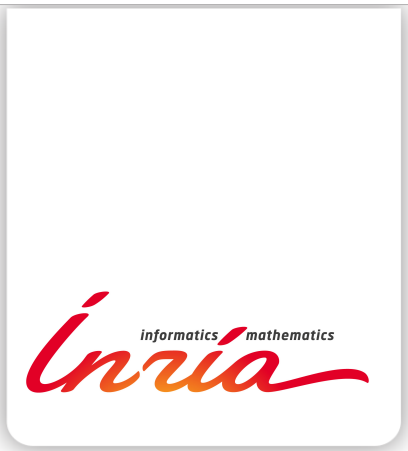
Unified Polyhedral Modeling of Temporal and Spatial Locality

Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, Albert Cohen

RESEARCH REPORT
N° 9110
October 2017
Project-Team Parkas

ISSN 0249-6399 ISRN INRIA/RR--9110--FR+ENG



Unified Polyhedral Modeling of Temporal and Spatial Locality

Oleksandr Zinenko*, Sven Verdoolaege†, Chandan Reddy*‡, Jun Shirako§, Tobias Grosser¶, Vivek Sarkar§, Albert Cohen*

Project-Team Parkas

Research Report n° 9110 — October 2017 — 37 pages

Abstract: Despite decades of work in this area, the construction of effective loop nest optimizers and parallelizers continues to be challenging due to the increasing diversity of both loop-intensive application workloads and complex memory/computation hierarchies in modern processors. The lack of a systematic approach to optimizing locality and parallelism, with a well-founded data locality model, is a major obstacle to the design of optimizing compilers coping with the variety of software and hardware. Acknowledging the conflicting demands on loop nest optimization, 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 parameterizable optimization problems for locality and parallelism objectives over a polyhedral space of semantics-preserving transformations. The overall problem is not convex and is only constrained by semantics preservation. We discuss the rationale for this unified algorithm, and validate it on a collection of representative computational kernels/benchmarks.

Key-words: polyhedral model, loop nest optimization, automatic parallelization

* Inria Paris and DI, École Normale Supérieure
† KU Leuven
‡ PSL Research University
§ Rice Univresity
¶ ETH Zürich

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Unified Polyhedral Modeling of Temporal and Spatial Locality

Résumé : Malgré les décennies de travail dans ce domaine, la construction de compilateurs capables de paralléliser et optimiser les nids de boucle reste un problème difficile, dans le contexte d'une augmentation de la diversité des applications calculatoires et de la complexité de la hiérarchie de calcul et de stockage des processeurs modernes. L'absence d'une méthode systématique pour optimiser la localité et le parallélisme, fondée sur un modèle de localité des données pertinent, constitue un obstacle majeur pour prendre en charge la variété des besoins en optimisation de boucles issus du logiciel et du matériel. Dans ce contexte, nous proposons un nouvel algorithme unifié pour l'optimisation du parallélisme et de la localité dans les nids de boucles, capable de modéliser les effets temporels et spatiaux des multiprocesseurs et accélérateurs comportant des hiérarchies profondes de parallélisme et de mémoire. Cet algorithme coordonne la résolution d'une collection de problèmes d'optimisation paramétrés, portant sur des objectifs de localité ou et de parallélisme, dans un espace polyédrique de transformations préservant la sémantique du programme. La conception de cet algorithme fait l'objet d'une discussion systématique, ainsi que d'une validation expérimentale sur des noyaux calculatoires et benchmarks représentatifs.

Mots-clés : modèle polyédrique, transformations de nids de boucles, parallélisation automatique

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 a 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 Wolfe (1995); Kennedy and Allen (2002). Much of the past work focused on specific transformations, such as fusion Kennedy and McKinley (1993), interchange Allen and Kennedy (1984) or tiling Wolfe (1989); Irigoin and Triolet (1988), or specific objectives, such as parallelization Wolfe (1986) or vectorization Allen and Kennedy (1987).

The *polyhedral framework* of compilation introduced a rigorous formalism for representing and operating on the control flow, data flow, and storage mapping of a growing class of loop-based programs Feautrier and Lengauer (2011). 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 embeddings into general or domain-specific compiler frameworks Pop et al. (2006); Grosser et al. (2012). 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 valid schedule is the primary goal of a polyhedral optimizer. Feautrier's algorithm computes minimum delay schedules Feautrier (1992b) 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 Bondhugula et al. (2008b, 2016). 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.

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 Trifunovic et al. (2009); Vasilache et al. (2012); Kong et al. (2013). 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 those obtained by decoupling locality optimization and parallelization. In addition, these techniques model non-convex optimization problems through the introduction of additional discrete (integer, boolean) variables and of 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 un-

der 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 Shirako et al. (2014). Prior to PolyAST, the pionneering Pluto compiler itself already relied on a heuristic loop permutation to improve spatial locality after the main polyhedral optimization aiming for parallelism and temporal locality Bondhugula et al. (2008b). 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). Our work is based on the contributions of the `isl` scheduler Verdoolaege and Janssens (2017). In particular, we formulate a collection of parameterizable optimization problems, with configurable constraints and objectives, that rely on the scheduler not insisting on the initial schedule rows being linearly independent for lower-dimensional statements in imperfectly nested loops. Our approach to locality-enhancing fusion builds on the "clustering" technique, also introduced in the `isl` scheduler, which allows for a precise intertwining of 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 by using linearly dependent dimensions in affine schedules that are out of reach of a more greedy polyhedral optimizer.

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.

2 Background

The polyhedral framework is based on a linear algebraic representation of the program parts that are "sufficiently regular". It represents arithmetic expressions surrounded by loops and branches with conditions that are affine functions of outer loop iterators and runtime constants Feautrier and Lengauer (2011). 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 driving power for research on loop optimization and parallelization in the last two decades Feautrier (1992b); Bastoul (2004); Bondhugula et al. (2008b); Grosser et al. (2012).

