Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling

Oleksandr Zinenko
Inria & DI ENS
Paris, France
oleksandr.zinenko@inria.fr

Sven Verdoolaege
Polly Labs & KU Leuven
Leuven, Belgium
sven@cs.kuleuven.be

Chandan Reddy
Inria & DI ENS
Paris, France
chandan.reddy@inria.fr

Jun Shirako
Georgia Tech
Atlanta, Georgia, US
shirako@gatech.edu

Tobias Grosser
ETH Zürich
Zürich, Switzerland
tobias.grosser@inf.ethz.ch

Vivek Sarkar
Georgia Tech
Atlanta, Georgia, US
vsarkar@gatech.edu

Albert Cohen
Inria & DI ENS
Paris, France
albert.cohen@inria.fr

Abstract

The construction of effective loop nest optimizers and parallelizers remains challenging despite decades of work in the area. Due to the increasing diversity of loop-intensive applications and to the complex memory/computation hierarchies in modern processors, optimization heuristics are pulled towards conflicting goals, highlighting the lack of a systematic approach to optimizing locality and parallelism. Acknowledging these conflicting demands on loop nest optimization, we propose an algorithmic template capable of modeling the multi-level parallelism and the temporal/spatial locality of multiprocessors and accelerators. This algorithmic template orchestrates a collection of parameterizable, linear optimization problems over a polyhedral space of semantics-preserving transformations. While the overall problem is not convex, effective algorithms can be derived from this template delivering unprecedented performance portability over GPU and multicore CPU. We discuss the rationale for this algorithmic template and validate it on representative computational kernels/benchmarks.

CCS Concepts • **Software and its engineering → Compilers**;

Keywords Polyhedral Model, Compiler Optimizations

ACM Reference Format:
Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179507

CC'18, February 24–25, 2018, Vienna, Austria

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 the performance offered by such systems is a task of ever growing difficulty. Optimizing compilers transform a high-level, easy-to-read program into more complex but efficient, target-specific code. Performance portability requires modeling architectural effects that do not fit a convex optimization problem, and may require conflicting transformations. In this context, the systematic exploration of the space of semantics-preserving transformations remains a primary challenge in compiler construction.

Ten years ago, the Pluto algorithm made a significant contribution to the theory and practice of affine scheduling for locality and parallelism [7]. It is rooted in the *polyhedral framework* of compilation, a rigorous formalism to represent and operate on the control and data flow of a growing class of loop-based programs [11]. It provides a unified approach to loop nest optimization, offering precise analyses, aggressive transformations and code generation. The past decade saw the emergence of robust and scalable implementations and integration of polyhedral techniques into general-purpose compilers [4, 12, 22]. However, modern processor architectures made it imperative to model deep memory hierarchies that favor consecutive accesses to improve performance. Our work revisits the design of Pluto in light of these architectural features. We build a model of these features suitable for affine scheduling with heuristics based on linear programming, leveraging positive effects (e.g., locality) and avoiding the negative ones (e.g., false sharing). Rather than a unified algorithm, we propose a template built upon a parameterizable scheduling problem and a pair of interchangeable optimization objectives. In particular, we contribute a "clustering" technique for loop fusion, intertwining the iterations of different statements while maintaining the execution order within each loop, and we extend the loop sinking options when aligning imperfectly nested loops to the same depth. We address spatial effects by extending

the optimization objective and allowing for linearly dependent dimensions in affine schedules that are out of reach of a typical polyhedral optimizer. Our iterative approach to non-convex optimization does not restrict the optimization space and is particularly effective when negative coefficients are necessary to tile iteration spaces while aligning dimensions with the direction of consecutive memory accesses.

We evaluate two target-specific instances of our template, modeling complex loop nest transformations as a single affine schedule, where the state of the art required a combination of polyhedral and syntactic transformations [26].

2 Background

The polyhedral framework is a linear algebraic representation of the program parts that are "sufficiently regular". It may represent imperative statements surrounded by loops and branches whose conditions are affine functions of outer loop iterators and runtime constants [11]. 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.

The individual executions of statements inside loops, or *statement instances*, are identified by a named multidimensional vector, where the name identifies the statement and the coordinates correspond to iteration variables of the surrounding loops. The set of all named vectors is called the *iteration domain* of the statement, and can be expressed using Presburger formulas [24]. For example, a R surrounded by three loops i, j, j all iterating from 0 to N has the domain $\mathcal{D}_{\mathsf{R}}(N) = \{\mathsf{R}(i,j,k) \mid 0 \le i,j,k < N\}$. We use parametric named relations as proposed in `iscc` [31]; note that set vectors in are prefixed with the statement name. Unless otherwise specified, we assume all values to be integer, $i, j, \dots \in \mathbb{Z}$.

Polyhedral modeling of the control flow maps statement instances to multidimensional logical execution dates [10]. The instances are executed following the lexicographic order of their execution dates. This mapping is called a *schedule*, a piecewise (quasi-)affine function over the iteration domain $\mathcal{T}_{\mathsf{S}}(\boldsymbol{p}) = \{\boldsymbol{i} \to \boldsymbol{t} \mid \{t_j = \phi_{\mathsf{S},j}(\boldsymbol{i},\boldsymbol{p})\} \wedge \boldsymbol{i} \in \mathcal{D}_{\mathsf{S}}\}$, which are disjoint unions of affine functions defined on a finite partition of the iteration domain, allowing integer division by constants. They capture arbitrary loop traversals and interleavings of statement instances. In this paper, $\boldsymbol{x}$ denotes a row vector and $\vec{x}$ denotes a column vector.

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 [15]. 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 $\mathcal{A}^{\text{write}}_{\mathsf{S}\to\mathsf{C}} = \{\mathsf{S}(i,j) \to \mathsf{C}(a_1,a_2) \mid a_1 = i \wedge a_2 = j\}$. Then, in *memory-based* dependence analysis, pairs of statement instances accessing the same array element where at least one access is a write combined to define a *dependence relation*. For example, the dependence between statements S and R is defined by a relation $\mathcal{P}_{\mathsf{S}\to\mathsf{R}} = \{\mathsf{S}(i,j) \to \mathsf{R}(i',j',k) \mid i = i' \wedge j = j' \wedge (i,j) \in \mathcal{D}_{\mathsf{S}} \wedge (i',j',k) \in \mathcal{D}_{\mathsf{R}}\}$. From this relation, one may compute exact *data flow* given a schedule using *value-based* dependence analysis [9].

A dependence relation is *satisfied* by a schedule if all the statement instances in its domain are scheduled before their counterparts in its range. To transform a program in the polyhedral framework, one defines a new schedule. A program transformation is *valid*, i.e., preserves original program semantics, if all dependences are satisfied. Optimization algorithms navigate the set of valid schedules, optimizing for latency [10], parallelism [7] or locality [5].

