach outperformed PolyAST by the factor of 1.4× improvement as reported in Section 5.9.

6 Auto scheduling Deep learning pipelines

In recent years there has been an exponential growth in deep learning techniques. The key factor for the success of these deep learning has been large amount of data and the availability of enormous compute power. Due to which we can build an highly accurate model by training them on millions of images. The basic units of deep learning pipelines is combination of affine operator typically convolution and pointwise non-linear operator typically Relu. The typical deep learning pipeline consists of multiple layers of such combination stacked together. During the training phase, the weights of convolution and fully connected layers are learnt from the training data. Initially these weights are randomly chosen. Each images in the training set propagated through the network to produce the output, this is output is compared against the actual output, the difference between the two is back propagated. During back propagation, the weights in the each layers are adjusted according to the gradient dissent. This training process is repeated for millions of images resulting in highly accurate model. Once the model is built, it can evaluated for any new input by propagating through the model, this is refereed to as inference. Deep learning pipeline has all the computations that are affine and are amendable for polyhedral compilation. In this chapter We study advantages and challenges of using polyhedral compilation techniques for deep learning tensor computations. Figure 6.1 shows the overview of the system. We extract polyhedral intermediate representation from TVM [CMJ+18], a end-to-end compiler stack that has support for many popular deep learning frameworks such as Tensorflow, MXNet, Keras etc.. TVM follows the same principle as Halide [RKBA+13] of separating computation and scheduling specification. We use polyhedral scheduling with autotuning to automatically compute the TVM schedule.

6.1 TVM

TVM [CMJ+18] is an end-to-end compiler stack that is designed to optimize and fine-tune deep learning workloads for a wide variety of hardware platforms. It is mainly composed of two intermediate layers: a computational DAG (directed acyclic graph) and low-level tensor description language. The graph IR is similar to TensorFlow XLA [Goo] where nodes represent

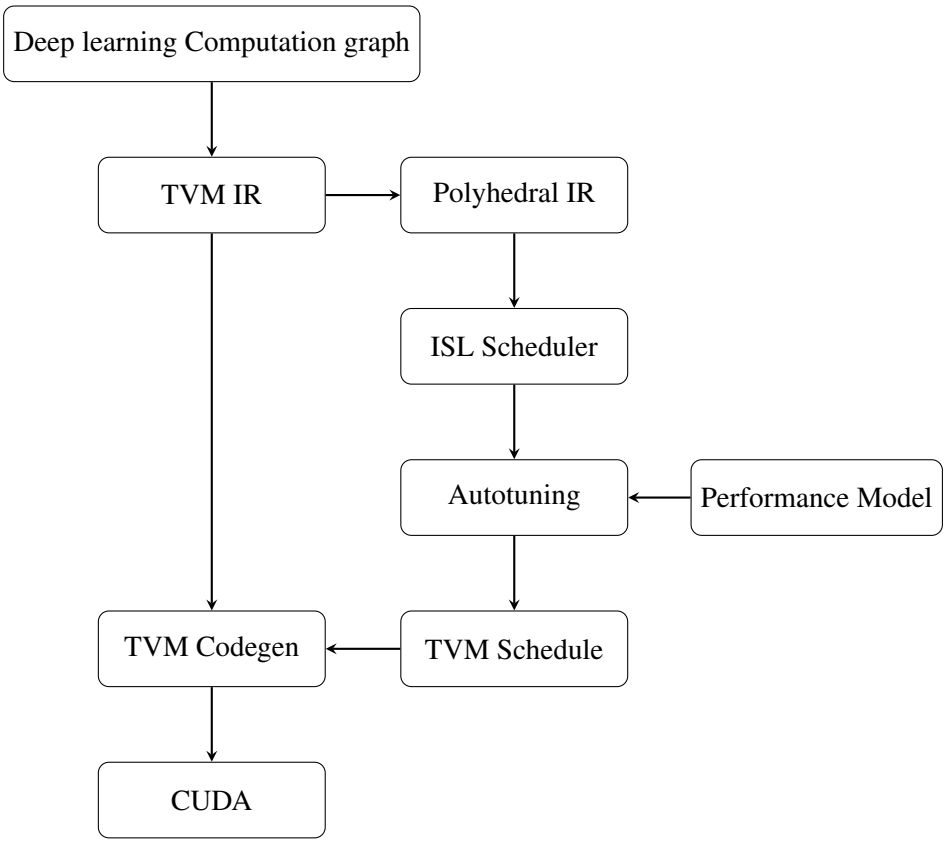

Figure 6.1 – TVM auto scheduling overview

basic tensor operations like convolution and edges represent the data flow dependency. The tensor description language is inspired by Halide [RKBA+13] and follows the same principle of decoupling computation and scheduling descriptions. The Figure 6.1 shows the computation part of TVM IR for the matrix multiply kernel. The input tensors with its dimensions are explicitly specified and the tensor operations are specified as a computation on input tensor index expressions in compute statement. The reduction variables must be explicitly declared using reduce-axis and the rest of index variables represent parallel loop dimensions.

```
import tvm
n, m, l = tvm.var('n'), tvm.var('m'), tvm.var('l')
A = tvm.placeholder((n, l), name='A')
B = tvm.placeholder((m, l), name='B')
k = tvm.reduce_axis((0, l), name='k')
C = tvm.compute(
  (n, m),
  lambda ii, jj: tvm.sum(A[ii, k] * B[jj, k], axis=k),
  name='CC')
```

6.1.1 TVM schedule primitives

TVM scheduling primitives provide fine grain control for the programmer to express various program transformations needed to obtain high-performance implementations of the tensor computations. The schedule primitives such as `split, reorder, tile, fuse, unroll` are used to perform high-level loop transformations. The `bind` primitive is used to map a loop to a GPU block or thread. The `cache_read` and `cache_write` primitives are used for shared memory and private memory promotion. The loop can be vectorized using `vectorize` primitive. The `double_buffer` primitive can be used to hide the latency of the global memory reads by overlapping computation and communication. A complete description of all the scheduling primitives can be found in [CMJ+18, tvm].

Figure 6.1.1 shows the schedule primitives used for optimizing GEMM kernel to GPUs. The `tvm.create_schedule` function is used to create schedule object for a given tensor operation (C.op in the Figure 6.1.1). We can access the individual loops of the tensor operation using `op.axis`. The reduction loops are explicitly specified in TVM IR that are separated from the normal loops and can be accesses using `op.reduce_axis`. The schedule primitives operate on these loops and can produce new loops. The `split` primitive is used to split a loop into two parts either by specifying `block_factor` or `num_parts`. The newly created inner and outer loop are returned by the `split` primitive. The `reorder` primitive is used to permute the loops in the order specified by the function arguments. We can perform tiling on a band of loops first by splitting individual loops and then reordering tile and point loops. There is also `tile` primitive which does the exact same operation. Multiple consecutive loops can be fused into a single loop using `fuse` primitive.

The individual loops can be mapped into different GPU block/thread axis using the `bind` primitive. We first need to create a GPU mapping axis using `tvm.thread_axis` primitive by passing axis name ("blockIdx.x" or "threadIdx.x") and an optional grid/block size. If a loops is mapped to a `thread_axis` of size `B` then the i iteration of loop is mapped to thread i/B. If block/grid size is not specified then grid/block size will be equal to the trip count of the loop being mapped. While this group mapping is enhances the cache locality, in some cases this may lead to uncoalesced accesses and shared memory bank conflicts. In order to achieve consecutivity TVM introduces another virtual thread axis called `vthread`. With the `vthread` mapping an iteration i of a loop is mapped to thread $i\%B$. In order to exploit both cache locality and consecutivity, an axis is split twice and mapped to blocks, vthread and threads starting from outer loops.

GPUs have limited global memory bandwidth hence most of the unoptimized tensor applications are bandwidth bound. A a very common GPU optimization to overcome this is to cache the global memory reads in the shared memory. If there is a data reuse access multiple threads of the block then caching global memory reads/writes in shared memory will decrease number of global memory requests. Shared memory promotion will also help in case of uncoalesced accesses. The same optimization can be applied to decrease the traffic between shared memory and registers by caching shared memory reads/writes in registers. The `cache_read` and `cache_write` primitives can be used to cache a read/write of a tensor. We can use this primitive to do shared or private memory promotion of a tensors. The output of `cache_read` or `cache_write` is a tensor computation `op_node` which can be scheduled like normal tensor operations. It has same number of axis as the dimensions of tensor being cached. These axis are used to map the reads/write loops to GPU thread hierarchy using `split` and `bind` primitives. We can change the root of caching loops using `compute_at` primitive which moves the root of tensor computation to a specified position. For shared memory promotion the copying loops can additionally vectorized (four consecutive memory loads with single instruction) using `vectorize` primitive. There is also primitive to enable overlapping global memory loads with the computation using `double_buffer`. In the Figure 6.1.1 for the GEMM kernel there is data reuse for reads of both A and B tensors hence they are promoted to shared memory, and the coping loops are vectorized. Furthermore these reads are cached in private memory to exploit register reuse. The write to tensor C is also cached at the register level to avoid multiple writes of intermediate values to the global memory.