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

```
 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 1: Naive implementation of matrix multiplication.

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 Pugh and Wonnacott (1994). For example, the code fragment in Figure 1 contains two statements, `S` and `R` with iteration domains $\mathcal{D}_{\mathtt{S}}(N) = \{\, \mathtt{S}(i,j) \mid 0 \leq i,j < N \,\}$ and $\mathcal{D}_{\mathtt{R}}(N) = \{\, \mathtt{R}(i,j,k) \mid 0 \leq i,j,k < N \,\}$ respectively. In this paper, we use parametric named relations as also used in `iscc` Verdoolaege (2011); note that the vectors in $\mathcal{D}_{\mathtt{S}}$ and $\mathcal{D}_{\mathtt{R}}$ 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 Feautrier (1992b). The instances are executed following the lexicographic order of their execution dates. This mapping is called a *schedule*, typically defined by piecewise (quasi-)affine functions over the iteration domain $\mathcal{T}_{\mathtt{S}}(\boldsymbol{p}) = \{\, \boldsymbol{i} \rightarrow \boldsymbol{t} \mid \{\, t_j = \phi_{\mathtt{S},j}(\boldsymbol{i},\boldsymbol{p}) \,\} \wedge \boldsymbol{i} \in \mathcal{D}_{\mathtt{S}} \,\}$, which are disjoint unions of affine functions defined on a finite partition of the iteration domain. The use of quasi-affine functions allows integer division by constants. This form of schedule allows 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* Kelly and Pugh (1995) in the schedule or by using a *schedule tree* structure that directly encodes enclosure and statement-level ordering Verdoolaege et al. (2014). For example, the schedule that preserves the original execution order in Figure 1 can be expressed as $\mathcal{T}_{\mathtt{S}}(N) = \{\, \mathtt{S}(i,j) \rightarrow (t_1,t_2,t_3,t_4) \mid t_1 = 0 \wedge t_2 = i \wedge t_3 = j \wedge t_4 = 0 \,\}$, $\mathcal{T}_{\mathtt{R}}(N) = \{\, \mathtt{R}(i,j,k) \rightarrow (t_1,t_2,t_3,t_4) \mid t_1 = 1 \wedge t_2 = i \wedge t_3 = j \wedge t_4 = k \,\}$. 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 Kennedy and Allen (2002). 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 $\mathcal{A}^{\text{write}}_{\mathtt{S}\rightarrow\mathtt{C}} = \{\, \mathtt{S}(i,j) \rightarrow \mathtt{C}(a_1,a_2) \mid a_1 = i \wedge a_2 = j \,\}$. 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 $\mathcal{P}_{\mathtt{S}\rightarrow\mathtt{R}} = \{\, \mathtt{S}(i,j) \rightarrow \mathtt{R}(i',j',k) \mid i = i' \wedge j = j' \wedge (i,j) \in \mathcal{D}_{\mathtt{S}} \wedge (i',j',k) \in \mathcal{D}_{\mathtt{R}} \,\}$. 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 Feautrier (1991). 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`: $\mathcal{P}_{\mathtt{S}\rightarrow\mathtt{R}} = \{\, \mathtt{S}(i,j) \rightarrow \mathtt{R}(i',j',k) \mid i = i' \wedge j = j' \wedge k = 0 \wedge (i,j) \in \mathcal{D}_{\mathtt{S}} \wedge (i',j',k) \in \mathcal{D}_{\mathtt{R}} \,\}$.

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 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 Feautrier (1992b), parallelism Bondhugula et al. (2008b), locality Bondhugula et al. (2008a) 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 $\mathtt{S}(\boldsymbol{i})$ and $\mathtt{R}(\boldsymbol{i'})$. 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*. A 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 Kennedy and Allen (2002).

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 represented directly 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 a *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 Irigoin and Triolet (1988) and are also referred to as *tilable bands*.

2.2 Feautrier's Algorithm

Feautrier's algorithm is one of the first to systematically compute a (quasi-)affine schedule if there exists exits one Feautrier (1992a,b). 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 dependences 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 $e_k \leq \phi_{\mathtt{R}_k,j}(\boldsymbol{i}', \boldsymbol{p}) - \phi_{\mathtt{S}_k,j}(\boldsymbol{i}, \boldsymbol{p})$ for all $\mathtt{S}_k(\boldsymbol{i}) \to \mathtt{R}_k(\boldsymbol{i}')$ in $\mathcal{P}^k \subseteq \mathcal{P}_{\mathtt{S_k} \to \mathtt{R_k}}$, with $\mathcal{P}^k$ a group of dependences between source $\mathtt{S}_k$ and sink $\mathtt{R}_k$. The condition $e_k = 1$ holds iff the entire group of dependences $\mathcal{P}^k$ 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 Feautrier (1988). 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 (groups of) dependences as possible by introducing a penalty for each non-carried group and by minimizing the penalty. 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

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

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.

2.3 Pluto Algorithm

The Pluto algorithm is one of the core parallelization and optimization algorithms Bondhugula et al. (2008b). Multiple extensions have been proposed, including different search spaces Vasilache et al. (2012), specializations and cost functions for GPU Verdoolaege et al. (2013) and proofs of the existence of a solution Bondhugula et al. (2016).

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 Kennedy and Allen (2002). 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 SCCs 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 The affine transformation computed by Pluto is based on the observation that dependence distance $(\phi_{\mathtt{R},j}(\boldsymbol{i}',\boldsymbol{p})-\phi_{\mathtt{S},j}(\boldsymbol{i},\boldsymbol{p}))$ 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, a 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 $(\phi_{\mathtt{R},j}(\boldsymbol{i}',\boldsymbol{p}) - \phi_{\mathtt{S},j}(\boldsymbol{i},\boldsymbol{p})) \leq \boldsymbol{u}\vec{p} + w$, which can be minimized in an ILP problem as

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

where $n_p = \dim \vec{p}$, and $c_{S,k}$ are the coefficients of $\phi_{\mathtt{S},j}$. 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 Bondhugula et al. (2016).

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, the Pluto implementation relies on the *smart fusion* heuristic that separates the DDG into a pair of subgraphs by cutting an edge, hence performing loop *fission*, based on how "far" in terms of original execution time the dependent instances are. Extensions exist to set up an integer programming problem to find auxiliary dimension constant values that maximize fusion between components while keeping the number of required prefetching streams limited Bondhugula et al. (2010a). 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 has been computed. It is applied by inserting a copy of the band's dimensions immediately before the band and modifying them to have a larger stride. The new dimensions correspond to *tile loops* and the original ones now correspond to *point 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 different schedules to be constructed using the following two post-affine modifications. First, a *wavefront* schedule allows parallelism to be exposed 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 the 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 some leverage for 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 $\mathcal{A} \circ \mathcal{T}^{-1}$

$$j : L_j = n_{\mathtt{S}} \cdot \left(\sum_i (2s_i + 4l_i - 16o_i) + 8v \right) \to \max, \tag{2}$$

where n_S is the number of statements for which j-th schedule dimension is an actual loop rather than a constant, $s_i = 1$ if the scheduled access $\mathcal{A}_i \circ \mathcal{T}^{-1}$ features spatial locality and $s_i = 0$ otherwise; $l_i = 1$ if it yields temporal locality 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 $o_i = 0$ for all i.[1] Spatial locality is observed along j-th dimension if it appears in the last array subscript with a small stride and does not appear in previous subscripts: $a_i = f_{\perp t_j}(\boldsymbol{t}) + g(\boldsymbol{u}) + w, 1 \leq i < n_a \wedge a_{n_a} = f_{\perp t_j}(\boldsymbol{t}) + kt_j + g(\boldsymbol{u}) + w, 1 \leq k \leq 4, n_a = \dim(\mathrm{Dom}\,\mathcal{A})$, where $f_{\perp t_j}(\boldsymbol{t})$ denotes a linear function independent of t_j. Temporal locality is observed along j-th dimension if it does not appear in any subscript $a_{n_i} = f_{\perp t_j}(\boldsymbol{t}) + g(\boldsymbol{u}) + w, 1 \leq i \leq n_a$.