3 Polyhedral Scheduling in isl

We present a template for polyhedral scheduling algorithms, inspired by Pluto [5] and implemented in the `isl` library [30].[1] We occasionally refer to the embedding of the scheduling algorithm in a parallelizing compiler called `ppcg` [32]. Let us first present the algorithmic template and discuss key contributions before the extension for spatial locality in Section 4.

3.1 Scheduling Problem Formulation in isl

Our scheduler offers more control through different groups of relations suitable for specific optimization purposes: (1) *validity relations* impose a partial execution order on statement instances, i.e., they are dependences sufficient to preserve program semantics; (2) *proximity relations* connect statement instances that should be executed as close to each other as possible in time; (3) *coincidence relations* connect statement instances that, if not executed at the same time, prevent parallel execution. In the simplest case, all relations are the same and come directly from dependence analysis; *live range reordering* uses different relations to remove *false* dependences due to reusing the same variables for different values [33].

The scheduler iteratively determines sequences of statement-wise 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.

Consider the affine form $(\phi_{\mathsf{R},j}(\boldsymbol{i},\boldsymbol{p}) - \phi_{\mathsf{S},j}(\boldsymbol{i},\boldsymbol{p}))$, defined for a dependence between sources S and sinks R. This form represents the *distance* between dependent statement instances. Positive distance means the dependence is *strongly satisfied* (*carried*), zero distance— *weakly satisfied* and negative distance—*violated*. Using the affine form of Farkas's lemma—a fundamental result in linear algebra that states that an affine form $\boldsymbol{c}\vec{x} + d$ is non-negative everywhere in the (non-empty) set defined by $A\vec{x} + \vec{b} \ge 0$ if it is a linear

[1] Many of these features have been available since `isl-0.06-43-g1192654`, but 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.

combination $\boldsymbol{c}\vec{x} + d \equiv \lambda_0 + \boldsymbol{\lambda}(A\vec{x} + \vec{b})$, where $\lambda_0, \boldsymbol{\lambda} \geq 0$—to dependence distance relations, one can obtain *constraints* on schedule coefficients $\boldsymbol{c}_j$ under which the dependences have non-negative distance, i.e., are weakly satisfied over the iteration domain. One can then apply integer linear programming (ILP) to optimize a linear objective function over the constrained space of schedule coefficients.

3.2 Affine Transformations

Affine transformation is based on the observation, made in Pluto [5], that dependences distances are also *reuse* distances. Hence minimizing them may improve locality. Zero distances imply that all accesses are performed within the same iteration and thus parallelization is possible. An upper bound on the dependence distance $(\phi_{\mathsf{R},j}(\boldsymbol{i},\boldsymbol{p}) - \phi_{\mathsf{S},j}(\boldsymbol{i},\boldsymbol{p})) \leq \boldsymbol{u}\vec{p} + w$ can be obtained using Farkas' lemma and used in a minimization objective of an ILP problem. The bound may involve negative coefficients without necessarily being negative itself. Schedule coefficients may also become negative, driving the minimization to negative infinity; at the same time, all-zero coefficients would not constitute a loop. In practice, we want to obtain their minimum non-zero *absolute value*.

Negative Coefficients `isl` introduces support for negative coefficients by substituting dimension x with its negative and positive part $x = x^+ - x^-$, where $x^+, x^- \geq 0$ in a non-negative optimization problem. This decomposition is performed for schedule coefficients c and bound coefficients u.

Prefix Dimensions To minimize multiple values simultaneously, isl scheduler uses a special lexmin objective, proposed in PIP tool and resulting in the lexicographically smallest vector of the search space [8]. Intuitively, it minimizes the foremost component before moving to the next one. Such behavior may be undesirable for schedule coefficients as it will prefer (a_1, a_2) over (b_1, b_2) if $a_1 < b_1$ even though $a_2 \gg b_2$, yet large coefficients yield worse performance [23]. Therefore, `isl` introduces as leading components (1) sum of all parameter coefficients in the distance bound; (2) constant term of the distance bound; (3) sum of all parameter coefficients in all per-statement schedule functions; (4) 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.

ILP Formulation The `isl` scheduler optimizes

$$\text{lexmin} \sum_{i=1}^{n_p}(u_i^- + u_i^+), w, \sum_{i=1}^{n_p}\sum_{j=1}^{n_s} d_{j,i}, \sum_{j=1}^{n_s}\sum_{i=1}^{\dim \mathcal{D}_{S_j}} (c_{j,i}^- + c_{j,i}^+) \ldots \quad (1)$$

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, this requirement is relaxed. If the problem remains unsolvable, `isl` performs band splitting as described in the following subsection.

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

3.3 Ensuring Progress and Flexibility

Iteratively optimizing the same function over the same space produces the same result, which would not prevent the scheduler from progressing. Therefore, for each subsequent schedule function, isl further constrains the schedule coefficients so that a vector thereof is linearly independent from the previous ones. We refer to linearly-dependent (and zero) ILP solution vectors as *trivial*.

Lazy Enforcement of Linear Independence `isl` computes a subspace with a basis $\boldsymbol{r}_k$ orthogonal to the vectors of already computed schedule coefficients. For another vector to be linearly independent from previous ones, it is sufficient to have a non-zero component along one of $\boldsymbol{r}_k$.

`isl` tries to find a solution $\boldsymbol{x}$ directly and only enforces non-triviality if an actual trivial solution was found. More specifically, it defines *non-triviality regions* in the solution vector $\boldsymbol{x}$ that correspond to schedule coefficients for a particular statement. 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.

This iterative approach allows `isl` to support negative coefficients without limiting their absolute values while avoiding the trivial zero solution. However, it requires the scheduler to closely interact with the ILP solver. In the worst case this approach considers an exponential number of sign constraints. Practically however, as validity constraints are derived from a loop-based program, ensuring non-triviality for one region often makes other regions non-trivial as well.

Slack for Smaller-Dimensional Statements An n-dimensional schedule for an m-dimensional domain only needs m linearly independent dimensions if $m < n$. Given a schedule with k linearly independent dimensions, `isl` does not enforce linear independence until the last $(m-k)$ dimensions.

3.4 Permutable Bands and Tiling

In general, `isl` scheduler looks for a sequence of schedule functions that satisfy the same set of constraints. Such sequences are referred to as *permutable bands* since individual functions in them can be interchanged without affecting

the semantics of the program. Permutable bands satisfy the sufficient condition for loop tiling, an important locality-improving loop transformation [14]. The scheduler itself does not perform loop tiling, delegating it to the ppcg compiler. The latter tiles outermost permutable bands along with parallelization and GPU mapping if requested [32].