The schedule primitives of TVM gives fine grain control to the user at individual loop level to perform various GPU optimizations need to achieve good performance. Once the schedule is specified we can generate code for many different targets. TVM supports multiple backends including cuda, opencl, x86, metal, ROCm etc..

```
1 s = tvm.create_schedule(C.op)
2 i,j = C.op.axis
3 k = C.op.reduce_axis
4
5 scale = 8
6 num_thread = 8
```

```
block_factor = scale * num_thread

by, yi = s[C].split(i, factor=block_factor)
bx, xi = s[C].split(j, factor=block_factor)
s[C].reorder(by, bx, yi, xi)

block_x = tvm.thread_axis("blockIdx.x")
thread_x = tvm.thread_axis((0, num_thread), "threadIdx.x")
block_y = tvm.thread_axis("blockIdx.y")
thread_y = tvm.thread_axis((0, num_thread), "threadIdx.y")
thread_xz = tvm.thread_axis((0, 2), "vthread", name="vx")
thread_yz = tvm.thread_axis((0, 2), "vthread", name="vy")

s[C].bind(by, block_y)
s[C].bind(bx, block_x)

AA = s.cache_read(A, "shared", [C])
BB = s.cache_read(B, "shared", [C])
AL = s.cache_read(AA, "local", [C])
BL = s.cache_read(BB, "local", [C])
CC = s.cache_write(C, "local")


tyz, yi = s[C].split(yi, nparts=2)
ty, yi = s[C].split(yi, nparts=num_thread)
txz, xi = s[C].split(xi, nparts=2)
tx, xi = s[C].split(xi, nparts=num_thread)
s[C].bind(tyz, thread_yz)
s[C].bind(txz, thread_xz)
s[C].bind(ty, thread_y)
s[C].bind(tx, thread_x)
s[C].reorder(tyz, txz, ty, tx, yi, xi)
s[CC].compute_at(s[C], tx)

yo, xo = CC.op.axis
ko, ki = s[CC].split(k, factor=8)
kt, ki = s[CC].split(ki, factor=1)
s[CC].reorder(ko, kt, ki, yo, xo)
s[AA].compute_at(s[CC], ko)
s[BB].compute_at(s[CC], ko)
s[AL].compute_at(s[CC], kt)
s[BL].compute_at(s[CC], kt)
# Schedule for A's shared memory load
ty, xi = s[AA].split(s[AA].op.axis[0], nparts=num_thread)
_, xi = s[AA].split(s[AA].op.axis[1], factor=num_thread * 4)
tx, xi = s[AA].split(xi, nparts=num_thread)
s[AA].bind(ty, thread_y)
```

```
s[AA].bind(tx, thread_x)
s[AA].vectorize(xi)
# Schedule for B' shared memory load
ty, xi = s[BB].split(s[BB].op.axis[0], nparts=num_thread)
_, xi = s[BB].split(s[BB].op.axis[1], factor=num_thread * 4)
tx, xi = s[BB].split(xi, nparts=num_thread)
s[BB].bind(ty, thread_y)
s[BB].bind(tx, thread_x)
s[BB].vectorize(xi)
s[AA].double_buffer()
s[BB].double_buffer()
```

6.2 Problem specialization

6.2.1 Convolutions

The primary computation in the deep neural network pipelines is performing the high-dimensional convolutions as shown in Figure 6.2. A typical layer applies localized linear affine transformation in form of convolutions followed by pointwise non linear function called as activation functions such as the rectified linear unit (ReLU). In this computation, the input activations of a layers are structured as a set of 2-D input feature maps, each of which is called a channel. Each channel is convolved with a distinct 2-D filter form the stack of filets, one for each channel. This stack of 2-D filter form single 3-D weights of a layer. The results of the convolution at each point is summed across all the channels to produce a single output activations that comprise one channel of output feature map. Additional 3-D filters are used on the same input to create many output channels. In addition, 1-D bias may be added to filtering results. Finally, multiple input feature maps may be processed together as a batch to potentially improve filter weight reuse.

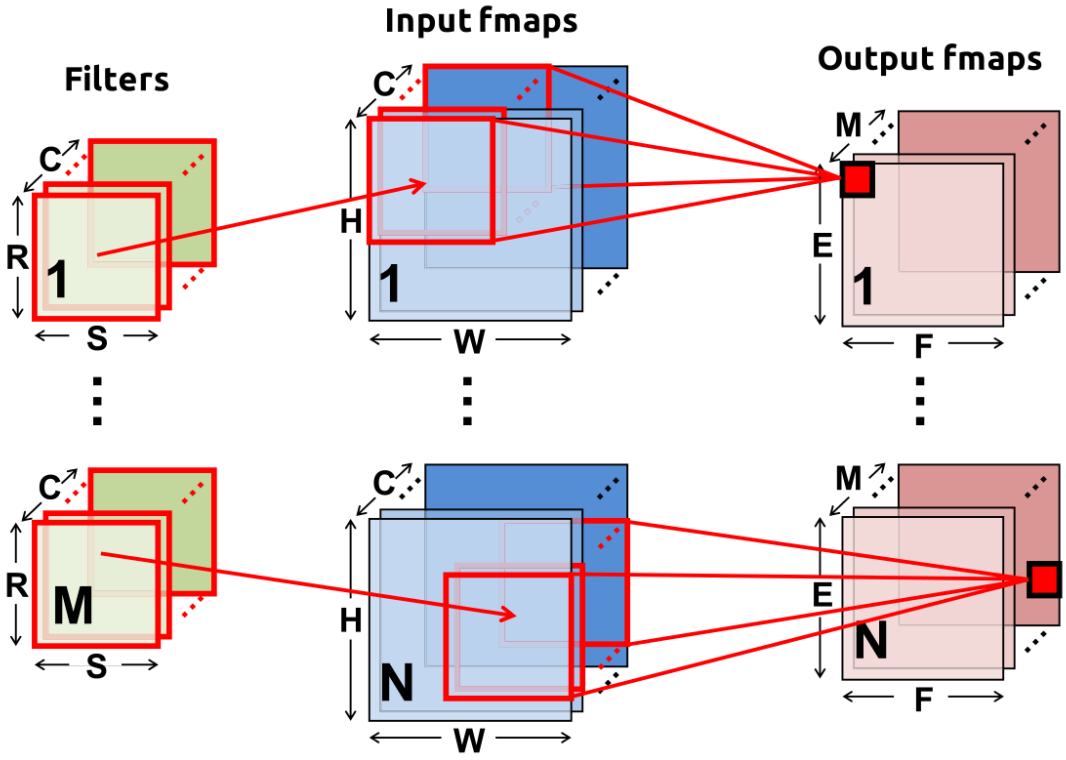

Figure 6.2 – Convolution computation

$$O[z][u][x][y] = ReLU(B[u] + \sum_{k=0}^{C-1} \sum_{i=0}^{R-1} \sum_{j=0}^{S-1} I[z][k][Ux+i][Uy+j] \times W[u][k][i][j]) \\ 0 \leq z < N, 0 \leq u < M, 0 \leq y < E, 0 \leq x < F \\ E = (H - R + U)/U, F = (W - S + U)/U \tag{6.1}$$

6.2.2 Data reuse in Convolutions

There are three different forms of data reuse for the convolution computation.

1. Filter Weight Reuse within Convolution: Each filter weight is reused $E \times F$ times in the same input plane: Each input pixel is reused $R \times S$ times in the same filter plane.
2. Input Feature Reuse within Convolution: Each input pixel is reused $R \times S$ times in the same filter plane.
3. Filter Reuse across Inputs: Each filter weight is reused across the batch of N inputs.
4. Input Reuse across Filters: Each of the input pixel is reused across M filters to generate M output channels.

Each of the input data reuse is captured by RAR dependence in polyhedral model. We want to find the transformation and mapping that maximizes data reuse and minimize data movement across different threads. In convolutions the amount of data reused across different dependences vary significantly with the value of parameters. For e.g if the batch size is high data reuse from type (3) is significant compared to other types. For the layers with larger $H \times W$ than input channels C, which is common for first few layers, the reuse with convolution (type 1 and 2) dominates. For the layers with larger C than input image size $H \times W$, which is case for last few layers in typical deep learning pipelines, reuse across inputs and filters (of type 3 and 4) are significant

The cost function used in the Pluto is shown to be an effective heuristic that finds the best transformation for most of the benchmarks. However the heuristic is agnostic to parameter values i.e. it finds the same transformation for different problem sizes. The optimal transformations for a given kernel will vary for different problem sizes. There is need to adapt these transformations with the problem size especially if they vary dramatically. If we know the value of parameters at compile time then, it is possible to find better transformations than the ones found by the current Pluto algorithm by finding specialized schedules for a given problem size. This is especially true for deep learning pipelines where we know the values of different convolutions are known a priori and they vary drastically across different layers. Table 6.1 shows different problem sizes of all the different layers of popular Resnet network. The sizes vary drastically from first layer to the last. Typically the initial few layers operate have larger height and width (224 x 224 for layer

layer	height	width	in_filter	out_filter	hkernel	wkernel	hpad	wpad	hstride	wstride
0	224	224	3	64	7	7	3	3	2	2
1	56	56	64	64	3	3	1	1	1	1
2	56	56	64	64	1	1	0	0	1	1
3	56	56	64	128	3	3	1	1	2	2
4	56	56	64	128	1	1	0	0	2	2
5	28	28	128	128	3	3	1	1	1	1
6	28	28	128	256	3	3	1	1	2	2
7	28	28	128	256	1	1	0	0	2	2
8	14	14	256	256	3	3	1	1	1	1
9	14	14	256	512	3	3	1	1	2	2
10	14	14	256	512	1	1	0	0	2	2
11	7	7	512	512	3	3	1	1	1	1

Table 6.1 – Resnet Problem Sizes

0) with smaller depth (3 for layer 0) whereas last layers have smaller height and width (7 x 7 for layer 11) and high value for depth (512 for layer 11).

6.2.3 Parameter specific cost function

Consider the synthetic example shown in the figure 6.5. In this simple example there is data reuse across both i and j loops. These reuse is captured by two RAR dependences for arrays B and C. The actual amount of data reuse depends on parameters N and M. Two instances of the problem with different parameter values of N and M are shown in the Figures 6.3 and 6.4. In this example if we parallelize loop i i.e different iterations of loop i are executed by the different threads, then the amount of data reuse is $\mathcal{O}(M)$. Each thread will execute the entire j loop and element $B[i]$ will be reused M times. Similarly if we interchange loops and then parallelize j loop, then the amount of data reuse is $\mathcal{O}(N)$. Each thread will execute the entire i loop and element $A[i]$ will be reused N times. Hence the actual amount of data reuse depends on the parameter values. We can make use of the actual parameter values to find the better loop transformations that maximize data reuse. In deep learning pipelines we already know the values of the parameters at compile time so we can make use of this to find the problem specific transformations.