The loop `j` with the largest L_j value is put innermost in the band, which corresponds to *loop permutation*. The validity of skewing and permutation is guaranteed by permutability of the band.

Pluto+ Recent work on Pluto+ extends the Pluto algorithm by proving its completeness and termination as well as by enabling negative schedule coefficients Bondhugula et al. (2016) using a slightly different approach than previous work Vasilache et al. (2012); Verdoolaege et al. (2013). Pluto+ imposes limits on the absolute values of the coefficients to simplify the linear independence check and zero solution avoidance.

3 Polyhedral Scheduling in isl

Let us now present a variant of the polyhedral scheduling algorithm, inspired by Pluto and implemented in the `isl` library Verdoolaege (2010). We occasionally refer to the embedding of the scheduling algorithm in a parallelizing compiler called `ppcg` Verdoolaege et al. (2013). We will review the key contributions and differences, highlighting their importance in the construction of a unified model for locality optimization. We will also extend this algorithm in Section 4 to account for the spatial effects model.

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 Verdoolaege et al. (2013).[2] A separate technical report is available for more detailed information about the algorithm and implementation Verdoolaege and Janssens (2017).

3.1 Scheduling Problem Specification in isl

The scheduler we propose offers more control 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;

[1] Reverse-engineered from the Pluto 0.11.4 source code

[2] 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.

- *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. If it is impossible, the loop is considered sequential. The *live range reordering* technique introduces additional *conditional validity* relations in order to remove *false* dependences induced by the reuse of the same variable for different values, when the live ranges of those values do not overlap Verdoolaege and Cohen (2016).

3.2 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. They both use 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 Pouchet et al. (2011). 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, the $(\phi_2 - \phi_1) \leq 0p_1 + 100p_2$ distance bound would be preferred over the $(\phi_2 - \phi_1) \leq p_1 + 0p_2$ 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 positive and negative part $x = x^+ - x^-$, with $x^+ \geq 0$ 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 a 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

$$\operatorname{lexmin} \sum_{i=1}^{n_p} (u_i^- + u_i^+), w, \sum_{i=1}^{n_p} \sum_{j=1}^{n_s} d_{j,i}, \sum_{j=1}^{n_s} \sum_{i=1}^{\dim \mathcal{D}_{S_j}} (c_{j,i}^- + c_{j,i}^+), \dots \tag{3}$$

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, the zero-distance

requirement is relaxed, unless outer parallelism is required by the user and the band is currently empty. 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, then the `isl` scheduler finishes the current schedule band, removing relations that correspond to fully carried dependences and starts a new band. If the current band is empty, then a variant of Feautrier's scheduler Feautrier (1992b) using *validity* and *coincidence* relations as constraints Verdoolaege and Janssens (2017) is applied instead.

3.3 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 Verdoolaege and Janssens (2017). 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 (i.e., linearly dependent) solution 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 the 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+ Bondhugula et al. (2016), 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 Vasilache et al. (2012) and Pluto+ Bondhugula et al. (2016). 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 means that a schedule corresponding to the original loop order will be preferred, unless a better solution can be found.

Although this iterative approach may consider an exponentially large number of sign constraints in the worst case, this 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.

3.4 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 the clusters after checking several profitability heuristics. Cluster combination is essentially loop fusion, except that it allows for rescheduling of individual clusters with respect to each other by *composing* per-statement schedules 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*. 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.

3.5 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* Verdoolaege et al. (2013). In contrast to Pluto, no other transformation is performed at 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. Additionally, accessed data can be copied to the GPU shared memory and registers, see Verdoolaege et al. (2013).

4 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 adjacent 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 the relation is not zero. If a schedule function carries a spatial proximity relation, this results in adjacent statement instances accessing adjacent 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.

4.1 Modeling Line-Based Access

The general feature of the memory hierarchies we model is that *groups* of adjacent 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 consecutive elements. The constant C is used to choose 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 $\mathcal{A}' = \mathcal{C} \circ \mathcal{A}$ where $\mathcal{C} = \{\boldsymbol{a} \to \boldsymbol{a}' \mid a'_{1..(n-1)} = a_{1..(n-1)} \wedge a'_n = \lfloor \frac{a_n}{C} \rfloor\}$, and $n = \dim \vec{a} = \dim(\mathrm{Dom}\, \mathcal{A})$. 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 access 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, this results in a relation 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 $\operatorname{Im}\mathcal{A}'_{\mathtt{S}_i\to\mathtt{A}_j} \leftarrow \operatorname{Im}\mathcal{A}'_{\mathtt{S}_i\to\mathtt{A}_j} \cap \operatorname{Im}\bigcup_k \mathcal{A}_{\mathtt{S}_k\to\mathtt{A}_j}$.

Accesses to scalars, treated as zero-dimensional arrays, are excluded from the 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.

4.2 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 Feautrier (1991), which we adapt to all kinds of dependences rather than just flow dependences. Note that we also consider spatial Read-After-Read (RAR) "dependence" relations as they are an important source of spatial reuse. For all kinds of dependences, only statement instances adjacent in time in the original program are considered to depend on each other. 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 2. Statement `S1` features a spatial RAR relation on B characterized by

$$\mathcal{P}_{S1\to S1,B} = \{(i,j)\to(i',j') \mid (i'=i+1\wedge\lfloor j'/C\rfloor=\lfloor j/C\rfloor)\vee(i'=i\wedge\lfloor j'/C\rfloor=\lfloor j/C\rfloor)\}.$$

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:

$$\mathcal{P}_{S2\to S2,D} = \{(i,j,k)\to(i',j',k') \mid (i'=i\wedge\lfloor k'/C\rfloor=\lfloor j/C\rfloor)\vee(i'=i\wedge\lfloor j'/C\rfloor=\lfloor k/C\rfloor)\}.$$

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.

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 *patterns* is hard to exploit in an affine schedule. Two access relations are considered to have different patterns if there is at least one index expression, excluding the last one, that differs between them. The last index expression is also considered, but without the constant factor. That is,