Band Splitting If the ILP problem does not admit a solution, `isl` finishes the current band, removes fully carried dependences and starts a new band. If the first function in the band cannot be computed, the scheduler applies a variant of Feautrier's scheduler [10]. The general idea of this algorithm is to carry as many dependences as possible, ensuring progress. It does so by introducing a penalty e_k for each non-carried dependence, $e_k \le \phi_{\mathsf{R}_k,j}(\boldsymbol{i},\boldsymbol{p}) - \phi_{\mathsf{S}_k,j}(\boldsymbol{i},\boldsymbol{p})$. It then solves the the ILP problem

$$\text{lexmin} \sum_k (1-e_k), \sum_{j=1}^{n_s}\sum_{i=1}^{n_p} d_{j,i}, \sum_{j=1}^{n_s}\sum_{i=1}^{\dim \mathcal{D}_{S_j}} c_{j,i}, e_1 \ldots e_k \ldots \quad (2)$$

where $n_p = \dim \vec{p}$ and n_s is the number of statements. The search space is constrained using Farkas' lemma to values of $c_{j,i}$ that weakly satisfy the *validity* and *coincidence* relations as constraints [34]. Feautrier's algorithm is used as a fallback for `isl` scheduler guaranteeing its termination.

3.5 Data-Dependence Graph Clustering

On the outer level, `isl` scheduler operates on a data-dependence graph (DDG) whose nodes are statements and (typed) edges correspond to dependences between them. Before performing affine transformations, the scheduler separates the graph into strongly-connected components. For each of them, it computes per-statement schedules. Then it selects a pair of clusters that have a *proximity* edge between them. 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 schedule between clusters, that respects inter-cluster validity dependences using the same ILP problem as inside clusters. If such a schedule exists, `isl` combines clusters after checking several profitability heuristics. Otherwise, the scheduler advances to the next candidate pair. The process continues until a single cluster is formed or until all edges are considered. Cluster combination is essentially *loop fusion*, where per-statement schedules are *composed* with schedules between clusters, rescheduling individual clusters with respect to each other. The final clusters are topologically sorted using the validity edges.

Clustering Heuristics Clustering provides control over parallelism preservation and locality improvement during fusion. The scheduler prefers pairs of clusters where schedule dimensions can be completely aligned. Then it checks whether clustering makes the dependence distance along at least one proximity edge constant and sufficiently small. Finally, when parallelism is the objective, `isl` checks that the schedule between clusters contains at least as many coincident dimensions on all individual clusters.

4 Unified Model for Spatial Effects

Modern architectures feature deep memory hierarchies that may affect performance in both positive and negative ways. CPUs typically have multiple levels of cache memory that speed up repeated accesses to the same memory cells—*temporal locality*. Because loads into caches are performed with cache-line granularity, accesses to subsequent memory cells are also sped up—*spatial locality*. However, 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 groups simultaneous accesses from parallel threads to adjacent locations into a single memory request in order to compensate for very long access times. Current polyhedral scheduling algorithms mostly account for the *temporal locality* 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. Unlike dependences, spatial proximity relations do not constrain the execution order. However absolute values of distances along them characterize (spatial) reuse potential, with the value equal to access stride. We loosely refer to a spatial proximity relations as *carried* when the distance along it is not zero.

Spatial proximity relations are used to set up two different ILP problems: one is designed as a variant of (1) to carry as little spatial proximity as possible; another is a variant of (2) intended to carry spatial proximity relations while discouraging skewed schedules. Choosing between these problems allows `isl` to account for memory effects.

4.1 Modeling Line-Based Access

The general feature of the memory hierarchies we model is that *groups* of C subsequent memory cells rather than individual elements can be accessed. For example if $C = 4$, different instances of `A[5*i]` are not spatially related.

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 $\mathcal{A}' = \mathcal{A} \circ C$ where $C = \{\boldsymbol{a} \to \boldsymbol{a}' \mid a'_{1..(n-1)} = a_{1..(n-1)} \wedge a'_n = \lfloor \frac{a_n}{C} \rfloor\}$, and $n = \dim \vec{a} = \dim(\text{Dom}\,\mathcal{A})$. This operation replaces the last array index with a virtual number that identifies memory accesses mapped to the same cache line. We use integer division with rounding to zero to compute the desired value. An individual memory reference now accesses a set of array

elements, and multiple memory references that originally accessed distinct array elements now access the same set.

We use the over-approximative nature of the scheduler to mitigate the actual dynamically-assigned cache lines not being aligned with those we model statically. Before constraining the space of schedule coefficients using Farkas' lemma, `isl` eliminates existentially-quantified variables necessary to express integer division. Combined with transitively-covered dependence elimination, it results in a relations between pairs of (adjacent in time) statement instances potentially accessing the same line. The over-approximation is that the line may start at *any* element and is *arbitrarily large*. While this can be encoded directly, our approach has two benefits. First, if C is large enough, the division-based approach will cover strided accesses. Second, it limits the distance at which *fusion* may be considered beneficial to exploit spatial locality between accesses to *disjoint* sets of array elements.

Accesses to scalars, treated as zero-dimensional arrays, are excluded from line-based access relation transformation since we cannot know in advance their position in memory.

4.2 Spatial and Temporal Proximity Relations

Given line-based read and write access relations, we compute the *spatial proximity* relation using a variant of the dataflow-based procedure to eliminate transitively-covered dependences [9] (s.t. the only statement instances in spatial or temporal relation were adjacent in time in the original program). Note that we also consider spatial Read-After-Read (RAR) "dependence" relations as they are an important source of spatial reuse, and they do not limit parallelism extraction since it is only affected by *coincidence* relations.

Access Pattern Separation The S1 statement `A[i][j] += B[i][j] + B[i-1][j]`, surrounded by two loops, `i` and `j`, features a spatial proximity RAR relation on B characterized by $\mathcal{P}_{S1\to S1,B} = \{(i,j) \to (i',j') \mid (i' = i+1 \wedge \lfloor j'/C \rfloor = \lfloor j/C \rfloor) \vee (i' = i \wedge \lfloor j'/C \rfloor = \lfloor j/C \rfloor)\}$. The first disjunct connects two references that access different parts of the array `B`. Therefore, spatial locality effects are unlikely to appear.

Consider now a statement `S2: C[i][j] += D[i][k] * D[i][j]` enclosed by three loops, `i`, `j` and `k`. Its spatial proximity relation on D is $\mathcal{P}_{S2\to S2,D} = \{(i,j,k) \to (i',j',k') \mid (i' = i \wedge \lfloor k'/C \rfloor = \lfloor j/C \rfloor) \vee (i' = i \wedge \lfloor j'/C \rfloor = \lfloor k/C \rfloor)\}$. Spatial locality may hold only for $|k-j| \leq C$, a significantly smaller number of instances than the iteration domain. The schedule would have to handle this case separately, resulting in inefficient branching control flow.