In polyhedral transformation framework, we formulate an ILP to find new loop transformations. The cost proposed in Pluto [BHRS08a] is minimize the reuse distance across iterations. In this section we propose to augment this cost function so that it also considers data reuse volume. The main challenge in this regard is formulation of reuse volume as linear cost function. In general it is very difficult to linearise the communication volume constraints. In order to compute the exact amount of data reuse, we need to count the number of points in the dependences. We can use the barvinok [VSB$^+$07] library to compute the number of points in a polyhedra. The barvinok function that counts the number of points in a polyhedra returns a polynomial expression of

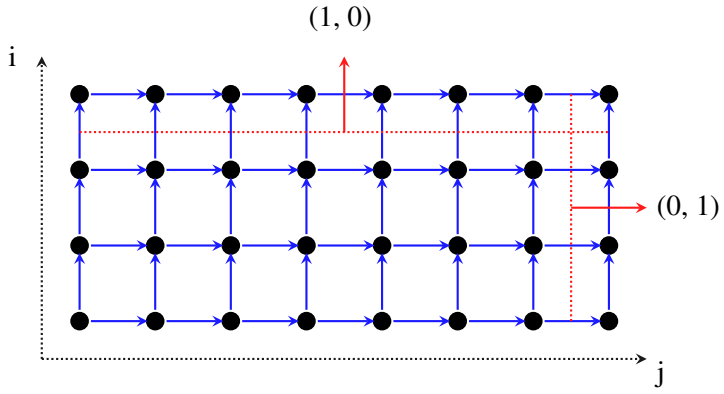

Figure 6.3 – Problem instance with i < 5 and j < 9

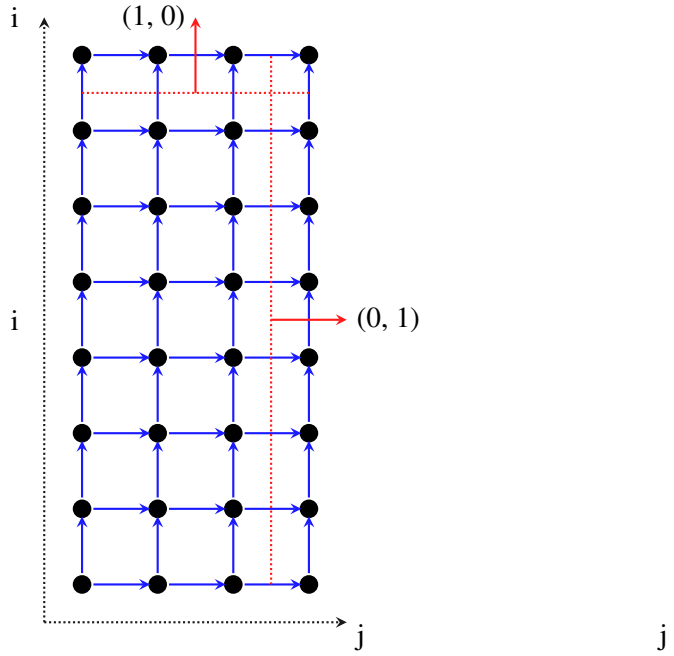

Figure 6.4 – Problem instance with i < 9 and j < 5

```
#pragma scop
   for(int i = 0; i < N; ++i) {
     for(int j = 0; j < M; ++j) {
       A[i][j] = B[i] + C[j];
     }
   }
#pragma endscop
```

Figure 6.5 – Synthetic example

program parameters. It is not possible to express this polynomial expression as a constraints to the ILP. Since we are restricting to cases where we know the exact value of parameters, we can compute the exact value of reuse by evaluating the expression for the particular values of the parameters. We use these values as weights in the cost function of ILP.

$$\text{lexmin} \sum_{i=1}^{n_p} (u_i^- + u_i^+), w, rb, \sum_{i=1}^{n_p} \sum_{j=1}^{n_s} d_{j,i}, \sum_{j=1}^{n_s} \sum_{i=1}^{\dim \mathscr{D}_{S_j}} (c_{j,i}^- + c_{j,i}^+), \tag{6.2}$$

We incorporate the reuse bound constraints by introducing new bound parameter, rb, after the bounds of pluto cost function as shown in 6.2. The ordering of the bounds parameter determines the priority for the new constraints. We are taking the lexmin, the leftmost bounds i.e u_i have the highest priority. In the equation 6.2, u_i represents the sum of positive and negative parts of the dependence distance bounds, w is the constant term of the dependence distance bound, $d_{j,i}$ represents the sum of parameter coefficients, $c_{j,i}$ represents the sum of positive and negative schedule coefficients. Because of the ordering of u_i and rb, reuse bounds gets lower priority than dependence distance bounds of pluto.

$$rb = \sum_{j=1}^{n_s} \sum_{i=1}^{\dim \mathscr{D}_{S_j}} -1 * w_{j,i} * (c_{j,i}^- + c_{j,i}^+), \tag{6.3}$$

The actual reuse bound constraints will be of the form 6.3. The $w_{j,i}$ coefficients captures the exact amount of reuse along the $c_{j,i}$. For each RAR and RAW dependence we compute the domain and range of the relation. For each sets in domain and range and for each dimension $c_{j,i}$ in the set, we project out all the dimension except for $c_{j,i}$, then we compute the total number of points in this projected set using $isl\_union\_set\_card$ from barvinok library. $w_{j,i}$ is the sum of all counts computed for particular value of j, i. Thus $w_{j,i}$ represents the total data reuse along original orthogonal dimensions. These values are negated because we want to maximize the data reuse and the ILP computes lexmin.

For the example shown in the Figure 6.3, the reuse bound constraints are shown in equation 6.4. In this equation $c_{0,1}^-, c_{0,1}^+$ represents the positive and negative parts of i loop dimension and $c_{0,0}^-, c_{0,0}^+$ represents the positive and negative parts of j loop dimension. These constraints forces i loop to be the outermost loop to exploit maximum reuse along j. Similarly for the Figure 6.4, the resue bounds are shown in the equation 6.5. These constraints forces j to be the outermost loop which results in maximum reuse along i dimension.

$$rb = (-5) * c_{0,0}^- + (-5) * c_{0,0}^+ + (-9) * c_{0,1}^- + (-9) * c_{0,1}^+ \tag{6.4}$$

$$rb = (-9) * c^{-}_{0,0} + (-9) * c^{+}_{0,0} + (-5) * c^{-}_{0,1} + (-5) * c^{+}_{0,1} \tag{6.5}$$

6.3 Auto tuning

The second part of the tvm schedule is finding the optimal grid sizes, block sizes, tile sizes, unroll factors etc.. These tuning parameters have a significant impact on the performance of the GPU kernel. Hence, in order to achieve peak performance it is very important to choose the optimal tuning parameters. Different tensor operators have different optimal values for tile sizes, grid sizes, block sizes etc.. Even for same tensor operator these optimal values varies across different GPU architectures. Also, these optimal values varies with sizes and shapes of the tensors. We need to find optimal values for tuning parameters for each tensor operation, for each GPU architecture that is specialized to fixed tensors size and shape.

The most common approach to finding optimal tuning parameters is exhaustive search. Foreach tuning parameter we define a valid range of values. The total search space is all possible combinations of values within these valid range. To find the optimal value of the tuning parameters we iterate over all the valid configurations in the search space while keeping track of configuration with least execution time. For each configuration we generate the instantiated kernel, compile it and run it to measure the execution time. This exhaustive search is guaranteed to find the optimal configuration. But, the iterating over every valid configuration takes a lot of time. Since the optimal configuration is different for different tensor sizes, architectures, we need to repeat the entire search if one of these varies. It is not practical to do expensive exhaustive search for every change in problem size since it will take hours to complete.

There are several techniques proposed to reduce the search space needed to explore. These techniques use several heuristics to selectively choose which candidates to explore based on the performance of already evaluated candidates. These heuristics does not guaranteed to find the optimal configuration even if it is allowed to run for hours. The genetic search for example in each generation a certain number of workers start with different configurations that are slightly different from each other and pick the best performing one the next generation. With each generation there is a definite improvement of execution time but there is no upper bound on the number of generations need to find the optimal configuration. These techniques still take few hours to find a better configuration and the search needs to be repeated for every different architectures and tensor sizes.

Another approach for autotuning is to use GPU performance model. In these techniques, build an analytical model that can estimate the execution time of a kernel. These models is used to filter the search space to eliminate the need to evaluate the worst performing configurations. The success of these methods depends on the accuracy of the performance model to estimate kernel execution time. If the model‘s predicted execution time is far from the actual execution time, we will not be able to prune the large parts of search space or we might prune the optimal configuration. An accurate modelling of GPU performance is a very challenging task. Even for a single kernel, the GPU kernel execution time varies across different architectures and input sizes. To build an accurate performance model one need to model how a given kernel is mapped into gpu warps, how these warps are scheduled, latencies of various instructions, memory access latencies,

warp scheduler etc.. Modelling each of these aspects in a very challenging. For example to model the memory access latencies depends on peak memory bandwidth of the system, weather access is coalesced or not, weather the requested data is cached or not, number of outstanding memory requests. Most of the prosed model are either not very accurate enough or only applicable for limited set of kernels.

6.3.1 Performance model from data

In this section we will show how to build an accurate performance model from sample data of actual execution and use this model during autotuning to select the top few configurations. We want the model that learns the underlying pattern of kernel execution and its variance across input parameter space. We use Multilayer Perceptron (MLP) to build performance model that estimates the execution time. The basic unit of MLP is a linear model that does the dot product of all the inputs and the weights in the layer. A layer is formed by may such independent units that are connected to same inputs but with different weights. A MLP consists of many such layers stacked together with the outputs of previous layer connected as a input to the current layer. It is shown that an MLP with suitable weights can approximate any function. Initially these weights are chosen at random. During the supervised training phase, inputs are propagated through the network to produce the output, which is compared with the actual output to produce the error. In the backpropogation pass, this error is propagated back through all the layers while weights are adjusted using gradient descent. This process is repeated for all the training data, while the weights are getting tuned.

6.3.2 Operator specific model

The main objective to model is to capture the underlying GPU execution pattern and how this various with changes in input sizes and tuning parameters. We build a MLP model that predicts the execution time of a given kernel. The inputs to this model is problem sizes, tuning parameters with execution time as single output. To train the model we collect the data by running kernel for different input sizes and tuning parameters. Since the input and tuning parameter space is possibly infinite we restrict them by choosing reasonable upper bounds. We collect the execution data for the random configurations obtained by uniform sampling of the search space. Note that not all the configurations sampled this way will be valid. If some of the selected configurations turned out to be invalid then we set execution time to be infinity. We train model with 90% of data, the rest is used as validation data set to measure the mean square error. Figure 6.8 shows the changes in error with number of data samples used to train. As the model gets trained with more and more data the mean square errors keeps reducing until it stabilizes. The MLP model like most neural networks a lot of data to converge hence, we may need to iterate the data many times before the model converges. If the obtained error rate is not low enough, then we can either increase the number of hidden layers or increase the amount of training data. In practice we found that a 5 layer MLP model with several thousand samples iterated over multiple times was

sufficient for the error to converge.

