Extending Pluto-Style Polyhedral Scheduling with Consecutivity

Sven Verdoolaege
Polly Labs and KU Leuven
sven.verdoolaege@gmail.com

Alexandre Isoard
Xilinx
alexandre.isoard@gmail.com

Abstract

The Pluto scheduler is a successful polyhedral scheduler that is used in one form or another in several research and production compilers. The core scheduler is focused on parallelism and temporal locality and does not directly target spatial locality. Such spatial locality is known to bring performance benefits and has been considered in various forms outside and inside polyhedral compilation. For example, the Pluto compiler has some support for spatial locality, but it is limited to a post-processing step involving only loop interchange. Consecutivity is a special case of spatial locality that aims for stride-1 accesses, which can be useful for constructing burst accesses and for vectorization. Stride-1 accesses have been targeted by an approach based on one-shot scheduling, but it is fairly approximative and not directly transferable to a Pluto-style scheduler. This paper describes an approach for consecutivity that is integrated in a Pluto-style polyhedral scheduler, as implemented in `isl`. Both intra-statement and inter-statement consecutivity is considered, taking into account multiple references per statement and the integration into a component based incremental scheduler.

1 Introduction and Motivation

A program is said to exhibit locality if it reuses some data element stored in some form of cache before it gets evicted. A distinction is usually made between temporal locality, where the same element is reused, and spatial locality, where the reuse may be of some other element that is loaded into the cache together with the element that was accessed first, e.g., because they share a cache line. Improving spatial locality therefore usually brings performance benefits by increasing cache hit rate.

Consecutivity is a special case of spatial locality, where consecutive accesses to memory access consecutive elements. Consecutivity facilitates memory access vectorization and usually allows the hardware cache prefetcher to successfully predict the next memory access. The main motivation for this paper, however, is coalescing consecutive accesses into a single burst request, which is useful on architectures such as FPGAs to compensate for the difference in clock frequency between the logic and the external memory interface and to allow the memory controller to optimize the accesses, usually guaranteeing close to one (widened) memory access per cycle, thus fully utilizing the available memory bandwidth.

Xilinx (2017, Chapter 6) notably recommends using memory ports as wide as 512 bits (e.g., vectors of 16 elements for 32 bits integer) and bursting memory transfers from off-chip global memory. Xilinx (2017, Appendix B) further suggests storing the data into temporary buffers in on-chip memory (BlockRAM) so as to freely perform all the memory accesses of each array in a few bursts. This means, in particular, that reads and writes should be made consecutive separately and that there is little use in mixing consecutive accesses with non-consecutive accesses to the same memory/array.

There are many conceivable strategies for trying to achieve consecutivity or, more generally, spatial locality. The optimization can be performed as a post-processing step or it can be integrated into a scheduler. Within polyhedral compilation, a popular scheduling approach is to construct constraints on schedule coefficients through an application of the Farkas lemma (Schrijver 1986, Corollary 7.1h, page 93; Feautrier 1992a, Theorem 7), but there are also other approaches such as those based on transitive closures (e.g., Bielecki et al. 2017). Within the Farkas based approaches, there are two main groups, those such as the Pluto scheduler (Bondhugula, Baskaran, et al. 2008) that compute a schedule row by row and those that compute a multi-dimensional schedule in one shot based on a convex space of valid schedules (Vasilache 2007; Pouchet et al. 2011).

Each such approach has its own advantages and disadvantages, but a detailed comparison is beyond the scope of this paper. Instead, this paper focuses on one choice and describes how to add support for consecutivity to the `isl` scheduler (Verdoolaege, Juega, et al. 2013; Verdoolaege and Janssens 2017), a scheduler based on the Pluto scheduler that is used in GCC/graphite (Trifunovic, Cohen, et al. 2010), LLVM/Polly (Grosser et al. 2012) and `PPCG` (Verdoolaege, Juega, et al. 2013). In particular, this paper describes

- the derivation of constraints on schedule coefficients for trying to achieve consecutivity in a row-by-row polyhedral scheduler, without introducing any additional variables in the ILP problem,
- an approach for solving these constraints in conjunction with other constraints directed at correctness, parallelism and/or temporal locality,

IMPACT 2018 (http://impact.gforge.inria.fr/impact2018)
Eighth International Workshop on Polyhedral Compilation Techniques,
In conjunction with HiPEAC 2018.
January 23, 2018, Manchester, United Kingdom

- an algorithm for combining consecutivity constraints derived from multiple references, and
- the integration into a component based incremental scheduler.

Note that this paper only focuses on consecutivity and, in particular, does not explain how to ensure that the consecutive accesses can also be executed in parallel, which would be an additional requirement for vectorization. If the innermost tilable band in the generated schedule happens to be fully parallel, then this will be the case. Otherwise, additional techniques may be required, as briefly discussed in Section 6. More details on the consecutivity support are available from Verdoolaege and Isoard (2017). A prototype implementation is available in `consecutivity_CW_709` of git://repo.or.cz/isl.git and git://repo.or.cz/ppcg.git. This implementation is oriented towards optimizing the innermost fully parallel loops for consecutivity.