Generalizing these cases, (group-)spatial locality between access with different *access patterns* is difficult to exploit efficiently in an affine schedule. Two access relations are considered to have different patterns if there is at least one access function that differs between them. The last function is considered without the constant factor, that is `D[i][j]` has the same pattern as `D[i][j+2]`, but not as `D[i][j+N]`.

Access Completion Consider the matrix multiplication core statement `C[i][j] += A[i][k] * B[k][j]`, surrounded by three loops. There exists, among others, a spatial RAR relation between its instances induced by reuse on `B`: $\mathcal{P}_{R\to R,B} = \{(i,j,k) \to (i',j',k') \mid ((i' = i \wedge j' = j+1 \wedge \lfloor j'/C \rfloor = \lfloor j/C \rfloor \wedge k' = k) \vee (\exists \ell \in \mathbb{Z} : i' = i+1 \wedge j' = C\ell \wedge j = C\ell + C - 1 \wedge k' = k))\}$. The second disjunct expresses spatial reuse between iterations of the outer loop, `i`, which, again, only exists for a small number of statement instances if the trip count is larger than C. To exploit this reuse, the scheduler may *skew* the inner loop by $(C-1)$ resulting in inefficient control flow. Pattern separation is useless in this case since `B[k][j]` is the only reference with the same pattern. However, we can prepend an access function `i` to simulate that different iterations of the loop `i` access disjoint parts of `B`.

Note that the array reference `B[k][j]` only uses two iterators out of three available. Collecting the coefficients of affine access functions as rows of matrix A, we observe that such problematic accesses do not have full column rank. Therefore, we *complete* this matrix by prepending *linearly independent* rows until it reaches full column rank. We proceed by computing the Hermite Normal Form $H = A \cdot Q$ where Q is $n \times n$ unimodular matrix and H is an $m \times n$ lower triangular matrix, i.e. $h_{ij} = 0$ for $j > i$. Any row-vector v with at least one non-zero element $v_k \neq 0, k > m$ is linearly independent from all rows of H. We pick $(n-m)$ standard unit vectors $\hat{e}_k = (0 \ldots 0, 1, 0, \ldots 0), m < k \leq n$ to complete the triangular matrix to an n-dimensional basis. Transforming the basis with unimodular Q preserves its completeness. In our example, it performs the desired transformation from `B[k][j]` to `B[i][k][j]`.

The combination of *access pattern separation* and *access completion* keeps a reasonable subset of spatial proximity relations, exploitable by an affine scheduler, while limiting the number of constraints the ILP solver has to handle.

4.3 Carrying Few Spatial Proximity Relations

Depending on the target architecture and on the scheduling step, we need an affine schedule function that carries either few or many spatial proximity relations. Let us first describe an ILP problem for carrying *few* spatial proximity, which corresponds to making the distance zero along *many* relations. Unlike *coincidence* relations, *some* of them may be carried and unlike *proximity* relations, small non-zero distances are seldom beneficial. Therefore, we systematically remove carried *spatial proximity* relations from further consideration.

In presence of contradictory requirements, e.g. spatial locality for `A[i][j]` and `B[j][i]`, minimizing the *sum* of distance bounds (as for temporal proximity) makes ϕ_k = `i` and ϕ_k = `j` indistinguishable for the ILP. Instead, we consider bounds for separate *groups* of spatial proximity relations, each of which is carried independently of others These groups will be described in Section 4.4 below.

Attempting to force zero distances for the largest possible number of groups with relaxation on failure is combinatorially complex. Instead, we minimize the distances and only keep the relations for which the distance is zero. Intuitively, this removes the first group that must be carried if the previous groups are not. This encoding does not guarantee a *minimal number* of groups is carried, however it allows for an *external non-linear input* to the scheduler by means of ordering the groups in the lexmin formulation.

Combining Temporal and Spatial Proximity Generally, we expect *spatial locality* to be less beneficial for performance than *temporal locality*, which we prioritize. We achieve this by grouping temporal proximity relations in the same way as spatial 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. Any proximity relations carried by the current partial schedule are removed iteratively.

The new ILP minimization objective is

$$\begin{aligned}&\text{lexmin } \textstyle\sum_{i=1}^{n_p}(u^{T+}_{1,i}+u^{T-}_{1,i}), w^T_1, \sum_{i=1}^{n_p}(u^{S+}_{1,i}+u^{S-}_{1,i})\ldots,\\ &\textstyle\sum_{i=1}^{n_p}(u^{T+}_{n_g,i}+u^{T-}_{n_g,i}), w^T_{n_g}, \sum_{i=1}^{n_p}(u^{S+}_{n_g,i}+u^{S-}_{n_g,i}), w^S_{n_g}\ldots\end{aligned} \tag{3}$$

where $u^T_{j,i}$ are coefficients of the parameters and w^T_j is the constant factor in the distance bound for the j^{th} group of proximity relations, $1 \le j \le n_g$, and $u^S_{j,i}$, w^S_j are their counterparts for temporal proximity relations. The remaining non-bound variables are similar to those of (1): the sum of schedule coefficients and parameters, and coefficient values.

4.4 Grouping and Prioritizing for Spatial Proximity

Proximity relation grouping resolves carry-conflicts by prioritization and reduces the number of ILP variables. Therefore, it is performed except if, at some minimization step, one of the relations must be carried while the other should not.

Initial Groups Consider again the statement `C[i][j] += A[i][k] * B[k][j]` surrounded by three loops, `i`, `j` and `k`. It features spatial reuse on `A` carried by `k` as well as on `B` and `C` carried by `j`. Considering relations that characterize it together would prevent the scheduler from taking reasonable decisions and make it choose the original loop order: (`i`, `j`, `k`). However, inverting `k` and `j` loops will exploit spatial locality for `B` and `C` and temporal locality for `A`. Therefore, we introduce a group for each *array reference*.

Dependence distance bounds are computed per group and ordered in the lexmin to prioritize carrying those groups that are potentially less profitable in case of conflict. We avoid carrying groups in which reuse can still be exploited and those that correspond to multiple references. This is achieved by lexicographically sorting them following the decreasing access *rank* and *multiplicity*, which are defined below.

Access Rank Each array reference is characterized by an access relation $\mathcal{A} \subseteq (\vec{i} \to \vec{a})$. If all subscripts are already fixed by the current partial schedule, subsequent decisions will not modify the locality of this reference. Non-fixed subscripts can still be aligned with schedule dimensions to exploit locality, and their number defines the *access rank*. Given the partial schedule $\mathcal{T} \subseteq (\vec{i} \to \vec{o})$, we compute the scheduled access relation $\mathcal{A} \circ \mathcal{T}^{-1} \subseteq (\vec{o} \to \vec{a})$. Fixed subscripts correspond to equations defining this relation. Therefore, the *rank* is computed as difference between the number of subscripts $\dim \vec{a}$ and the number of equations in $\mathcal{A} \circ \mathcal{T}^{-1}$.