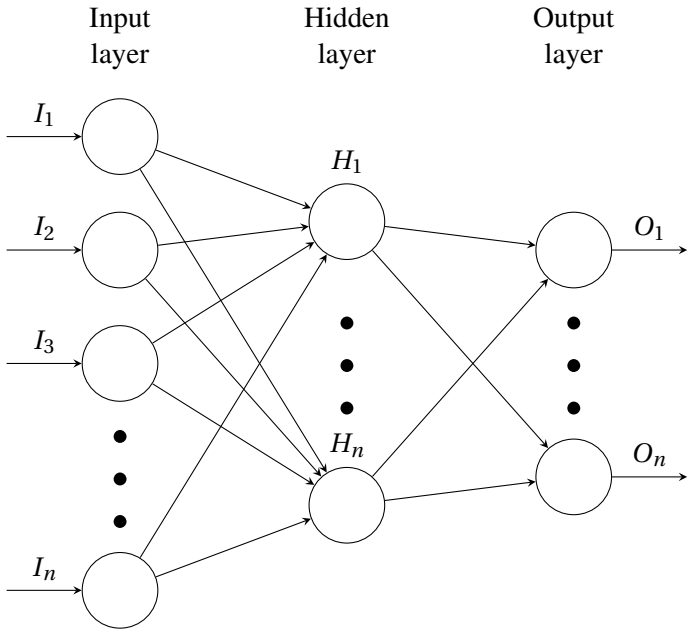

Figure 6.6 – Multi Layered Perceptron(MLP)

Now we have a model that can predict the execution time of an operator for a given configuration. This can used to prune the search space and pick the best configuration for any given problem sizes. We evaluate the model for all valid configurations in the search space while keeping track of top N configurations that have the minimum execution time. Only these top candidates are evaluated on the GPU to choose the best performing configuration. The time to evaluate the model is always a constant for any configuration and is order of magnitude faster than running the kernel on the actual GPU. Hence, even though we evaluate model in the entire search space, there is significant reduction in the time taken to find the best candidates compared to exhaustive evaluation on GPU. Moreover the model can be evaluated in parallel to speedup this process. In practice the MLP model based tuning will take under a minute compared to hours required for exhaustive search on GPU. An accurate model ensures that we will miss any good configurations to evaluate while pruning the bad configurations that take a lot of time to run on GPU. The most time consuming operation in the process is collection of data to build the model, which is in order of hours. Once the model is built for a given operator, the optimal configuration for any given problem size can be obtained in matter of seconds. This approach is good fit for deep learning models where several operators are frequently used with different problem sizes.

6.3.3 Transfer learning across architectures

The performance model that we learned from data is specific to a given architecture. The kernel execution time various across different GPU architectures. Hence, we need to build a different model for each of the different architectures. This requires collecting data on each of these

architectures which will take hours. There is some similarity between different GPU architectures. We can exploit these similarity by using already learnt model as the starting point of training rather than random values. This will reduce the amount of new data needed to converge. We need just few thousand samples to fine tune the model to adapt to new architecture. This transfer learning enables us to quickly port the an operator‘s model to new architectures. Once the model is built offline for an operator on a given architecture, it can ported to other architectures with very little new data.

6.3.4 Transfer learning across operators

We have seen how to build an accurate performance model for a given specific tensor operator. The most time consuming part of the process is the collection of data needed to build the model. An accurate model requires a few hours of sample data. Once the model is built it can predict the execution time for different problem sizes but only for specific operator. Even if the tensor operator various slightly we need to build an new model from the scracth which requires few more hours to data. In this section we explore the possibility of transfer learning across different tensor operators.

In general the performance characteristics greatly differs across different class of programs hence, it will be very challenging to build a universal model that predicts the execution time for all class of programs. However the performance characteristics within same class of programs tend to similar e.g. most 1D stencil computations tend to behave similar to each other. Hence, the idea is to build a performance model for specific class of programs. The key problem is how to differentiate between different class of problems. We need a set of features that are similar for programs within a given class and differ with the programs in a different class. There are various metrics such as amount of data and computation, number of different instructions etc. that are used as feature vectors. For the programs that can be represented in polyhedral model, we think there is a much more accurate feature vector that abstracts the program execution i.e. dependences.

In the polyhedral model, a dependence exits between two iterations if and only if both the iterations access the same data. Depending on the type of accesses the dependence could be Read-after-Read (RAR), Read-after-Write (RAW) , Write-after-Read (WAR) or Write-after-Write (WAW). RAR and RAW dependences capture the exact data reuse between iterations. This data-flow information at the granularity of iterations acts as accurate signature for a given program. Note that, in most of the AST representations the data-flow information is Also, theses dependences can be concisely represented in matrix form. These two properties makes dependences an excellent choice as feature vector for building performance models. By using dependences as feature vectors we are grouping the programs with similar dependence into same class. The programs with similar dependence pattern tend to have similar performance characteristics and hence they act as a good classifier.

We try to build a general performance model for TVM operators with dependences as input feature vector. Various TVM operator could have different number of iterators. We assume that there could be maximum of 7 iterators, this is was sufficient for the deep learning primitives that we are interested in. The trip count of different loops is captured in the domain, these are also encoded in the feature vector. Different TVM operators could have variable number of dependences e.g. Matmul operator has two RAR dependences for the access A and B, and one RAW dependence for C. We merge multiple type of dependences together by taking a union. In addition we also add maximum reuse weights along each dimension to account for the amount of reuse. The complete feature vector include the loop bounds, dependence vectors along with the reuse bounds and the tuning parameters. To build the general performance model, we collect the uniformly sampled data with the mix of different operators. The model now captures the execution profile for different operators classified as per dependence pattern. Thus the dependence as a feature vector enables transfer learning across different TVM operators.

Problem Size	abbreviation	Pluto	Adaptive Pluto
(1, 64, 56, 256, 1, 1, 0)	ps1	0.99	1.13
(1, 256, 56, 64, 1, 1, 0)	ps2	2.44	2.6
(1, 256, 56, 128, 1, 2, 0)	ps3	2.5	3.1
(1, 128, 28, 512, 1, 1, 0)	ps4	2.4	2.9
(1, 256, 56, 512, 1, 2, 0)	ps5	2.1	2.7
(1, 512, 28, 128, 1, 1, 0)	ps6	3.4	3.9
(1, 512, 28, 256, 1, 2, 0)	ps7	2.4	3.2

Table 6.2 – SpeedUp of Convolution kernel in Resnet-50 with NCHW layout

6.4 Experimental results

All the experiments are performed on with Nvidia Quadro P6000 GPU, powered by Nvidia Pascal architecture attached to Intel Xeon CPU E5-2683 with 128GB RAM. All the benchmarks are compiled with the NVIDIA CUDA 9.0 toolkit, with the `-O3` optimization flag.

6.4.1 Adaptive schedule

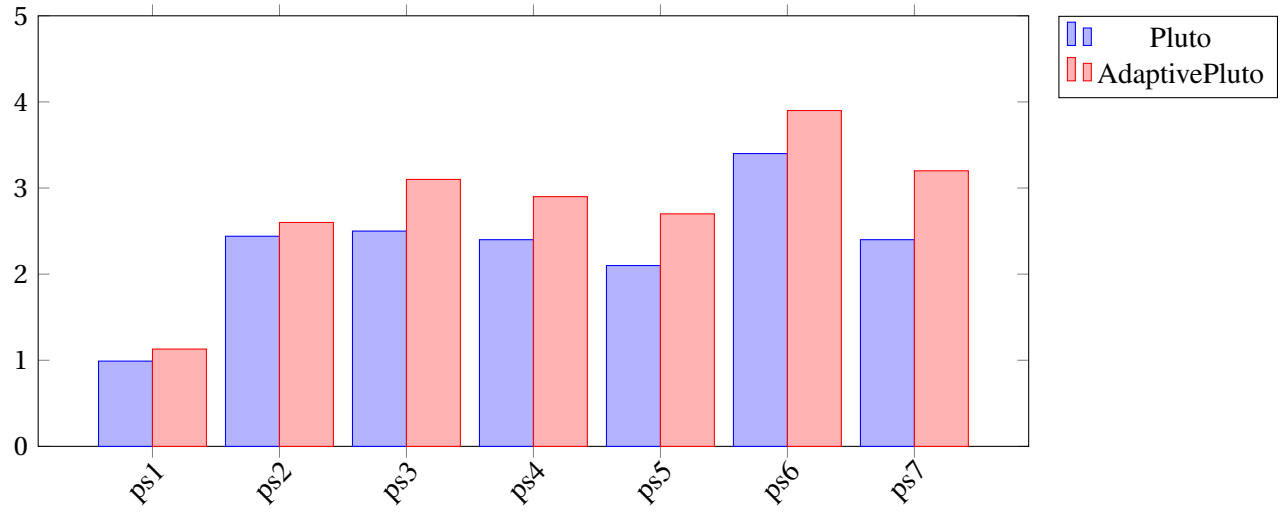

Figure 6.7 – SpeedUp of Convolution kernel with Resnet-50 workloads

Table 6.2 shows the speedup of convolution kernel with Resnet-50 workloads. The baseline for the experiments is the manual tvm schedule. For all the experiments the schedule is computed with scheduler and optimal parameter values are computed through autotuning. The adaptive schedule performs better than the default pluto schedule in all the cases. This shows the significance of data reuse constraints, which adopts the schedule as per tensor sizes and batch sizes. When the

batch size is small, it is more profitable to parallelize channel, height and width loop, on the other hand if the batch size is large it is more profitable to parallelize batch, channel, height loops. The adaptive schedule enables such transitions compared to the fixed schedule of default pluto.