2 Background

2.1 Terminology

For the purpose of consecutivity, only purely affine references will be considered, i.e., a single expression defined over a universe domain that does not involve any quasi-affine elements. For such an affine array reference $\mathtt{A}[F\mathbf{i}+\mathbf{c}]$, the matrix F will be called the *linear part* and it will often be split into the final row H and the remaining rows G, i.e.,

$$F = \begin{bmatrix} G \\ H \end{bmatrix}. \tag{1}$$

The linear part of the schedule transformation for a particular statement will be represented by T.

Given a matrix M, its *null-space* is the set $\ker M = \{\, \mathbf{x} : M\mathbf{x} = \mathbf{0} \,\}$. The *orthogonal complement* of M is a matrix with as rows any basis for its null-space. Two forms of linear independence are considered in this paper. An $m \times n$-dimensional matrix M is said to be *linearly independent* if the rows of M are linearly independent, i.e., if $\operatorname{rank} M = m$. An $m_1 \times n$-dimensional matrix M_1 is said to be *linearly independent* of an $m_2 \times n$-dimensional matrix M_2 if there is no linear dependence among the combined rows that is not a linear dependence when restricted to the rows of M_1 (or M_2), i.e., if

$$\operatorname{rank} \begin{bmatrix} M_1 \\ M_2 \end{bmatrix} = \operatorname{rank} M_1 + \operatorname{rank} M_2. \tag{2}$$

Note that this is a symmetric property, meaning that if M_1 is linearly independent of M_2, then M_2 is also linearly independent of M_1. A *basis extension* of a matrix A to cover B, written $B \setminus A$ is formed by rows C that extend a basis of A to a basis that also covers B. One way of computing such a matrix C is described by Verdoolaege and Isoard (2017, Section 5.4).

2.2 The `isl` Scheduler

The `isl` scheduler was first introduced by Verdoolaege, Juega, et al. (2013) and is explained in detail by Verdoolaege and Janssens (2017). See also Appendix A. The scheduler takes as input a set of statement instances that need to be scheduled as well as different forms of schedule constraints. The most important schedule constraints are validity schedule constraints, which enforce a relative order between pairs of statement instances, proximity schedule constraints, which tell the scheduler to try and schedule pairs of statement instances close to each other, and coincidence schedule constraints, which tell the scheduler to try and schedule pairs of statement instances together for as long as possible.

The `isl` scheduler combines two scheduling algorithms, (a variant of) the Pluto scheduler (Bondhugula, Baskaran, et al. 2008), and the Feautrier scheduler (Feautrier 1992b). The Pluto scheduler tries to compute multiple, linearly independent schedule rows using the same schedule constraints. These schedule rows form the *members* of a *band*. Note that these members not only need to be linearly independent of each other, but also of members of outer bands. That is, if T_0 is the linear part of the schedule computed so far for a particular statement, then the next schedule row C for that statement needs to be such that

$$\neg\,(\exists Y : C = YT_0)\,. \tag{3}$$

This linear independence constraint is relaxed for statements with a dimension n that is smaller than the maximal statement dimension m. In particular, no constraint is imposed as long as a total number of n linearly independent rows can still be found in subsequent steps, i.e., if

$$n - r_1 < m - \ell, \tag{4}$$

with $r_1 = \operatorname{rank} T_0$ and ℓ the number of rows in T_0. If no more rows can be computed within a band, the schedule constraints that do not relate statement instances that are coscheduled by the band are removed and a new, nested band is constructed. If no such band can be constructed, then a single iteration of the Feautrier scheduler is used to create a schedule row, ignoring proximity schedule constraints and coincidence schedule constraints.

The Pluto scheduler variant implemented in `isl` first computes bands for each strongly connected component in the statement-level schedule constraint graph separately, after which the components are combined incrementally by scheduling them with respect to each other, rejecting combinations that do not optimize at least some proximity schedule constraints. For each component and for each band member, the scheduler constructs one or two ILP problems for computing the next schedule rows by translating the validity, proximity and coincidence schedule constraints to constraints on the schedule coefficients through an application of the Farkas Lemma. If there are any coincidence schedule constraints, then they are first included in the ILP and if this fails to produce a solution, a second ILP is constructed without them. Linear independence constraints (3) are imposed through backtracking. In particular, the orthogonal complement U of

T_0 is computed and for each statement where $UC^{\mathrm{t}} \neq \mathbf{0}$ does not hold, the cases

$$U_i C^{\mathrm{t}} \geq 1 \quad \text{or} \quad U_i C^{\mathrm{t}} \leq -1 \tag{5}$$

are considered for each row i of U in turn. The rows of U are also normalized to favor schedules with zero values for later schedule coefficients and a positive value for the first schedule coefficient involved.