Access Multiplicity For equal 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+2]` both have *multiplicity* = 2. Read and write accesses caused by compound assignment contribute to multiplicity twice.

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 their *multiplicities* are added.

4.5 Carrying Many Spatial Proximity Relations

Let us now describe the ILP problem for carrying many spatial proximity relations, with small (reuse) distance. Feautrier's ILP formulation (2) produces affine functions that carry as many dependences as possible but often does so by skewing. However, skewing often leads to loss of locality by introducing additional iterators in the array subscripts. Therefore, we modify (2) to discourage skewing by swapping the first lexmin components: first, minimize the sum of schedule coefficients thus discouraging skewing without avoiding it completely; second, minimize the number of *non-carried* dependence groups. Because the minimal achievable sum of schedule coefficients is zero, we also include the linear independence method of Section 3.3. It is slightly modified to remain in effect even if "dimension slack" is available. The objective defined for groups of Section 4.4 becomes

$$\text{lexmin} \sum_{i=1}^{\max \dim \mathcal{D}_S} \sum_{j=1}^{n_s}(c^-_{j,i}+c^+_{j,i}), \sum_{k=1}^{n_g}(1-e_k), \sum_{i=1}^{n_p}\sum_{j=1}^{n_s} d_{j,i}\ldots \tag{4}$$

where n_s is the number of statements, n_p is the number of parameters, e_k are defined similarly to (2) for each of n_g groups. Validity constraints must be respected, distances along coincidence relations are to be made zero if requested.

4.6 Scheduling for CPU Targets

On CPUs, spatial locality is likely to be exploited if the innermost loop accesses subsequent array elements. False sharing may be avoided if parallel loops do not access adjacent

elements. We expect to produce a good CPU schedule by using (3) for all dimensions except the last, where we apply (4).

Parallelism/Locality Trade-off As we exploit only one coarse-grained degree of parallelism with OpenMP pragmas, we relax *coincident* relations if one coincident dimension was found. The *clustering* mechanism now tolerates loss of parallelism as long as one coincident dimension is left.

Coarse-grained parallelism is featured by schedules with outer coincident dimensions. Unlike the default isl heuristic (see Section 3.2), CPUs require deeper tilable bands with the non-coincident outermost dimension. Instead, *wavefront* parallelism is extracted by skewing the outermost dimension by the subsequent one, which then becomes parallel.

Finally, marking innermost loops OpenMP parallel often results in excessive barrier synchronization. Therefore, we relax *coincidence* relations when two dimensions remain to schedule, even if no coincident dimension was found.

Post-tile Reordering We modified ppcg to optionally perform the *post-tile reordering*, borrowed from Pluto. If a schedule dimension is coincident and carries spatial proximity, it is likely to be placed outermost by the scheduler, exploiting parallelism. After tiling, the point loop dimension still carries spatial proximity and may be safely placed innermost, additionally exploiting spatial locality. The dimension to sink is chosen based on the number of scheduled accesses it carries.

Carrying Dependences to Avoid Fusion The *band splitting* procedure (see Section 3.4) often leads to separation of the DDG into components, which corresponds to *loop distribution*. 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. Uniqueness is required to consider repeated accesses to the same array with the same subscripts once. This heuristic is based on the assumption that each supplementary array access uses a register. It also penalizes deeply nested accesses by accounting for the input dimension of the access relation.

This heuristic applies when (3) does not produce an outer coincident dimension. When $h > h_{\text{lim}}$ we apply (2) to compute the outer dimension instead of (3). Otherwise, we relax the zero-distance constraint for *coincidence* relations and continue the band similarly to inner parallelism avoidance. Tuning h_{lim} to a particular system prevents *some* fusion with outermost parallel loops and thus decrease register pressure.

4.7 Scheduling for GPU Targets

Efficient scheduling for GPUs requires the scheduler to expose three or more degrees of parallelism and to be aware of how loops are mapped to blocks and threads. After tiling, ppcg maps the three outermost coincident tile(point) dimensions to blocks(threads) in inverse order, i.e., z, y, x.

We first apply (4) while enforcing zero distance along *coincidence* relations. The outer coincident dimension is preferred as offers the largest choice of spatial proximity relations to carry for coalescing. If no solution is found, we apply (2) in an attempt to expose *multiple* levels of inner parallelism. If a coincident solution is found, but it does not carry spatial proximity, we discard it and minimize (1) instead. In any case, we discard *spatial proximity* relations after one coincident dimension: if spatial reuse can be exploited, it will be present in the first member of the band because all members must carry the same relations. The following bands are produced using (1) and (2) as they are not mapped to blocks or threads. Finally, we alter the mapping if the outermost coincident dimension carries spatial proximity and place it to x threads.

5 Experimental Evaluation

We compared speedups obtained by our approach with those of other polyhedral schedulers on CPUs and GPUs. We instantiated our algorithmic template with and without spatial locality support, to highlight its specific performance impact.

5.1 Implementation Details

Our proposed algorithm is implemented as an extension to `isl-0.18-730-gd6628369` and `ppcg-0.07`.[2]

Miscellaneous improvements were introduced to `isl` alongside the design and implementation of the new scheduler. Optimizing (2) in integers instead of a rationals if the latter gives rational solutions avoids large schedule coefficients. Original loop iterator order is used in case of cost function ties. The sum of coefficients for *original*, rather than newly computed loop iterators is minimized in (1).

5.2 Experimental Protocol

The target platforms include multi-core CPUs and GPUs, strating from the same code to demonstrate performance portability. Our testbed includes:
ivy/kepler: 4× Intel Xeon E5-2630v2 (Ivy Bridge, 6 cores, 15MB L3 cache), NVidia Quadro K4000 (Kepler, 768 CUDA cores) on CentOS Linux 7.2.1511 with gcc 4.9 and nvcc 8.0.61.
skylake, Intel Core i7-6600u on Ubuntu 17.04 with gcc 6.3.0.
westmere, 2× Intel Xeon X5660 (Westmere, 6 cores, 12MB L3 cache) running RHEL Server 6.5 with icc 15.0.2.

We evaluate our tools on PolyBench/C 4.2.1. We removed a `typedef` from `nussinov` benchmark and introduced variants of `symm`, `deriche`, `doitgen` and `ludcmp` benchmarks with scalar/array expansion applied to expose more parallelism. On CPUs, all benchmarks are executed with `LARGE` data sizes to represent more realistic workloads. On GPUs, we used custom, often larger data sizes for GPUs reported in Figure 3.