6.4.2 Model accuracy

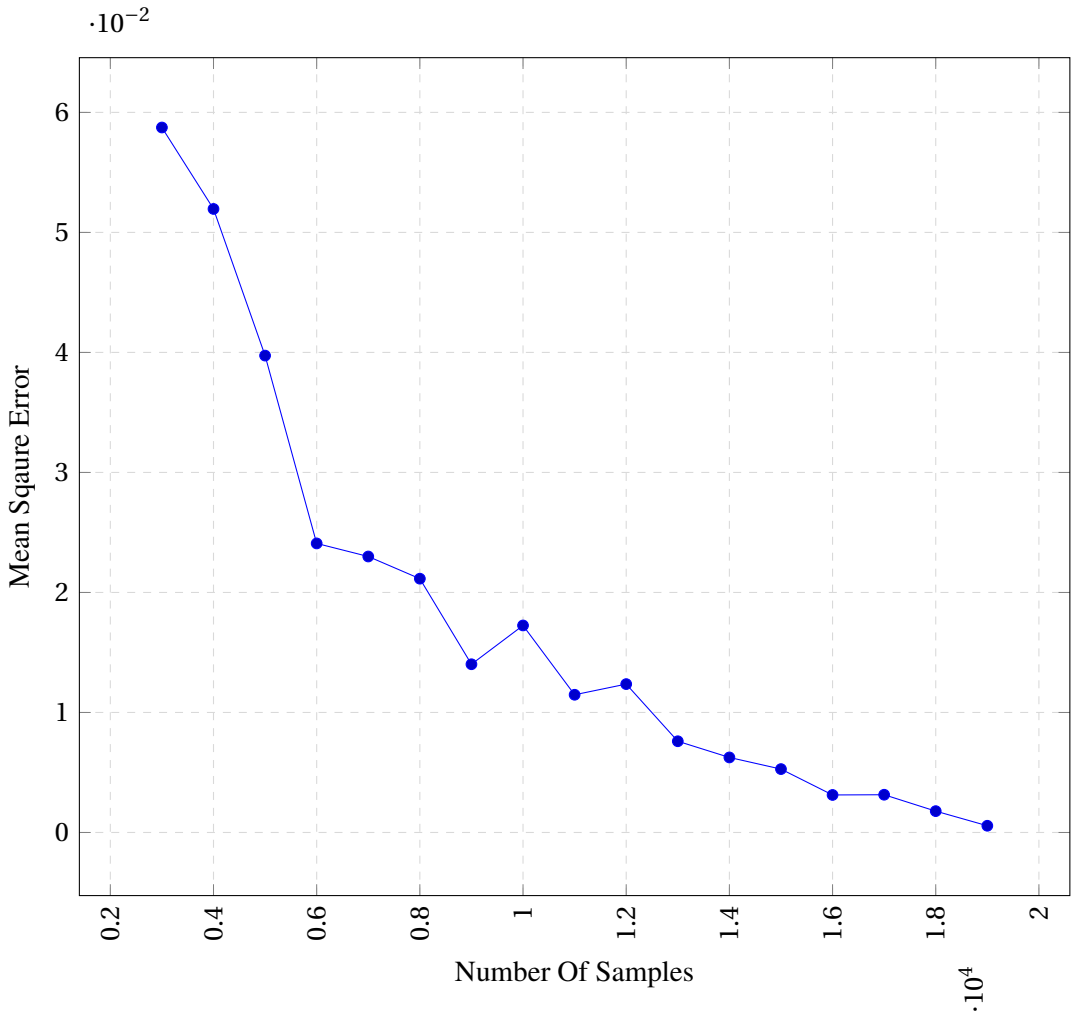

Figure 6.8 – Reduction in Mean square error with number of samples

Figure 6.8 shows the reduction in the mean square error with the increase in the number of samples used to build the model. The mean square error for at T samples is measured y training the MLP model with randomly chosen T samples and evaluating the mean square error of execution times for $N - T$ samples where N is the total number of samples in the search space. At the beginning of the training all the weights of the model are randomly initialized. The model is trained with batch size of 50. The choice of batch size up to 500 does not seem to have an effect on the accuracy of the model. The model is trained with the same training samples multiple

Problem Size	MLP Best Conf	Exhaustive Best Conf	MLP Time/ Exhautive time
(5124, 9124, 2560)	(8, 16, 4, 4, 8)	(4, 8, 4, 4, 16)	1.03
(35, 8457, 2560)	(4, 16, 8, 4, 4)	(4, 16, 8, 4, 4)	1.09
(7680, 16, 2560)	(4, 16, 8, 8, 4)	(4, 8, 16, 4, 4)	1.01
(3072, 64, 1024)	(4, 8, 16, 4, 4)	(4, 8, 16, 4, 4)	0.96
(7680, 5481, 2560)	(8, 32, 4, 8, 4)	(8, 32, 4, 8, 4)	1.00
(1024, 700, 512)	(8, 16, 4, 8, 4)	(8, 16, 4, 8, 4)	0.98
(8448, 24000, 2816)	(8, 32, 4, 8, 4)	(8, 32, 4, 8, 4)	1.00
(8448, 48000, 2816)	(8, 32, 4, 8, 4)	(8, 32, 4, 8, 4)	0.99
(512, 48000, 2816)	(4, 32, 4, 16, 4)	(4, 32, 4, 16, 4)	1.01

Table 6.3 – Matrix multiplication kernel MLP tune vs Exhaustive search

times. This repeated training of model is referred to as multiple EPOCH training. The accuracy of the model continuous to increases with multiple EPOCH. We need up to 20 EPOCHs before the mean square error stabilizes. As shown in the figure 6.8 we get more accurate performance models with the higher number of randomly chosen training samples. The MLP model for matrix multiplication needs around 20 thousand training samples to build performance model that can be used for auto tuning. This whole process to build a performance model with 20 thousand samples takes only few minutes in Pytorch. The majority of time is spent on collecting the training data by randomly sampling the search space, generating the kernel for the chosen sample, running it on the GPU and collecting the execution time of the kernel. This entire process will take around 30 minutes for Matrix multiplication kernel for given GPU architecture.

6.4.3 MLP tuning vs Exhaustive

Once we built the model for a given kernel and a given architecture, it can used to find the optimal configuration for any given problem sizes. The model is evaluated on all the valid configurations in the search space. We keep track of top 100 configurations as evaluated by MLP performance model. A kernel is generated for each of the top 100 configurations. These kernels are run on GPU and the kernel with least running time is chosen among top 100 kernels. Table 6.3 shows the performance comparisons of MLP based tuning and Exhaustive search for a few selected problem sizes in DeepBench [Bai]. This experiments were conducted on reduced search space that includes only powers of two parameters values. The exhaustive search take hours to complete on full search space. As shown in the table MLP based tuning finds the same configuration as the Exhaustive search for most of the problem sizes. Even for the cases when the exhaustive configuration was different from the MLP tune configuration, the runtime of the kernels are very close. We did the same experiments for all the 150 different problem sizes listed in the DeepBench [Bai]. Out of 150 problem sizes both MLP tune and Exhaustive search finds the same configuration for 103 problem sizes. For the rest 47 problem sizes where the configurations were different the difference in kernel runtime is within 1% of the exhaustive time. This shows the effectiveness of MLP-tune in which we are able to find close to optimal configuration in minutes

rather than hours as in Exhaustive search.

6.4.4 Performance of Generated kernels

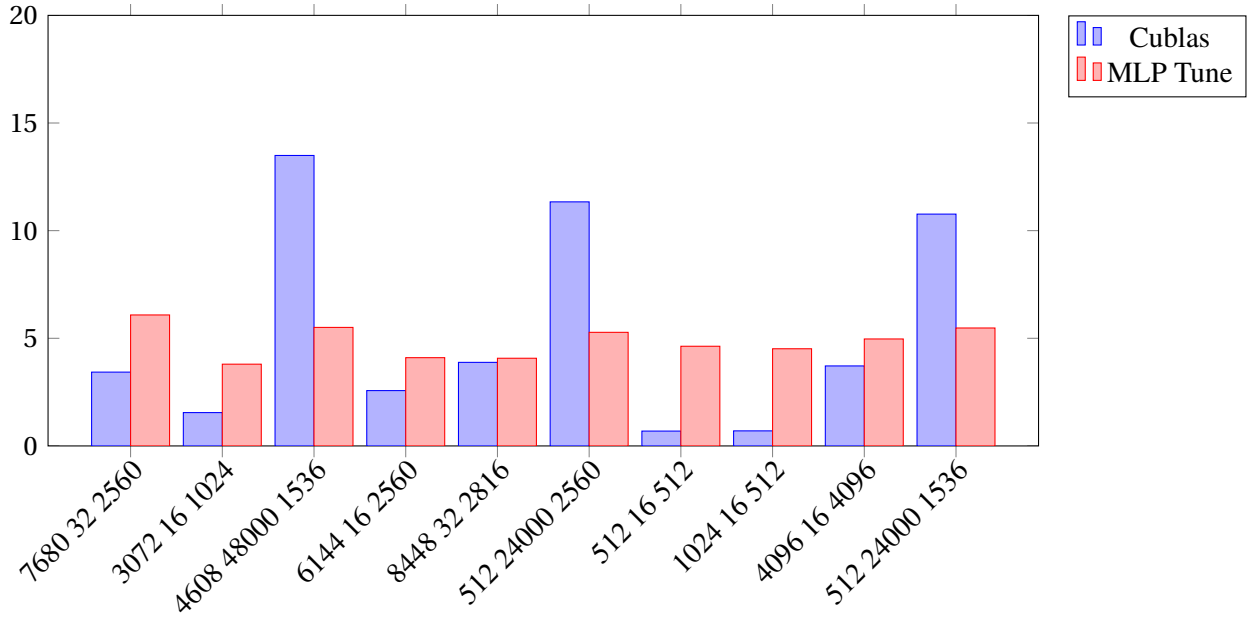

Figure 6.9 – Speedup MLP Tune vs. Cublas

Figure 6.9 shows the performance comparisons of MLP tune generated kernel vs Cublas (Nvidia's optimized GEMM library). The baseline used for comparisons is TVM manual schedule. The TVM library has a single manually specified schedule for GEMM. A single schedule will not be optimal for all problem sizes. A significant performance $2.4x$ to $20x$ can be gained by autotuning and finding optimal schedule for a given problem size. Cublas is a highly optimized library that includes various ninja optimizations to achieve near peak performance. Not all of these optimizations can be implemented with TVM especially low level assembly optimizations. Hence for some problem sizes Cublas is significant faster than MLP tune. However there are several problem sizes for which MLP tune outperforms Cublas. This shows the limitations of library based approaches. Even though highly optimized Cublas is not optimal for every problem size. Having a tunable compiler framework with autotuning can provide flexibility to generate optimal kernel for any given size. Compiler based approach will also be able to perform optimizations across library calls, such as fusion, and can generate highly optimal kernel for any given sequence of library calls.