```
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 2: Non-identical (`S1`) and non-uniform (`S2`) accesses to an array.

`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 1. There exists, among others, a spatial RAR relation between different instances of `R` induced by reuse on `B`:

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

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 the loop trip count is larger than C. In practice, an affine scheduler may generate a schedule with the 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 an $n \times n$ unimodular matrix and H is an $m \times n$ lower triangular matrix, i.e., $h_{ij} = 0$ for $j > i$. Any row-vector $\boldsymbol{v}$ 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^{-1} preserves its completeness. In our example, this 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.

4.3 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 belong to the

same statement as we cannot find a profitable affine schedule for these.

4.4 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 be 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 explained 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 4.5 below. Attempting to force zero distances for the largest possible number of groups with relaxation on failure is combinatorially complex. Instead, we iteratively minimize the distances and only keep the relations for which the distance is zero. The first (following the order of ILP variables) group of spatial proximity relations with a non-zero distance that follows the initial groups with zero distance is removed from further consideration. In other words, the first group that had to be carried is removed. The minimization restarts, and the process continues iteratively until all remaining groups have zero distances. 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

$$\operatorname{lexmin} \sum_{i=1}^{n_p} (u^{\mathrm{T}+}_{1,i} + u^{\mathrm{T}-}_{1,i}), w^{\mathrm{T}}_1, \sum_{i=1}^{n_p} (u^{\mathrm{S}+}_{1,i} + u^{\mathrm{S}-}_{1,i}). \cdots \sum_{i=1}^{n_p} (u^{\mathrm{T}+}_{n_g,i} + u^{\mathrm{T}-}_{n_g,i}), w^{\mathrm{T}}_{n_g}, \sum_{i=1}^{n_p} (u^{\mathrm{S}+}_{n_g,i} + u^{\mathrm{S}-}_{n_g,i}), w^{\mathrm{S}}_{n_g}, \dots \tag{4}$$

where $u^{\mathrm{T}}_{j,i}$ are coefficients of the parameters and w^{T}_j is the constant factor in the distance bound for the j^{th} group of temporal proximity relations, $1 \le j \le n_g$, and $u^{\mathrm{S}}_{j,i}$, w^{S}_j are their counterparts for spatial proximity relations. The remaining non-bound variables are similar to those of (3), namely the sum of schedule coefficients and parameters and individual coefficient values.

4.5 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 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 together, 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. The descending order makes the lexmin objective consider carrying groups with the greatest *rank* and *multiplicity* last.

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 $\mathcal{A} \subseteq (\vec{i} \rightarrow \vec{a})$. Its *rank* is calculated as the number of not yet fixed dimensions. In particular, given the current partial schedule $\mathcal{T} \subseteq (\vec{i} \rightarrow \vec{o})$, we compute the relation between schedule dimensions and access subscripts through composition $\mathcal{A} \circ \mathcal{T}^{-1} \subseteq (\vec{o} \rightarrow \vec{a})$. The number of equations in $\mathcal{A} \circ \mathcal{T}^{-1}$ 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 $\mathcal{A} \circ \mathcal{T}^{-1}$.

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.

4.6 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 3.3 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

$$\text{lexmin} \sum_{i=1}^{\max \dim \mathcal{D}_S} \sum_{j=1}^{n_s} (c^-_{j,i} + c^+_{j,i}), \sum_{k=1}^{n_g} (1 - e_k), \sum_{i=1}^{n_p} \sum_{j=1}^{n_s} d_{j,i}, \dots \tag{5}$$

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.

4.7 Scheduling for CPU Targets

On CPUs, spatial locality is likely to be exploited if the innermost loop accesses adjacent 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 (4) for all dimensions. On the last dimension we apply (5).

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 fewer 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 `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 $h = \sum_{i,k \,:\, \text{aff}\,\mathcal{A}_{S_i \to k} \text{unique}} \dim(\text{Dom}\,\mathcal{A}_{S_i \to k})$ where $\mathcal{A}_{S_i \to k}$ have unique affine hulls across the SCC: $\forall i, j, \forall k \neq l, \text{aff}\,\mathcal{A}_{S_i \to k} \neq \text{aff}\,\mathcal{A}_{S_j \to l}$. 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 $h > h_{\text{lim}}$, 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, `ppcg` has been modified to optionally perform *post-tile loop reordering* using the Pluto heuristic (2).

4.8 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) while enforcing zero distance along *coincidence* constraints. If successful, then the mapping described below will map the coincident dimension to the `x` thread, which can result in spatial locality (even though this is not guaranteed). 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 (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 (3) or Feautrier's ILP.

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

5 Experimental Evaluation

The evaluation consists of 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.

5.1 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 described in Section 4 apply on top of the development versions of both tools (`ppcg-0.07-8-g25faadd`, `isl-0.18-730-gd662836`) available from `git://repo.or.cz/ppcg.git` and `git://repo.or.cz/isl.git`.

Additional Modifications to the `isl` Scheduler Various improvements have been introduced in the development version of `isl`, independently of 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 the 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.

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.

5.2 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 the 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, 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 Bondhugula et al. (2016) 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*: Section 4, with and without *post-tile* reordering; [3]
- *Pluto*: Pluto 0.11.4 with `--parallel --tile` options when appropriate;
- *PolyAST*: with reductions and DoAcross parallelism support disabled.[4]

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.

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.

5.3 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 3(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

[3] Embedded in this report: ppcg-spatial.tar.gz

[4] Reduction is ignored at the parallelization phase and DoAcross is converted into wavefront DoAll.

CLooG, used in Pluto. For `gemm`, Pluto code is slightly more efficient because `ppcg` decided to fuse the initialization and the computation statements, 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.

5.4 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.[5] The speedup of the transformed and parallelized code over the original sequential version is shown in Figure 3(middle).

Similarly to the 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 the `gramschmidt` (8.8× and 2.9× speedup, respectively) benchmark as well as on `nussinov`. Just as for the sequential versions, this difference is caused by `ppcg`'s inability to perform loop fusion after tiling.