Since the Pluto+ implementation cannot handle several of the Polybench 4.2.1 benchmarks, we compare against Pluto given that [4] reports that Pluto+ and Pluto generate identical

[2] Available at git://repo.or.cz/ppcg.git and git://repo.or.cz/isl.git

schedules for PolyBench. We compare
- ppcg *stable*: latest ppcg release (`ppcg-0.07`, `isl-0.18`)
- ppcg *w/o spatial*: see implementation details;
- ppcg *spatial*: Section 4, w/ and w/o *post-tile* reordering;[3]
- *Pluto*: Pluto 0.11.4 with `--parallel --tile` options;
- *PolyAST*: disabling reduction and DoAcross parallelism.[4]

Loops were tiled with size 32 on CPUs and 16 on GPUs to better fit into memory. No tile size tuning was performed.

We collected execution times using the PolyBench timing facility on CPU, and using the NVidia CUDA profiler on GPUs (total kernel execution time reported). We report a median of 5 measurements for each condition.

5.3 Sequential Code Performance

The **skylake** system with AVX2 instruction set allowed us to evalute performance improvements on *sequential* programs with vector parallelism and multi-level caches. For all schedulers, we requested tiling and post-tile optimizations. For ppcg, we additionally considered spatial proximity for fusion. The speedups are shown in Figure 1(top).

Spatial effects-aware scheduling improved performance with two ppcg versions for `2mm`, `3mm`, `gemver`, `mvt` and `symm`. Pluto was unable to transform `symm` while our flow achieves 2.4× speedup. For `atax`, `deriche`, `jacobi-1d`, `ludcmp`, all variants of ppcg generate faster code due to (1) a different loop fusion structure thanks to clustering and (2) live-range reordering. Small performance changes between Pluto and ppcg-spatial, in `covariance`, `correlation` or `trmm`, are due to the differences in code generation algorithms: ppcg may generate simpler control flow than CLooG, used in Pluto. Finally, Pluto outperforms ppcg for `adi`, `gesummv` and `gramschmidt` since it may tile imperfectly nested loops, contrary to ppcg. Post-tile reordering had only a marginal effect.

5.4 Parallel CPU Code Performance

The **ivy** system running 24 threads allowed us to expore the interplay between parallelism and locality. We requested parallelization, tiling and post-tile reordering in all cases, and enabled all heuristics presented in this paper in ppcg. The speedups are reported in Figure 1(middle).

Our flow results in significant speedup over Pluto for numerous benchmarks. For example, speedup for `3mm` grows from 6.5× to 16.7×. Both the clustering technique and the spatial effects-aware model contribute to these imporvements. Furthermore, spatial model corrects performance of multiple cases where baseline ppcg was counterproductive. It is also able to achieve up to 1.4× for stencil-like codes `heat-3d` and `jacobi-1d` where Pluto yields a 2× slowdown. Similarly to sequential version, Pluto outperforms ppcg on `gramschmidt` (8.8× and 2.9× speedup, respectively) and `nussinov` due to ppcg's inability to tile imperfectly nested loops.

[3] Available at https://pollylabs.org/spatial.html

[4] Parallel reductions ignored and DoAcross converted into wavefront DoAll.

Syntactic post-tile reordering is not always beneficial in our flow: it increases the speedup for `covariance` from 30.5× to 32.4× and *decreases* it from 33× to 28.7× for `correlation`.

5.5 Comparison with Affine+Syntactic Approach

We compared our results with those of PolyAST, a state-of-the-art hybrid tool that combines affine scheduling for locality and syntactic transformations for parallelism. The speedups on **westmere** are shown in Figure 1(bottom).

Overall, the observed performances for PolyAST and ppcg are very close and so are the schedules, which confirms our intuition that a fully-polyhedral scheduler can compute schedules comparable to a hybrid approach. Identical schedules were produced for `2mm`, `3mm` and `floyd-warshall` with minor performance variations for the latter due to differences in code generation. Without tuning to **westmere**, the register pressure reduction heuristic was less efficient: while ppcg obtains 2.9× speedup on `heat-3d` where *PolyAST* has 1.2×, it obtains only 3.7× on `jacobi-2d` where *PolyAST* has 6.5×. Setting $h_{\text{lim}} = 32$ for this system would produce identical schedules. For `atax` and `trmm`, both Pluto and ppcg-spatial outperform PolyAST as the latter places non-doall loops outermost and loses outer parallelism. Finally, PolyAST could not handle `adi` and `nussinov` in the polyhedral framework.

5.6 Parallel GPU Code Performance

We only evaluated variants of ppcg on **kepler** GPUs as Pluto and PolyAST-GPU rely on drastically different code generation schemes for GPUs. Spatial effects modeling affected the schedule in six benchmarks, see Figure 2.

For all cases except `lu`, ppcg discovers no outer parallelism and resorts to repeated kernel calls, see Figure 3 for cumulative numbers. Thanks to different fusion structure for, our flow reduces the number of kernel calls and the related overhead for `lu` and `gramschmidt`. Kernel execution is faster thanks to improved memory coalescing, e.g. on `symm`. For `trisolv`, the kernel execution time is marginal in the total execution time, resulting in close to zero speedups. Finally, for `seidel-2d`, ppcg witnesses performance regressions. In fact, the values of the cost function for the two innermost loops are identical and the stable ppcg happened to interchange them while the two others always preserve the original loop order. Thus, the superior performance of stable ppcg was accidental, and not a result of a scheduling decision. Correcting this regression requires the scheduling algorithm to jointly optimize for different memory spaces.

Beyond these cases, spatial effects modeling did not affect the schedule since parallelism is prioritized over locality for GPUs. Larger benchmarks with longer execution time would be necessary to fully assess the benefits of our flow on GPUs.