6.4.5 Tuning Convolution kernels

Figure 6.10 shows the speedup with MLP tune and exhaustive search for Convolution kernel in HWCN data layout. Here H and W are height and width of the input image, C represents number

Problem Size	Abbreviation	SpeedUp With Exhaustive	SpeedUp With MLP Tune
[1, 3, 224, 64, 7, 2]	ps1	18.38395385	16.64207728
[1, 64, 56, 64, 3, 1]	ps2	6.969599756	6.642836342
[1, 64, 56, 64, 1, 1]	ps3	4.862487361	4.448658649
[1, 64, 56, 128, 3, 2]	ps4	7.019564531	6.289228159
[1, 64, 56, 128, 1, 2]	ps5	4.907020873	4.098256735
[1, 128, 28, 128, 3, 1]	ps6	7.234597156	6.465480729
[1, 128, 28, 256, 3, 2]	ps7	19.74151355	17.59855636
[1, 128, 28, 256, 1, 2]	ps8	5.122047244	4.38047138
[1, 256, 14, 256, 3, 1]	ps9	7.780074551	7.016809291
[1, 256, 14, 512, 3, 2]	ps10	9.284042899	8.331000292
[1, 256, 14, 512, 1, 2]	ps11	5.882042254	5.47704918
[1, 512, 7, 512, 3, 1]	ps12	19.9703272	19.57655868
[1, 128, 122, 128, 3, 1]	ps13	7.436881548	7.398414548
[1, 1, 224, 64, 5, 1]	ps14	18.71373782	18.68936724
[1, 64, 224, 64, 3, 1]	ps15	7.66477562	7.595536122
[1, 64, 224, 32, 3, 1]	ps16	36.97738925	33.1224634
[1, 32, 224, 9, 3, 1]	ps17	22.74328874	22.98921855

Table 6.4 – SpeedUp MLP Tune and Exhaustive Search over TVM Manual schedule

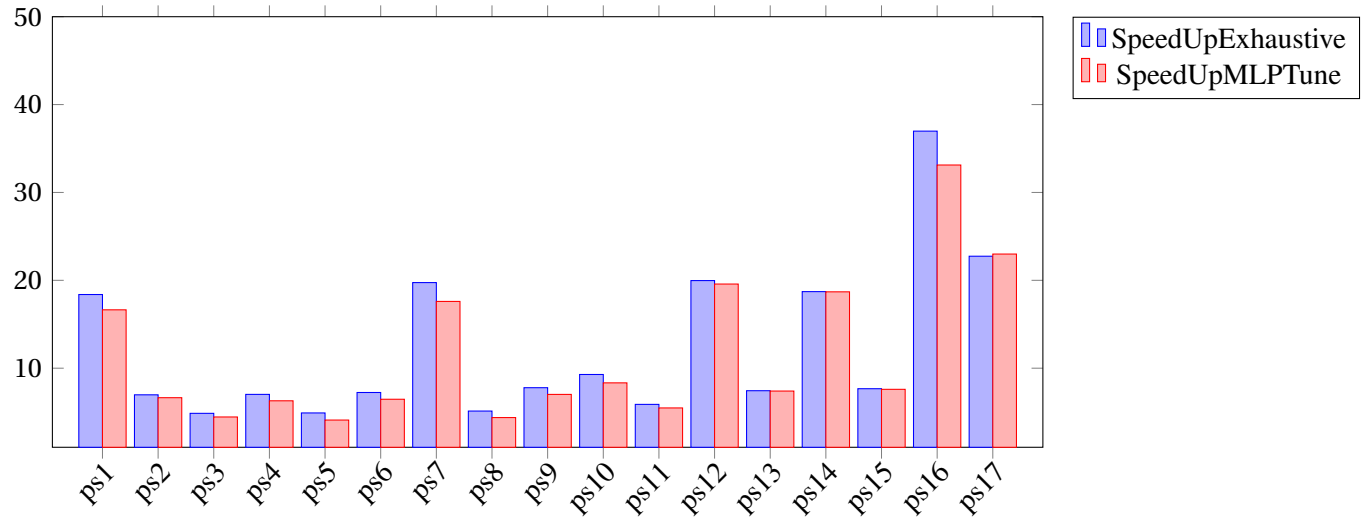

Figure 6.10 – Speedup MLP Tune vs. Exhaustive for Convolution HWCN

of channels in the input and N in the number of images in the batch. We can observe that there is significant speedup with autotuning and MLP based tuning closely follows the speed up with Exhaustive search even with the large search space for convolution kernel. The search space for convolution tuning is exponentially larger than matrix multiplication kernel in the order of $10E7$. Even with such large space MLP tune pruned most of the candidates and pick the top 100 candidates whose performance matches closely to Exhaustive search obtained by evaluating the

entire search space.

6.4.6 MLP Tune vs other search techniques

There are several search techniques that can be used for autotuning. [MAS+16] proposed a heuristics based approach to automatically find the schedules for CPUs in Halide. The heuristics are derived based on the maximizing the arithmetic intensity using bounds analysis. [LGA+18] extended this approach to find schedules for convolution kernels on GPUs. Compared to this manually derived heuristics that are specific to given operator we learn the performance model automatically in a operator and architecture agnostic way. Opentuner [AKV+14] a framework for proposed for building domain-specific multi-objective program autotuners. The OpenTuner uses ensembles of disparate search techniques simultaneously for autotuning. It will automatically select the best performing search technique for a given objective and search space. In this section we present comparisons MLP tune to various other search techniques in Opentuner.

The comparison with other search techniques is performed on convolution kernel with problem sizes as $(1, 3, 224, 64, 7, 2)$. The search space is reduced with only powers of two for the parameter values. The baseline for all the measurements is tvm default schedule. The MLP tune for this example will result in 17.5× speed up. All the search techniques are run with 15 mins and 30 mins time out value. When the best time is infinity indicates that the technique was not able to find any valid schedule before the timeout. As shown in the table 6.5 there are several techniques such as RandomNelderMead, RandomTorczon etc. which are unable to find any valid scheudle even after 30 mins. The maximum speedup is obtained is 11.01× by ga-OX3 (a variant of genetic algorithm) that ran for 30 mins. This is much less than 17.5× speedup obtained using MLPTune within few minutes. This shows the benefits of performance model based MLP tuning. The other big advantage is reuse of patters that is learned only once in MLP tuning where as with the generic search techniques with every new problem sizes, search is started afresh. There is no reuse of the knowledge that is learnt from previous searches. The same performance model once built is reused for different problem sizes.

6.4.7 Transfer Learning

The largest time consuming part of the MLP tuning is collecting data to build the performance model. Depending on size of search space it will take around 30 mins to 1 hr to collect uniformly sampled training data. Ideally we need to build performance model one for each architecture since each architecture could have different performance characteristics. But since there is some similarity between GPU architectures. We can exploit this similarity by reusing the already learnt weights across architectures.

Figure 6.10 shows the Mean Square Error with random initialized weights vs. starting with trained performance model on another architecture. As we can observe the starting error is quite low if we start with the weights of the pre trained model. It requires fewer training samples on the

Technique	Best time (30 min)	Speedup (30 min)	Best time (15 mins)	Speed up (15 mins
PureRandom	2.16	8.10	1.63	10.74
ga-OX3	1.59	11.01	inf	nan
ga-OX1	1.63	10.75	1.90	9.23
ga-PX	inf	nan	1.67	10.48
ga-CX	1.63	10.75	1.67	10.49
ga-PMX	1.86	9.43	inf	nan
ga-base	1.63	10.74	1.73	10.15
UniformGreedyMutation05	inf	nan	1.72	10.17
UniformGreedyMutation10	1.73	10.15	1.60	10.97
UniformGreedyMutation20	1.76	9.98	1.63	10.75
NormalGreedyMutation05	inf	nan	inf	nan
NormalGreedyMutation10	3,713.82	0.00	130.75	0.13
NormalGreedyMutation20	inf	nan	inf	nan
DifferentialEvolution	1.87	9.40	1.86	9.42
DifferentialEvolutionAlt	1.86	9.44	1.90	9.24
DifferentialEvolution_20_100	1.60	10.98	1.63	10.73
RandomNelderMead	inf	nan	inf	nan
RegularNelderMead	inf	nan	5.75	3.05
RightNelderMead	1,131.74	0.02	inf	nan
MultiNelderMead	5.28	3.32	13.68	1.28
RandomTorczon	inf	nan	1.59	11.02
RegularTorczon	3.63	4.83	inf	nan
RightTorczon	inf	nan	10,519.13	0.00
MultiTorczon	5.33	3.29	4.75	3.69
PatternSearch	inf	nan	inf	nan
PseudoAnnealingSearch	1.63	10.76	1.86	9.43
pso-OX3	1.84	9.52	inf	nan
pso-OX1	1.67	10.47	1.75	10.00
pso-PMX	1.83	9.58	2.23	7.87
pso-PX	2.01	8.72	2.13	8.24
pso-CX	inf	nan	1.73	10.14
GGA	3,385.71	0.01	inf	nan
AUCBanditMutationTechnique	inf	nan	inf	nan
AUCBanditMetaTechniqueA	1.69	10.39	1.73	10.15
AUCBanditMetaTechniqueB	1.59	11.01	1.90	9.23
AUCBanditMetaTechniqueC	2.77	6.33	2.76	6.34
PSO_GA_Bandit	1.73	10.13	2.02	8.70
PSO_GA_DE	1.63	10.74	1.60	10.96
ComposableDiffEvolution	1.87	9.39	2.02	8.68
ComposableDiffEvolutionCX	1.90	9.22	1.68	10.45
AUCBanditMetaTechniqueA	1.63	10.74	inf	nan