Note that the syntactic post-tile reordering transformation is not always beneficial when our algorithm 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] `--target=c --openmp --tile --isl-schedule-single-outer-coincidence --no-isl-schedule-maximize-coincidence --isl-schedule-outer-typed-fusion --isl-schedule-spatial-fusion --isl-schedule-outer-coincidence --isl-schedule-avoid-inner-coincidence --wavefront=single --posttile-reorder=pluto --no-isl-schedule-force-outer-coincidence --spatial-model=groups`

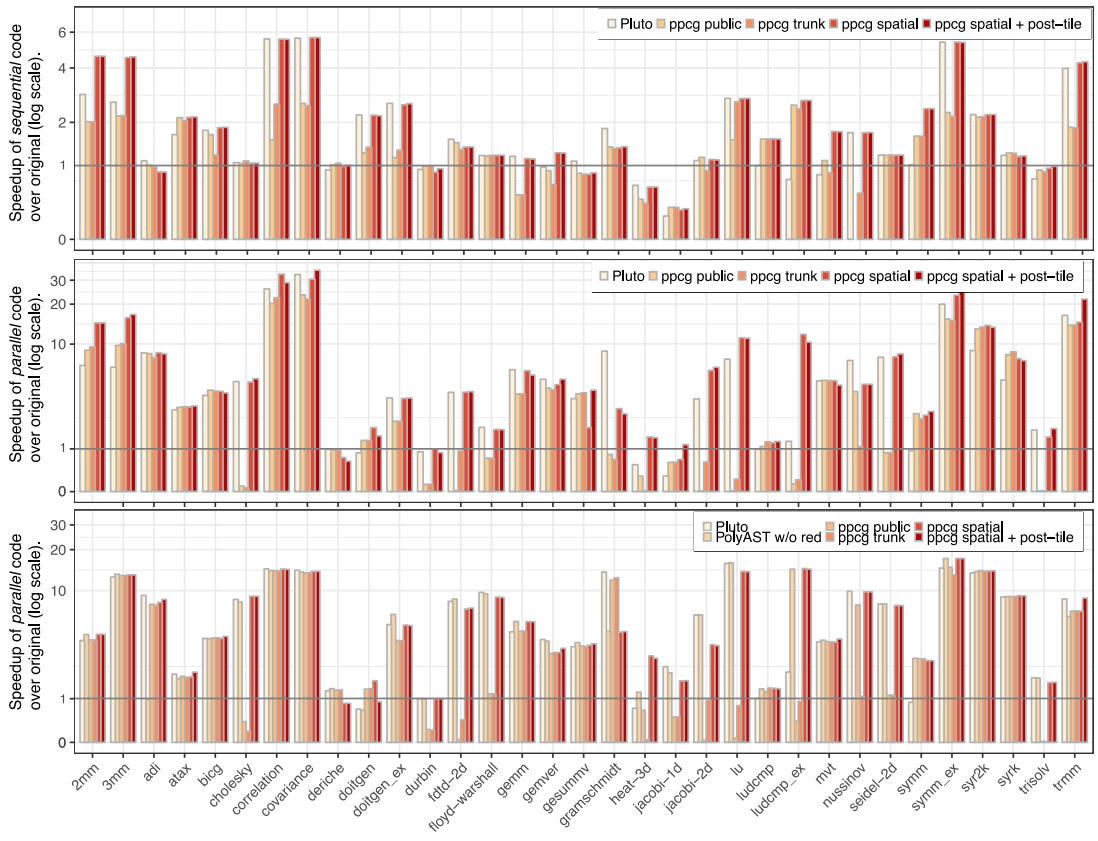


Figure 3: Speedup of the optimized tiled code over the original code with different scheduling algorithms; top: sequential code on **skylake**, middle: parallel code on **ivy**; bottom: parallel code on **westmere**.

5.5 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 3(bottom). As PolyAST was designed to yield efficient code for `icc`, we used this compiler across all tools. Pluto was run with the `--noprevector` flag to disable `icc`-specific pragmas it is able to generate 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 reverse, `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 the `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.

5.6 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 4. For all cases except `lu`, `ppcg` discovers no outer parallelism and resorts to repeated kernel calls from the generated host code. Figure 5 summarizes the number of different kernels and the cumulative number of kernel invocations. In such cases, an important benefit of our approach lies in 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, the 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

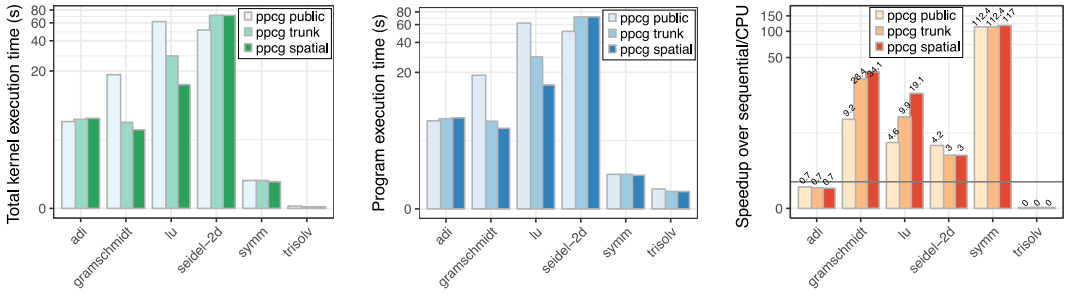


Figure 4: Left and center graphs show total kernel execution time and program execution time (lower is better). Right graph shows speedup over CPU sequential version (higher is better).

	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: 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).

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 onto 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 loops, 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.

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, the spatial effects-aware scheduler 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.

6 Differences in Schedules: Case Study Discussions

6.1 Two Matrix Multiplications

`2mm` is a linear algebra kernel that computes an $I \times L$ matrix $D = \beta C \cdot \alpha(A \cdot B)$ where A is an $I \times K$, B is a $K \times J$, and C is a $J \times L$ matrix; α, β are scalars as shown in Fig 6.

The schedule computed by our algorithm for the OpenMP target is

$$\begin{aligned}&\{\, \texttt{S1}(i,j) \to (0,i,0,j) \,\} \cup\\ &\{\, \texttt{S2}(i,j,k) \to (0,i,k,j) \,\} \cup\\ &\{\, \texttt{S3}(i,j) \to (1,i,0,j) \,\} \cup\\ &\{\, \texttt{S4}(i,j,k) \to (1,i,k,j) \,\}\end{aligned}$$

whereas Pluto proposes

$$\begin{aligned}&\{\, \texttt{S1}(i,j) \to (0,i,j,1,0) \,\} \cup\\ &\{\, \texttt{S2}(i,j,k) \to (1,i,j,0,k) \,\} \cup\\ &\{\, \texttt{S3}(i,j) \to (0,i,j,0,0) \,\} \cup\\ &\{\, \texttt{S4}(i,j,k) \to (1,i,k,1,j) \,\},\end{aligned}$$

`ppcg`-trunk proposes

$$\begin{aligned}&\{\, \texttt{S1}(i,j) \to (0,i,j,0) \,\} \cup\\ &\{\, \texttt{S2}(i,j,k) \to (0,i,j,k) \,\} \cup\\ &\{\, \texttt{S3}(i,j) \to (1,i,j,0) \,\} \cup\\ &\{\, \texttt{S4}(i,j,k) \to (1,i,j,k) \,\}\end{aligned}$$

and PolyAST proposes

$$\begin{aligned}&\{\, \texttt{S1}(i,j) \to (i,0,j) \,\} \cup\\ &\{\, \texttt{S2}(i,j,k) \to (i,1,k,j) \,\} \cup\\ &\{\, \texttt{S3}(i,j) \to (i,2,j) \,\} \cup\\ &\{\, \texttt{S4}(i,j,k) \to (i,3,k,j) \,\}.\end{aligned}$$

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 Ferrante et al. (1991); Sarkar (1997) 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 the 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

```
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];
   }
}
```

Figure 6: Code of the `2mm` benchmark with labeled statements.