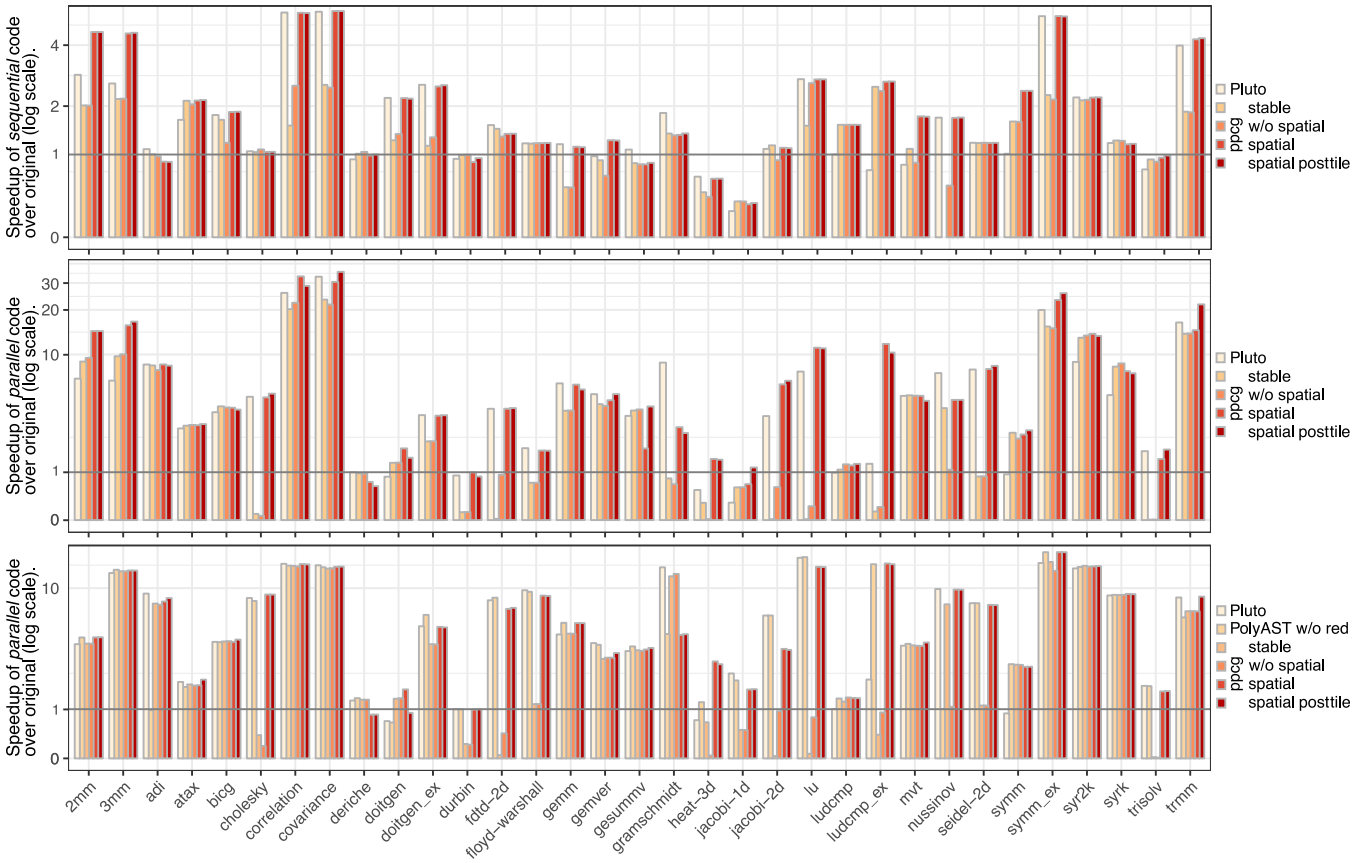

Figure 1. Speedup of the optimized tiled code over the original code with different scheduling algorithms; top: sequential code on **skylake**, middle: parallel code on **ivy**; bottom: parallel code on **westmere**.

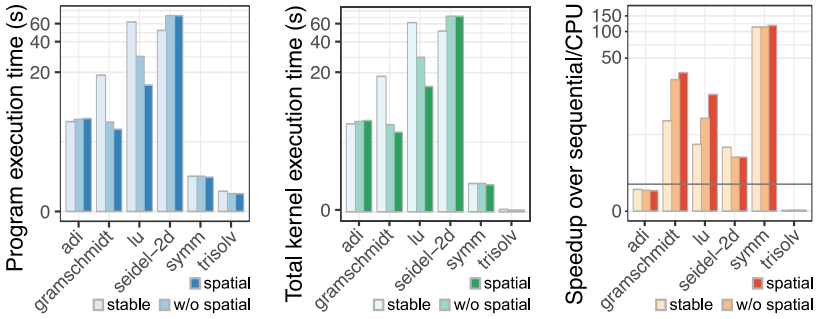

Figure 2. Left and center: total kernel execution time and program execution time (lower is better). Right: speedup over sequential CPU (higher is better).

	adi	gram.	lu
parameter value	512	2048	4096
Original: # kernels	14	7	3
# invocations	7168	28643	20471
Spatial # kernels	6	7	2
# invocations	3072	12287	8190
	seidel-2d	symm	trisolv
parameter value	$1k \times 4k$	2048	4096
Original: # kernels	1	2	3
# invocations	16372	2	12286
Spatial: # kernels	1	2	3
# invocations	16372	2	8192

Figure 3. Parameter values, # of kernels generated and cumulative kernel invocations (lower is better).

6 Discussion and Future Work

Before summarizing our findings, let us discuss some of the algorithmic design choices hinting at possible 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., for fusion.

Dependence Analysis for Spatial Proximity Relations Proximity relations result from a typical dependence analysis, pruning transitively closed dependences. They only capture statement instances that have spatial proximity *in the original program*; it may eliminate a read-after-read relation transitively covered by other relations. This allows to associate each relation with a constant access stride. While it it possible to preserve the full relations by pruning locally when computing access strides, this would damage algorithmic complexity with no significant performance improvement.

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 to tweak the behavior of the algorithm without modifying the ILP formulation itself. 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.

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 *tunable* heuristic to choose between two alternative ILPs and to leverage their side effects. Other, complementary approaches may be used to reduce register pressure further [13, 27].

Post-Tile Reordering Some benchmarks require additional transformation after tiling. However, one of the scheduling objectives is to maximize the depth of tilable bands. A two-phase affine scheduling may be required to produce more profitable schedules, the second phase being applied after tiling and preserving the band structure. It will provide a more robust alternative to post-tile heuristics for locality and wavefront parallelization, and allow for simultaneous fusion and rescheduling after tiling.

7 Related Work

Within the polyhedral framework, automatic scheduling has been the subject of active research over the past three decades. Feautrier's algorithm [10] produces minimal-delay schedules with fine-grained parallelism by forcing the outermost loops to carry the maximum number of dependences. Lim and Lam's algorithm [18] aims to minimize synchronizations, hence maximizing coarse-grain parallelism. Pluto [7] combines parallelization and locality optimization through tiling. It resorts to a post-scheduling loop reordering heuristic to account for spatial locality whereas our approach consistently models spatial effects in the ILP allowing to avoid undesirable effects such as false sharing. Recent work on Pluto+ [4] introduces support for negative coefficients but, unlike our approach, imposes constant bounds on the optimization space. Recent work integrates access consecutivity as a polyhedral scheduling objective. Trifunovic et al. [28] propose a scheduling strategy for automatic vectorization, but consider loop permutations only. Kong et al. [17] encode vectorizability of point loops as an ILP and rely on a domain-specific SIMD code generator. Building on a work by Bastoul et al. [2], Vasilache et al. [29] proposed contiguity constraints to capture innermost reuse along one dimension of an arary reference. All aforementioned approaches restrict the space of possible schedules which, as we demonstrated, misses profitable opportunities that rely on linearly dependent dimensions or exploit non-contiguous accesses.