Table 6.5 – Search techniques comparison

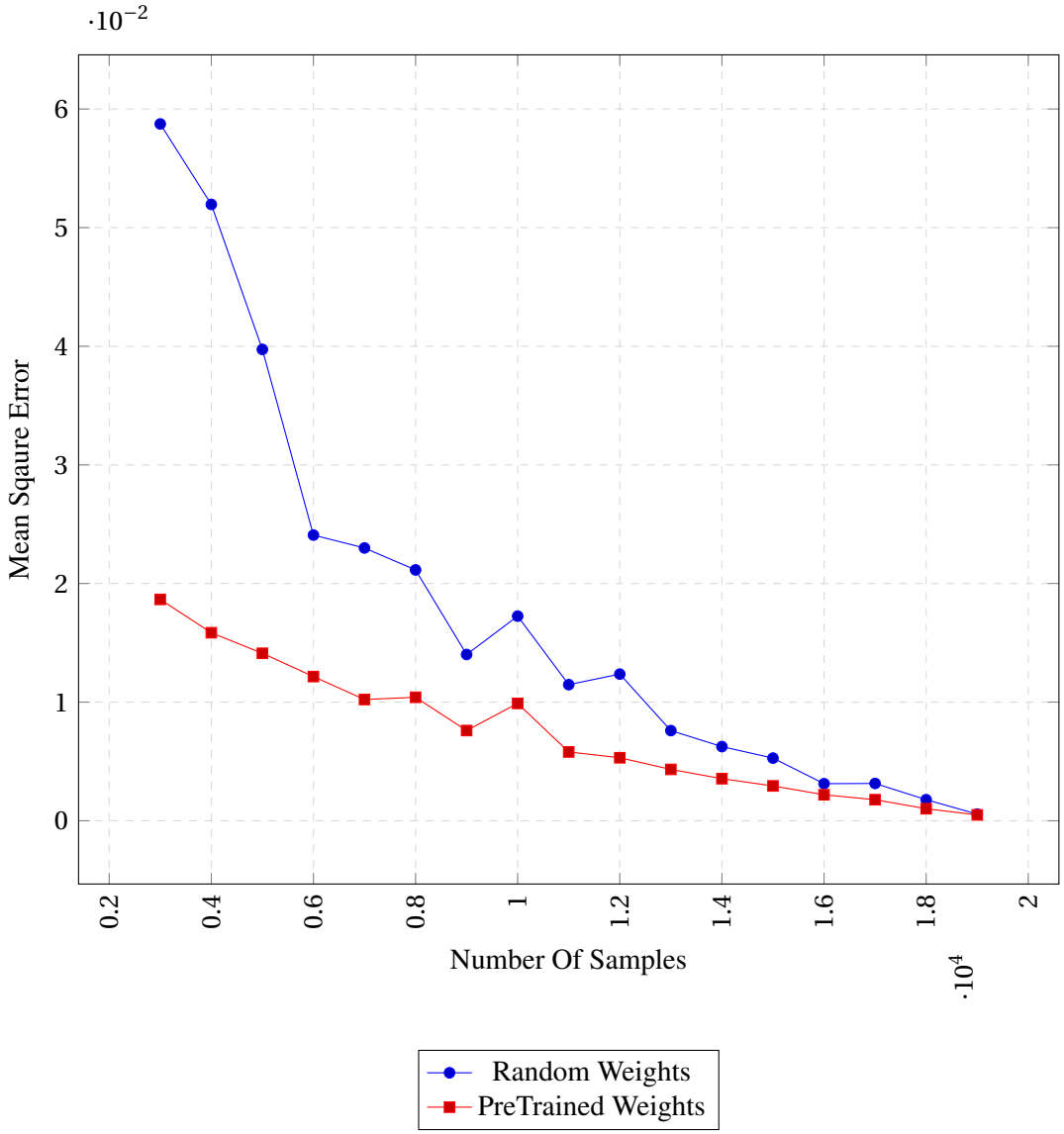

Figure 6.11 – Reduction in Mean square error with pre existing model

new architecture to build performance model. This is a very attractive feature of the MLP tunes with allows us to reuse performance model patterns across architectures. All the other search based tuning techniques need to start from scratch.

7 Conclusion and Perspectives

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

7.1 Conclusion

Loop transformations such as tiling and fusion are essential to achieve high performing on complex modern hardware with deep cache hierarchies. In recent years, the polyhedral transformation model is proven to a very effective technique for performing loop transformations in both industry and research compilers. The main advantage of the model is the rigorous formalism it provides in terms of precisely analysing loop dependencies at the granularity of individual iterations. These dependencies makes it possible to formulate the problem of finding profitable loop transformation as an ILP problem. There were many cost functions that were proposed as an objective to this ILP, Pluto being the most practical and successful. In this dissertation, we study few possible ways to improve upon the pluto heuristic by modelling hardware characteristics such as spatial locality and application specific information such as reductions. We also extend it to model data reuse volume so that schedules that are adaptive to problem sizes.

7.1.1 Handling reductions

We presented language constructs and compilation methods to express arbitrary reductions on user-defined data types, and dependence-based abstractions of reductions that enable polyhedral loop nest optimizations on loops carrying reductions. Combining polyhedral and template-based code generation, we are able to perform complex optimizations for user-defined reductions on GPUs, reaching close to peak performance. This approach enables a generic polyhedral compiler to produce highly efficient code for reductions while offering maximum expressiveness to the programmer.

7.1.2 Spatial locality

We proposed an affine scheduling algorithm that accounts for multiple levels of parallelism and deep memory hierarchies, modeling both temporal and spatial effects without imposing a priori limits on the space of possible transformations. The algorithm orchestrates a collection of parametrizable optimization problems, with configurable constraints and objectives, addressing non-convexity without increasing the number of discrete variables in linear programs, and modeling schedules with linearly dependent dimensions that are out of reach of a typical polyhedral optimizer.

Our algorithm is geared towards the unified modeling of both temporal and spatial locality effects resulting from the cache hierarchy of CPUs and GPUs. It generates sequential, parallel or accelerator code in one optimization pass, matching or outperforming comparable frameworks, whether polyhedral, syntactic, or a combination of both. We discuss the rationale for this unified algorithm in much detail, as well as its validation on representative computational programs.

7.1.3 SLAMBench

We studied the challenges of using polyhedral compilation for large end-to-end real-world benchmark called SLAMBench. We shown that static affine restrictions of the polyhedral model can be eliminated in some cases with the summary functions by encapsulating the non-affine parts of the program. This is a powerful abstraction that is needed for real-world benchmarks. We also show the need of a runtime to avoid redundant data copies between device and host. We propose and implement prl runtime library which allows programmers to express array data-flow information of non-affine parts of the program. Our experimental results show that we achieve same performance as the manually optimized SLAMBench kernels drastically reducing the programmer effort while proving portable performance.

7.1.4 DeepLearning pipelines

We explored the challenges with polyhedral compilation of deep learning pipelines. We extract the polyhedral representation from TVM and automatically determine the TVM schedule using polyhedral ILP scheduler and autotuning. We propose additional linear data reuse constraints for ILP scheduler with objective to minimize overall communication volume. This enables the ILP to adapt the schedule to the changes in tensor sizes and shapes. We also proposed a performance model based autotuning technique that drastically reduces the autotuning time. We showed that we can automatically build an accurate performance model for any given operator on given architecture. We pruned the search space with this accurate performance model and were able to pick only few hundred top candidates. We were able to cut down the autotuning time from hours to minutes since we evaluate only these top candidates on hardware. Our experimental results show that we were able to achieve the same performance as expensive exhaustive search in most cases. We showed that our technique performs much better that the traditional search

based techniques and also enables transfer learning.

7.2 Future Work

In this dissertation we studied some of the challenges of using polyhedral model for loop transformations in DSL compilers. We particularly extended the scheduler to handle applicable specific information such as reductions, parameter values and modelling hardware characteristics such as spatial locality, coalescing. However the cost function used is still a heuristic and does not model all the hardware characteristics. The major restriction is using all the new constraints has to be linear for the ILP solver. This is one of the major limitation in coming up with a more realistic cost function that accurately models the actual execution time on the hardware.

7.2.1 Learning hardware specific cost functions

Ideally we want to choose an accurate cost function specific to a given function and hardware that predictions the execution time for different schedules. Then we can choose the best performing schedule among all the valid schedules. This is a very challenging task. Given the recent advances with the deep learning, it is possible to automatically learn such cost functions given enough data. This problem has been explored before in the context of iterative optimizations [PBCV07, PBB+10]. There is scope to improve on the efficiency of such techniques using deep learning.

7.2.2 Fusion heuristics

Similar to work flow proposed for autotuning, we can build an accurate model to predict the performance of fused kernels. There are previous works [BDGR10, JB18] on building the analytical models for fusion which are not hardware specific. We can build much more accurate models automatically from the actual execution times on hardware. Most of the heuristics used in the compilers can be replaced with more accurate automatically learning deep learning models.

A An appendix

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

Bibliography

[AIM17] Martín Abadi, Michael Isard, and Derek G Murray. A computational model for tensorflow: an introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 1–7. ACM, 2017.

[AK84] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pages 233–246, New York, NY, USA, 1984. ACM.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542, October 1987.

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

[BAC16] Uday Bondhugula, Aravind Acharya, and Albert Cohen. The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests. *ACM Transactions on Programming Languages and Systems*, 38(3):12:1–12:32, April 2016.

[Bai] Baidu-Research. Deepbench.

[Bas04] Cedric Bastoul. Code Generation in the Polyhedral Model Is Easier Than You Think. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[BB13] Somashekaracharya G Bhaskaracharya and Uday Bondhugula. Polyglot: a polyhedral loop transformation framework for a graphical dataflow language. In *International Conference on Compiler Construction*, pages 123–143. Springer, 2013.

[BBC+15] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Javed Absar, Sven van Haastregt, Alexey Kravets, et al. Pencil: A platform-neutral compute intermediate language for accelerator programming. *Proc. Parallel Architectures and Compilation Techniques (PACT'15)*, 2015.

[BBK+08] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, Jagannathan Ramanujam, Atanas Rountev, and Ponnuswamy Sadayappan. Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, pages 132–146, Budapest, Hungary, 2008. Springer.

[BCG+15] Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Anton Lokhmotov, Javed Absar, Sven Van Haastregt, Alexey Kravets, and Alastair Donaldson. PENCIL Language Specification. Research Report RR-8706, INRIA, May 2015.

[BDGR10] Uday Bondhugula, Sanjeeb Dash, Oktay Gunluk, and Lakshminarayanan Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *Parallel Architectures and Compilation Techniques (PACT), 2010 19th International Conference on*, pages 343–352. IEEE, 2010.

[BGDR10] Uday Bondhugula, Oktay Gunluk, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. A Model for Fusion and Code Motion in an Automatic Parallelizing Compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 343–352, New York, NY, USA, 2010. ACM. IBM XL.

[BHRS08a] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *ACM SIGPLAN Notices*, 43(6):101–113, 2008.

[BHRS08b] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, volume 43, pages 101–113. ACM, 2008.