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: the `i` and `j` loops are outer parallel and are mapped to `y` and `x` blocks, respectively, the original fusion structure is maintained. However, without spatial effects modeling, the `ppcg` cost function cannot distinguish between the `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 a 1.5× slowdown in kernel execution time and a 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. A 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.

6.2 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 7, which is challenging for analysis due to the large number of non-uniform dependences.

Both Pluto and our algorithm resort to *wavefront* parallelism after tiling. For the OpenMP version, Pluto proposes the schedule

$$\{\, \mathtt{S1}(i,j,k) \to (i,j,k) \,\} \cup \{\, \mathtt{S2}(i,j) \to (i,j,j) \,\} \cup \{\, \mathtt{S3}(i,j,k) \to (i,j,k) \,\},$$

which essentially embeds `S2` in the innermost loop so as to respect dependences. After tiling, it interchanges the two innermost *point* loops to leverage spatial locality but keeps the tile loop order unchanged. Our algorithm computes directly the schedule

$$\{\, \mathtt{S1}(i,j,k) \to (i,k,j) \,\} \cup \{\, \mathtt{S2}(i,j) \to (i,j,j) \,\} \cup \{\, \mathtt{S3}(i,j,k) \to (i,k,j) \,\}$$

```
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 7: Code of the `lu` benchmark with labeled statements.

including this interchange. Using this schedule for *both tile* and *point* loops improves sequential performance thanks to avoiding false sharing effects. The public `ppcg` does not have support for wavefront parallelism and does not parallelize this kernel. PolyAST proposes a 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 is in no danger of missing doall parallelism discussed in Section 5.5.

For GPU targets, `ppcg`-trunk proposes the schedule

$$\{\,\texttt{S1}(i,j,k) \to (k,0,i,j)\,\} \cup \{\,\texttt{S2}(i,j) \to (j,1,i,j)\,\} \cup \{\,\texttt{S3}(i,j,k) \to (k,2,i,j)\,\},$$

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

$$\{\,\texttt{S1}(i,j,k) \to (k,1,j,i)\,\} \cup \{\,\texttt{S2}(i,j) \to (j,0,j,i)\,\} \cup \{\,\texttt{S3}(i,j,k) \to (k,1,j,i)\,\},$$

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.

7 Discussion and Future Work

Before summarizing our findings, let us discuss some of the algorithmic design choices hinting at possible alternatives and extensions.

Filtering Spatial Proximity Relations Defining the spatial proximity relations, we filter out some (non-uniform, single-statement) relations that we deemed unexploitable by the affine scheduler. Yet these relations encode spatial reuse information that might have been useful, e.g., to a fusion heuristic. Similarly, they may help data layout transformations to improve locality, which is not supported in `ppcg`.

Dependence Analysis to Extract Spatial Proximity Relations Temporal and spatial proximity relations result from a typical dependence analysis, pruning transitively closed dependences. As a result, the proximity relations will only capture statement instances that have some

spatial proximity *in the original program*; it may eliminate a read-after-read relation transitively covered by other relations. This allows each relation to be associates with a constant access stride.[6] It would be possible to preserve the full relations along the steps of the scheduling algorithm, pruning only locally when computing access strides, but this would severely damage algorithmic complexity and we only observed three cases where the schedule is impacted, with no significant performance difference.

Ordering Access Groups Our approach reorders access groups before each ILP to prioritize those groups that can still feature some locality given the current schedule. Lexicographical minimization does not guarantee that the maximum number of access groups will be optimized for locality. We see this ordering as a possibility for tweaking the behavior of the algorithm without modifying the ILP formulation itself. For example, our implementation puts spatial proximity distance bounds immediately after temporal proximity bounds for the same access group. This order may be changed to, e.g., propose a different trade-off between exploiting spatial locality for multiple groups and temporal locality for one group, or to prioritize locality optimization for certain groups based on an external heuristic. A weighted cost function would be preferable to ordering, yet it is difficult to propose one without limiting the possible reuse distances and thus the schedule coefficients.

Limited Schedule Space Multiple polyhedral approaches limit the absolute value of the schedule coefficients to derive bounds on reuse distances. Their main argument in favor of such limitation is that schedules with large coefficients do not yield good performance anyway. Our approach does not have such prerequisite, but makes it possible to bound schedule coefficients if necessary. However, we prefer to avoid situations where large coefficients are considered profitable by the objective function instead of restricting them.

Reducing Register Pressure Register pressure turned out to be one of the performance bottlenecks—on both CPU and GPU—even though our benchmarks remain relatively small. We proposed a simple heuristic to choose between two alternative ILPs and to leverage their side effects. This heuristic is controlled by a numerical parameter that may be tuned to a particular loop nest, as our results show that a single value does not fit all inputs. Multiple approaches can be used to reduce register pressure, such as data layout transformations Henretty et al. (2011), exploiting associativity Stock et al. (2013), careful selection of tile shapes and sizes Grosser et al. (2014), and more conventional copy propagation and live range splitting Kennedy and Allen (2002). These methods are complementary to affine scheduling.

Clustering Heuristics The clustering heuristics aim at preserving a sufficient amount of parallelism and at enabling loop fusion when it actually improves locality. At the same time, multiple other effects may be at play, including register pressure as discussed above, the availability of prefetching streams or the complexity of the generated code. For larger programs, more localized heuristics may refine the fusion decisions or adaptively prioritize the connected components to cluster.

Code Generation-Aware Scheduling Although the polyhedral model traditionally separates scheduling from code generation, it may be necessary to consider some aspects of the code generation algorithm in the scheduler. For example, the code generator may choose to distribute non-overlapping parts of the fused loops or to coalesce them in the same loop with specific guard

[6] The transitive closure of a uniform relation typically has variable and parametrically bounded distance.

conditions. In fact, our choices to filter spatial proximity relations is based on the observation of inefficient banching control flow in the generated code. Our evaluation uncovered several cases where the schedule's objective function improved, along with data locality, yet the generated code was less efficient.