Much of the past work focused on specific transformations, such as loop fusion [16, 21], initially designed as a locality-enhancing optimization in isolation from other loop nest transformations. These techniques often model temporal locality [3, 6] and introduce criteria similar to those of our clustering method [19]. Clustering combines fusion with scheduling to reduce the size of the linear problems to solve.

Outside the polyhedral framework, loop nest optimization holds a particular place in optimizing compilers [15]. Numerous syntactic locality-improving loop transformations were proposed, including loop interchange [1] and tiling [14, 36]. Syntactic methods apply a sequence of individual loop transformations driven by analytical cost models [20, 25], for parallelization or vectorization [35]. PolyAST [26] employs a two-stage approach: first, the polyhedral affine scheduling optimizes temporal and spatial locality, guided by the DL cost model [25]; second stage detects outermost forall, reduction, or doacross loop parallelism, using syntactic information on commutativity and associativity and on polyhedral dependence information. In isolation, optimization stages may end up undoing each other's work, hitting a compiler phase ordering problem. Our approach combines both optimization criteria in a single problem and prioritizes parallelism or locality if conflicting transformations are required.

8 Conclusion

We proposed a template for the construction of affine scheduling algorithms that accounts for multiple levels of parallelism and deep memory hierarchies. Our approach models both temporal and spatial effects, orchestrating a collection of parameterizable optimization problems with configurable constraints and objectives. The algorithmic template addresses non-convexity without increasing the number of discrete variables in linear programs, without imposing a priori limits on the space of possible transformations, and modeling schedules with linearly-depedendent dimensions that are out of reach of a typical polyhedral optimizer.

Our algorithmic template generates sequential, parallel, or accelerator code in a single optimization pass, matching or outperforming comparable frameworks, whether polyhedral, syntactic, or a combination of both. We discussed the rationale for this unified algorithm, as well as its validation on representative benchmarks.

Our results restore hope in the design of performance-portable loop nest optimizer that are also simpler and more elegant. We also believe our approach applies to domain-specific optimization, mapping high-level equations occurring in numerical simulations as well as machine learning algorithms, on both dense and sparse structures, targeting manycore and reconfigurable hardware.

Acknowledgments

This work was partly supported by the European Commission via the Eurolab-4-HPC coordination action id. 671610, by the French ANR through the European CHIST-ERA project DIVIDEND, by Swiss Universities in the context of ComPASC and by US DOE Exascale Computing Project in the context of SOLLVE. We also acknowledge the support of ARM Holdings PLC through the Polly Labs initiative. This work benefited from numerous discussions with Nicolas Vasilache.

References

[1] John R. Allen and Ken Kennedy. 1984. Automatic Loop Interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (SIGPLAN '84)*. ACM, New York, NY, USA, 233–246.

[2] Cédric Bastoul and Paul Feautrier. 2003. Improving Data Locality by Chunking. In *Compiler Construction*, Görel Hedin (Ed.). Number 2622 in Lecture Notes in Computer Science. Springer Berlin, 320–334.

[3] Cédric Bastoul and Paul Feautrier. 2005. Adjusting a program transformation for legality. *Parallel processing letters* 15, 01n02 (2005), 3–17.

[4] Uday Bondhugula, Aravind Acharya, and Albert Cohen. 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 (April 2016), 12:1–12:32.

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

[6] Uday Bondhugula, Oktay Günlük, Sanjeeb Dash, and Lakshminarayanan Renganarayanan. 2010. 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*. 343–352.

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

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

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

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

[11] Paul Feautrier and C. Lengauer. 2011. Polyhedron Model. In *Encyclopedia of Parallel Computing*, D. Padua (Ed.). Springer, 1581–1592.

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

[13] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J Ramanujam, and P Sadayappan. 2011. Data layout transformation for stencil computations on short-vector simd architectures. In *Compiler Construction*. Springer Berlin/Heidelberg, 225–245.

[14] F. Irigoin and R. Triolet. 1988. Supernode Partitioning. In *15th Symp. on Principles of Programming Languages*. ACM, NY, USA, 319–329.

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

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

[17] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 127–138.

[18] Amy W. Lim and Monica S. Lam. 1997. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 201–214.

[19] Amy W. Lim, Shih-Wei Liao, and Monica S. Lam. 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)*. ACM, New York, NY, USA, 103–112.

[20] Kathryn S. McKinley, Steve Carr, and Chau-Wen Tseng. 1996. Improving Data Locality with Loop Transformations. *ACM Trans. on Programming Languages and Systems* 18, 4 (July 1996), 424–453.

[21] Nimrod Megiddo and V. Sarkar. 1997. Optimal weighted loop fusion for parallel programs. In *Parallel Algorithms and Architectures*. 282–291.

[22] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. 2006. GRAPHITE: Polyhedral Analyses and Optimizations for GCC. In *Proceedings of the 2006 GCC Developers Summit*. 179–197.

[23] Louis-Noël Pouchet, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. 2011. Loop Transformations: Convexity, Pruning and Optimization. In *Symp. on Principles of Programming Languages*. ACM, NY, USA, 549–562.

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

[25] Vivek Sarkar. 1997. Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers. *IBM Journal of Research & Development* 41, 3 (May 1997).

[26] J. Shirako, L. N. Pouchet, and V. Sarkar. 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*. 287–298.

[27] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2013. A Framework for Enhancing Data Reuse via Associative Reordering. ACM Press, 65–76.

[28] Konrad Trifunovic, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen. 2009. Polyhedral-model guided loop-nest auto-vectorization. In *Conf. on Parallel Architectures and Compilation Techniques*. 327–337.

[29] Nicolas Vasilache, Benoît Meister, Muthu Baskaran, and Richard Lethin. 2012. Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization. In *2nd Intl. Workshop on Polyhedral Compilation Techniques*. Paris, France.

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

[31] Sven Verdoolaege. 2011. Counting Affine Calculator and Applications. In *1st Intl. W. on Polyhedral Compilation Techniques*. Chamonix, France.

[32] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. 2013. Polyhedral Parallel Code Generation for CUDA. *Transactions on Architecture and Code Optimization* 9, 4 (Jan. 2013), 54:1–54:23.

[33] Sven Verdoolaege and Albert Cohen. 2016. Live Range Reordering. In *6th W. on Polyhedral Compilation Techniques*. Prague, Czech Republic.

[34] Sven Verdoolaege and Gerda Janssens. 2017. *Scheduling for PPCG*. Report CW 706. Department of Computer Science, KU Leuven, Belgium.

[35] Michael Wolfe. 1986. Loop Skewing: The Wavefront Method Revisited. *Int. J. Parallel Program.* 15, 4 (Oct. 1986), 279–293.

[36] Michael Wolfe. 1989. Iteration Space Tiling for Memory Hierarchies. In *3rd Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, USA, 357–361.