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

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

[DCS02] Steven J Deitz, Bradford L Chamberlain, and Lawrence Snyder. High-level language support for user-defined reductions. *The Journal of Supercomputing*, 23(1):23–37, 2002.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[DSHB15] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. *arXiv preprint arXiv:1505.07716*, 2015.

[Fea88] Paul Feautrier. Parametric Integer Programming. *Revue française d'automatique, d'informatique et de recherche opérationnelle.*, 22(3):243–268, 1988.

[Fea91] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[Fea92a] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time. 21(5):313–347, October 1992.

[Fea92b] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[FHLLB09] Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[FL11] Paul Feautrier and Christian Lengauer. Polyhedron Model. In David Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer US, 2011.

[GGL12a] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010, December 2012.

[GGL12b] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly—performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010, 2012.

[Goo] Google. Tensor flow xla.

[GR06] Gautam Gupta and Sanjay V Rajopadhye. Simplifying reductions. In *POPL*, volume 6, pages 30–41, 2006.

[HVF+13] Tom Henretty, Richard Veras, Franz Franchetti, Louis-Noël Pouchet, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A stencil compiler for short-vector simd architectures. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pages 13–24. ACM, 2013.

[HWMD14] Ankur Handa, Thomas Whelan, John McDonald, and Andrew J Davison. A benchmark for rgb-d visual odometry, 3d reconstruction and slam. In *Robotics and automation (ICRA), 2014 IEEE international conference on*, pages 1524–1531. IEEE, 2014.

[ISO99] ISO. The ANSI C standard (C99). Technical Report WG14 N1124, ISO/IEC, 1999.

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

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

[JD89] Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd international conference on Supercomputing*, pages 186–194. ACM, 1989.

[Jou86] Pierre Jouvelot. Parallelization by semantic detection of reductions. In *ESOP 86*, pages 223–236. Springer, 1986.

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

[KM93] K. Kennedy and K. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, pages 301–320, 1993.

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

[KVS+13] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. When polyhedral transformations meet simd code generation. In *ACM SIGPLAN Notices*, volume 48, pages 127–138. ACM, 2013.

[LGA+18] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)*, 37(4):139, 2018.

[Mar] Mark Harris. Optimizing parallel reduction in cuda.

[MAS+16] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):83, 2016.

[Mic] Microsoft. Parallel patterns library.

[MPI96] MPIF MPIF. Mpi-2: Extensions to the message-passing interface. *University of Tennessee, Knoxville*, 1996.

[MVB15] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. Polymage: Automatic optimization for image processing pipelines. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 429–443. ACM, 2015.

[NBZ+14] Luigi Nardi, Bruno Bodin, M. Zeeshan Zia, John Mawer, Andy Nisbet, Paul H. J. Kelly, Andrew J. Davison, Mikel Luján, Michael F. P. O'Boyle, Graham D. Riley, Nigel Topham, and Steve Furber. Introducing slambench, a performance and accuracy benchmarking methodology for SLAM. *CoRR*, abs/1410.2167, 2014.

[NIH+11] Richard A Newcombe, Shahram Izadi, Otmar Hilliges, David Molyneaux, David Kim, Andrew J Davison, Pushmeet Kohi, Jamie Shotton, Steve Hodges, and Andrew Fitzgibbon. Kinectfusion: Real-time dense surface mapping and tracking. In *Mixed and augmented reality (ISMAR), 2011 10th IEEE international symposium on*, pages 127–136. IEEE, 2011.

[Nvia] Nvidia. Cub's collective primitives.

[Nvib] Nvidia. Thrust c++ library.

[Nvic] Nvidia forum. Faster parallel reductions on kepler.

[Opea] OpenMP forum. Openmp 3.0 specification.

[Opeb] OpenMP forum. Openmp 4.0 specification.

[PBB+10] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE Computer Society, 2010.

[PBB+11] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop Transformations: Convexity, Pruning and Optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM.

[PBCV07] Louis-Noel Pouchet, Cedric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *Code Generation and Optimization, 2007. CGO'07. International Symposium on*, pages 144–156. IEEE, 2007.

[PCB+06] 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*, pages 179–197, 2006.

[PCC+] Adam Paszke, Soumith Chintala, Ronan Collobert, Koray Kavukcuoglu, Clement Farabet, Samy Bengio, Iain Melvin, Jason Weston, and Johnny Mariethoz. Pytorch: Tensors and dynamic neural networks in python with strong gpu acceleration, may 2017.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*, pages 444–448. ACM, 1995.

[PP94] Shlomit S Pinter and Ron Y Pinter. Program optimization and parallelization using idioms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):305–327, 1994.

[PW94a] William Pugh and David Wonnacott. Static Analysis of Upper and Lower Bounds on Dependences and Parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278, July 1994.

[PW94b] 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.

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[RF93] Xavier Redon and Paul Feautrier. Detection of recurrences in sequential programs with loops. In *PARLE'93 Parallel Architectures and Languages Europe*, pages 132–145. Springer, 1993.

[RF94] Xavier Redon and Paul Feautrier. Scheduling reductions. In *Proceedings of the 8th international conference on Supercomputing*, pages 117–125. ACM, 1994.

[RF00] Xavier Redon and Paul Feautrier. Detection of scans. *PARALLEL ALGORITHMS AND APPLICATION*, 15(3-4):229–263, 2000.

[RKBA+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013.

[RP99] Lawrence Rauchwerger and David A Padua. The lrpd test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *Parallel and Distributed Systems, IEEE Transactions on*, 10(2):160–180, 1999.

[Sar97] Vivek Sarkar. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM J. Res. & Dev.*, 41(3), May 1997.

[SKG+14] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noel Pouchet, Fabrice Rastello, J Ramanujam, and P Sadayappan. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Notices*, volume 49, pages 65–76. ACM, 2014.

[SKN96] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th international conference on Supercomputing*, pages 18–25. ACM, 1996.

[SLLM06] Eric Schweitz, Richard Lethin, Allen Leung, and Benoit Meister. R-stream: A parametric high level compiler. *Proceedings of HPEC*, 2006.

[SPS14] J. Shirako, L. N. Pouchet, and V. Sarkar. Oil and Water Can Mix: An Integration of Polyhedral and AST-Based Transformations. In *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 287–298, November 2014.

[SSPS15] Alina Sbîrlea, Jun Shirako, Louis-Noël Pouchet, and Vivek Sarkar. Polyhedral optimizations for a data-flow graph language. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 57–72. Springer, 2015.

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

[The] The Portland Group. Pgi accelerator compilers with openacc directives.

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

[tvm] tvm. Tvm documentation.

[VC16] Sven Verdoolaege and Albert Cohen. Live Range Reordering. In *6th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*, Prague, Czech Republic, 2016.

[VCJC+13a] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.

[VCJC+13b] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral Parallel Code Generation for CUDA. 9(4):54:1–54:23, January 2013.

[Ver10] Sven Verdoolaege. Isl: An Integer Set Library for the Polyhedral Model. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, number 6327 in Lecture Notes in Computer Science, pages 299–302. Springer Berlin Heidelberg, September 2010.

[Ver11] Sven Verdoolaege. Counting Affine Calculator and Applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.

[VGGC14] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule Trees. In *4th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*, page 9, Vienna, Austria, January 2014.

[VJ17] Sven Verdoolaege and Gerda Janssens. Scheduling for ppcg. Report CW 706, Department of Computer Science, KU Leuven, Leuven, Belgium, June 2017.

[VJC+13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for CUDA. *ACM Transactions on Architecture and Code Optimization (TACO)*, January 2013. Selected for presentation at the HiPEAC 2013 Conference.

[VMBL12] Nicolas Vasilache, Benoît Meister, Muthu Baskaran, and Richard Lethin. Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques*, Paris, France, Jan 2012.

[VSB+07] 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, March 2007.

[VSHS14] Anand Venkat, Manu Shantharam, Mary Hall, and Michelle Mills Strout. Non-affine extensions to polyhedral code generation. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 185. ACM, 2014.

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

[Wol86] Michael Wolfe. Loop skewing: The wavefront method revisited. *Int. J. Parallel Program.*, 15(4):279–293, October 1986.

[Wol89] Michael Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *Euro-Par 2012 Parallel Processing*, pages 859–870. Springer, 2012.

[XKH04] Dana N Xu, Siau-Cheng Khoo, and Zhenjiang Hu. Ptype system: A featherweight parallelizability detector. In *Programming Languages and Systems*, pages 197–212. Springer, 2004.

[ZR12] Yun Zou and Sanjay Rajopadhye. Scan detection and parallelization in inherently sequential nested loop programs. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 74–83. ACM, 2012.

RÉSUMÉ

Dans cette thèse, nous étudions l'applicabilité du modèle polyédrique à la compilation de langages spécifiques à un domaine, principalement pour les pipelines de traitement d'images et d'apprentissage en profondeur. Nous identifions quelques limitations des chaînes d'outils de compilation polyédrique existants et proposons des solutions pour les résoudre. Nous étendons une chaîne d'outils polyédrique pour gérer les réductions, qui sont un élément clé des pipelines de traitement d'images et d'apprentissage en profondeur, et deviennent souvent des goulots d'étranglement s'ils ne sont pas parallélisés. Nous étendons également l'algorithme Pluto pour tenir compte de la localité spatiale en plus de la localité temporelle. Nous proposons également une méthode de spécialisation de l'ordonnancement pour des tailles de problèmes données. Nous proposons enfin des techniques permettant de réduire le temps d'autotuning de quelques heures à quelques minutes.

MOTS CLÉS

Parallélisation automatique, compilation de DSL, compilation pour le traitement d'image, compilation pour l'apprentissage en profondeur, génération de code pour GPU

ABSTRACT

In this thesis, we study the applicability of the polyhedral model to the compilation of Domain-specific languages, focussing on image processing and deep learning pipelines. We identify a few limitations of state-of-the-art polyhedral compilation toolchains and propose solutions to address these. We extend a polyhedral toolchain to handle reductions which are a key part of image processing and deep learning pipelines and often become the bottleneck if not parallelized. We also extend the Pluto algorithm to account for the spatial locality in addition to temporal locality. We also propose a schedule specialization method for given problem sizes. Finally, we propose techniques to reduce autotuning time from hours to minutes.

KEYWORDS

Automatic parallelization, DSL compilation, Image-processing compilation, Deep learning compilation, GPU code generation