Post-Tile Reordering As exemplified in the previous section, some benchmarks may require additional transformation after loop tiling. However, part of the affine scheduling objectives is to maximize the depth of tilable bands, making it inapplicable before tiling. Hence a two-phase approach may be required to produce profitable schedules in this case. The second phase should be performed after tiling and apply a full-fledged affine scheduler while preserving the band structure. Such an approach will provide a more robust alternative to post-tile heuristics for locality and wavefront parallelization, and allow for simultaneous fusion and rescheduling after tiling, the absence of which negatively impacts `ppcg`-generated code in several cases. The second phase may also be applied after the program is mapped to an accelerator to schedule memory copy operations across different memory spaces.

8 Related Work

One of the first algorithms to handle trade-offs between parallelism and data locality was proposed by Kennedy and McKinley (1992). It collects array references into groups when they access the same cache line and defines a cost-based heuristic for loop permutations. Simultaneous optimization and parallelization is achieved through strip-mining. This algorithm predates the wider adoption of the polyhedral model and focuses on performing individual loop transformations rather than rescheduling the entire static control part.

Within the polyhedral framework, automatic scheduling has been the subject of active research over the past three decades. The Feautrier (1992a) algorithm tends to favor fine-grained parallelism by forcing the outermost loops to carry the maximum number of dependences. The Lim and Lam (1997) algorithm aims to minimize synchronizations, hence maximizing coarse-grain parallelism. More recent work by Vasilache (2007) explored the convex modeling of the exact space of all valid multi-dimensional schedules. Vasilache et al. (2012) proposed contiguity constraints to capture innermost reuse along one dimension for a given array reference; contiguity is enforced by restricting the schedule such that, after substituting the schedule iterators (the so-called time dimensions) into array subscripts, the innermost transformed loop iterator appears in only a single index expression. A simpler version was proposed by Bastoul and Feautrier (2003) for the innermost memory dimension only. Vasilache et al. (2012) generalize this to any memory dimension, showing how to integrate these constraints with ILP-based affine scheduling, and tying together the contiguity constraints of all array references within a statement to yield vectorization constraints for a given dimension. Other approaches focused on heuristics for (generalized forms of) loop fusion in a polyhedral framework, modeling temporal locality Bastoul and Feautrier (2005); Bondhugula et al. (2010b); Bastoul (2016) and introducing criteria similar to those of our clustering method Lim et al. (2001).

In contrast, we use relations mimicking dependences to model spatial locality and try to minimize reuse distances, and unlike Vasilache et al., our algorithm also attempts at minimizing the stride along the innermost dimension when strict contiguity cannot be achieved. Trifunovic et al. (2009) devise a polyhedral scheduling strategy and a cost model to support automatic vectorization, but considers combinations of loop permutations only. Kong et al. (2013) state the profitability of affine scheduling for the vectorizability of point loops as an ILP. Their encoding of stride-0/1 constraints uses original loop iterators, unlike Vasilache et al. (2012) and vectorization

involves a powerful domain-specific SIMD code generator. The Pluto framework Bondhugula et al. (2008b) does not model spatial locality when solving ILPs to find permutable bands, but resorts to a post scheduling step called intra-tile-opt to reorder point loops according to spatial effects, whereas in our approach spatial effects are combined with temporal ones while solving ILP problems and influence both tile and point loops. Minimizing spatial locality dependences in the outer parallel loops helps reducing false sharing among threads, hence our scheduler often outperforms Pluto even when the point loop schedule is identical.

Syntactic loop transformations for locality have a long history Abu-Sufah et al. (1979); Wolfe (1989, 1995); Kennedy and Allen (2002). In particular, loop fusion heuristics were initially designed as locality-enhancing optimizations in isolation from other loop nest transformations Kennedy and McKinley (1993); Megiddo and Sarkar (1997); Singhai and McKinley (1997); in this regard, several polyhedral generalizations of loop fusion are reminiscent of typed fusion Kennedy and McKinley (1993), including our clustering approach. Syntactic methods apply a sequence of individual loop transformations including permutation, fusion, distribution, reversal, and tiling, driven by analytical cost models McKinley et al. (1996); Wolf et al. (1996); Sarkar (1997). As an example, the Distinct Lines (DL) model was designed to improve data locality in cache and TLB Ferrante et al. (1991), guiding locality optimizations in the IBM ASTI optimizer Sarkar (1997). As a hybrid framework combining polyhedral and syntactic transformations, PolyAST Shirako et al. (2014) employs a two-stage approach. The polyhedral first stage focuses on affine scheduling to aim for good temporal and spatial locality, guided by the DL cost model Ferrante et al. (1991); Sarkar (1997). The syntactic second stage attempts to detect outermost forall, reduction, or doacross loop parallelism, based on syntactic information regarding commutativity and associativity and on dependence information from the polyhedral stage. Designing cost models and objective functions is simpler in these syntactic and hybrid approaches, since optimization problems are phased more incrementally and semantics preservation is typically handled separately; however, a globally optimal solution may be missed due to a priori restrictions on the space of transformations, or to the overly eager selection of profitable ones.

9 Conclusion

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 parameterizable 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 depedendent 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.

Our results restore some hope in the construction of simpler, elegant and more performance-portable loop nest optimizers and parallelizers. We also believe these approaches will be widely applicable in domain- and target-specific optimization problems, mapping high-level equations to hardware accelerators of all kind. This includes numerical simulations as well as machine learning algorithms on both dense and sparse data structures, targeting manycore and reconfigurable

hardware.

References

Abu-Sufah, W., Kuck, D., and Lawrie, D. (1979). Automatic program transformations for virtual memory computers. In *Proc. of the 1979 National Computer Conference*, pages 969–974.

Allen, J. R. and Kennedy, K. (1984). Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '84, pages 233–246, New York, NY, USA. ACM.

Allen, R. and Kennedy, K. (1987). Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9(4):491–542.

Bastoul, C. (2004). 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. IEEE Computer Society.

Bastoul, C. (2016). Mapping deviation: A technique to adapt or to guard loop transformation intuitions for legality. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 229–239, New York, NY, USA. ACM.

Bastoul, C. and Feautrier, P. (2003). Improving Data Locality by Chunking. In Hedin, G., editor, *Compiler Construction*, number 2622 in Lecture Notes in Computer Science, pages 320–334. Springer Berlin Heidelberg.

Bastoul, C. and Feautrier, P. (2005). Adjusting a program transformation for legality. *Parallel processing letters*, 15(01n02):3–17.

Bondhugula, U., Acharya, A., and Cohen, A. (2016). 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.

Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., and Sadayappan, P. (2008a). Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In *Compiler Construction*, pages 132–146, Budapest, Hungary. Springer.

Bondhugula, U., Gunluk, O., Dash, S., and Renganarayanan, L. (2010a). 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. ACM. IBM XL.

Bondhugula, U., Günlük, O., Dash, S., and Renganarayanan, L. (2010b). A model for fusion and code motion in an automatic parallelizing compiler. In *19th International Conference on Parallel Architecture and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010*, pages 343–352.

Bondhugula, U., Hartono, A., Ramanujam, J., and Sadayappan, P. (2008b). A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *ACM SIGPLAN Notices*, 43(6):101–113.

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

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

Feautrier, P. (1992a). Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time. 21(5):313–347.

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

Feautrier, P. and Lengauer, C. (2011). Polyhedron Model. In Padua, D., editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer US.

Ferrante, J., Sarkar, V., and Thrash, W. (1991). On Estimating and Enhancing Cache Effectiveness. In *Languages and Compilers for Parallel Computing*, pages 328–343. Springer Berlin Heidelberg.

Grosser, T., Cohen, A., Holewinski, J., Sadayappan, P., and Verdoolaege, S. (2014). Hybrid Hexagonal/Classical Tiling for GPUs. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 66. ACM.

Grosser, T., Groesslinger, A., and Lengauer, C. (2012). Polly — Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04):1250010.

Henretty, T., Stock, K., Pouchet, L.-N., Franchetti, F., Ramanujam, J., and Sadayappan, P. (2011). Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction*, pages 225–245. Springer Berlin/Heidelberg.

Irigoin, F. and Triolet, R. (1988). 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. ACM.

Kelly, W. and Pugh, W. (1995). 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.

Kennedy, K. and Allen, J. R. (2002). *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

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

Kennedy, K. and McKinley, K. S. (1992). Optimizing for Parallelism and Data Locality. In *Proceedings of the 6th International Conference on Supercomputing*, ICS '92, pages 323–334, New York, NY, USA. ACM.

Kong, M., Veras, R., Stock, K., Franchetti, F., Pouchet, L.-N., and Sadayappan, P. (2013). When polyhedral transformations meet simd code generation. In *ACM SIGPLAN Notices*, volume 48, pages 127–138. ACM.

Lim, A. W. and Lam, M. S. (1997). Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA. ACM.

Lim, A. W., Liao, S.-W., and Lam, M. S. (2001). Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 103–112, New York, NY, USA. ACM.

McKinley, K. S., Carr, S., and Tseng, C.-W. (1996). Improving Data Locality with Loop Transformations. 18(4):424–453.

Megiddo, N. and Sarkar, V. (1997). Optimal weighted loop fusion for parallel programs. In *symposium on Parallel Algorithms and Architectures*, pages 282–291.

Pop, S., Cohen, A., Bastoul, C., Girbal, S., Silber, G.-A., and Vasilache, N. (2006). GRAPHITE: Polyhedral Analyses and Optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*, pages 179–197.

Pouchet, L.-N., Bondhugula, U., Bastoul, C., Cohen, A., Ramanujam, J., Sadayappan, P., and Vasilache, N. (2011). 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. ACM.

Pugh, W. and Wonnacott, D. (1994). Static Analysis of Upper and Lower Bounds on Dependences and Parallelism. *ACM Trans. Program. Lang. Syst.*, 16(4):1248–1278.

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

Shirako, J., Pouchet, L. N., and Sarkar, V. (2014). 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.

Singhai, S. and McKinley, K. (1997). A Parameterized Loop Fusion Algorithm for Improving Parallelism and Cache Locality. *The Computer Journal*, 40(6):340–355.

Stock, K., Kong, M., Grosser, T., Pouchet, L.-N., Rastello, F., Ramanujam, J., and Sadayappan, P. (2013). A Framework for Enhancing Data Reuse via Associative Reordering. pages 65–76. ACM Press.

Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., and Rosen, I. (2009). Polyhedral-model guided loop-nest auto-vectorization. In *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*, pages 327–337. IEEE.

Vasilache, N. (2007). Scalable program optimization techniques in the polyhedral model. *These de doctorat, Université de Paris-Sud*, 11.

Vasilache, N., Meister, B., Baskaran, M., and Lethin, R. (2012). Joint scheduling and layout optimization to enable multi-level vectorization. In *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques*, Paris, France.

Verdoolaege, S. (2010). Isl: An Integer Set Library for the Polyhedral Model. In Fukuda, K., van der Hoeven, J., Joswig, M., and Takayama, N., editors, *Mathematical Software – ICMS 2010*, number 6327 in Lecture Notes in Computer Science, pages 299–302. Springer Berlin Heidelberg.

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

Verdoolaege, S., Carlos Juega, J., Cohen, A., Ignacio Gómez, J., Tenllado, C., and Catthoor, F. (2013). Polyhedral Parallel Code Generation for CUDA. 9(4):54:1–54:23.

Verdoolaege, S. and Cohen, A. (2016). Live Range Reordering. In *6th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*, Prague, Czech Republic.

Verdoolaege, S., Guelton, S., Grosser, T., and Cohen, A. (2014). Schedule Trees. In *4th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*, page 9, Vienna, Austria.

Verdoolaege, S. and Janssens, G. (2017). Scheduling for ppcg. Report CW 706, Department of Computer Science, KU Leuven, Leuven, Belgium.

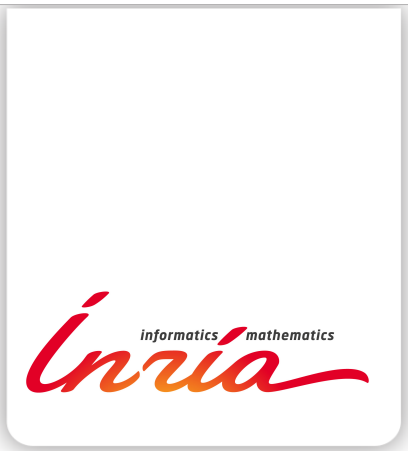
Wolf, M., Maydan, D., and Chen, D.-K. (1996). Combining loop transformations considering caches and scheduling. In *MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 274–286.

Wolfe, M. (1986). Loop skewing: The wavefront method revisited. *Int. J. Parallel Program.*, 15(4):279–293.

Wolfe, M. (1989). 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. Society for Industrial and Applied Mathematics.

Wolfe, M. J. (1995). *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Contents



RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399