Once a solution has been found, backtracking continues, but the search is narrowed to "significantly better" solutions. In practice, this means that a solution with a parametric bound on the distances over proximity schedule constraints may be replaced by one with a non-parametric bound and that a solution with a non-zero bound may be replaced by one with a zero bound.

2.3 Spatial Locality

Wolf and Lam (1991a) define the directions of *self-temporal reuse* to be those in $\ker F$ and those of *self-spatial reuse* to be those in $\ker G$, with G as in (1). They also consider group-spatial reuse between different references to the same array from the same schedulable unit (in their case, a loop nest), but these rarely bring any additional directions (Wolfe 1996). They partition the original loop iterators into those that do not appear in reuse directions and those that do, and apply their SRP algorithm (Wolf and Lam 1991b) to both groups of iterators separately. Tiling is performed on the innermost loops, but reorderings of the point loops are not considered because they do not affect reuse. For the purpose of consecutivity, however, such reorderings are important.

Anderson et al. (1995), Kandemir, Ramanujam, and Choudhary (1999), and Vasilache et al. (2012) show that data layout transformations are also important. This means that spatial locality may be considered in array dimensions other than the innermost (in `C` layout). This paper does not consider such transformation and the related work is reformulated in terms of the innermost dimension when appropriate.

Kandemir, Ramanujam, and Choudhary (1999) apply both data layout transformations and a loop nest transformation, i.e., with a single transformation matrix. They require the innermost transformed loop iterator to only appear in the innermost array index expression and to appear there on its own with coefficient one. In later steps, they also allow self-temporal reuse in the innermost index expression and self-spatial reuse in the second innermost expression. That is, they try to have $F_i T^{-1}$ equal to

$$\begin{bmatrix} X & \mathbf{0} \\ \mathbf{0}^{\mathrm{t}} & 1 \end{bmatrix} \qquad \text{or} \qquad \begin{bmatrix} X & \mathbf{0} & \mathbf{0} \\ \mathbf{0}^{\mathrm{t}} & 1 & 0 \\ \mathbf{0}^{\mathrm{t}} & 0 & 0 \end{bmatrix} \tag{6}$$

and use these constraints to fix elements of T^{-1}. The authors explain that their criterion is stronger than strictly needed for spatial locality. In fact, it is sufficient for consecutivity and forms the basis of the consecutivity objective of Section 3.1.

Kandemir, Ramanujam, Choudhary, and Banerjee (2001) compute a loop nest transformation by picking the last column of T^{-1} from $\ker F$ for self-temporal reuse or from $\ker G$ for self-spatial reuse. In the first case, they also try and pick the second to last column of T^{-1} from $\ker G$. The approach of the present paper computes multiple T matrices, one for each statement, row by row and therefore first needs to transform the objective into constraints on those rows.

Bastoul and Feautrier (2004) described how to obtain a (partial) schedule with a prescribed null-space. They take self-temporal reuse as an example, where the null-space is picked from $\ker F$, but explain that self-spatial reuse can be handled in a similar way (where the null-space would be $\ker G$). They start from a basis T that has the selected vector in its null-space and successively replace individual rows by linear combinations of the rows by solving for optimal linear combinations. A direct application to the `isl` scheduler is not straightforward because its scheduling problem is formulated in terms of the original schedule coefficients and not in terms of these linear combinations. However, computing a linear combination of the rows of T is the same as computing a row with a null-space that includes that of T. It is therefore sufficient to add some equality constraints (corresponding to the orthogonal complement of T) on the schedule coefficients. This latter method will be used in Section 3.3.

Bondhugula, Hartono, et al. (2008, Section 5.4) mention the possibility of optimizing spatial locality by performing interchanges in the intra-tile bands, but do not provide any details. Support for these intra-tile interchanges for spatial locality was later made available in `pluto` version `0.8.1-53-g63b86f2` (2012). Trifunovic, Nuzman, et al. (2009) perform an exhaustive search over all loop permutations (at the AST level) and pick the best based on a cost model. There is no mention of any validity check.

Whereas Bastoul and Feautrier (2004) compute a partial schedule that is orthogonal to a selected element of $\ker G$ (in case of self-spatial reuse), Vasilache et al. (2012) compute an outer schedule T_1 (all but the final row) that is orthogonal to any element of $\ker G$, i.e.,

$$\ker T_1 \subseteq \ker G. \tag{7}$$

In other words, the rows of G need to be linear combinations of the rows of T_1. Note that since $\ker T_1$ consists of multiples of the last column of T^{-1}, this criterion is essentially the same as that expressed by the left part of (6), except that this latter criterion also involves constraints derived from H, ensuring non-temporal spatial locality. Just like Bastoul and Feautrier (2004), they try to obtain a partial schedule that consists of linear combinations of some initial matrix (here, G), but they only do so for schedule rows where it is strictly needed to ensure that G is a linear combination of T_1. It is not entirely clear from the description what happens in other cases. In particular, it is not clear if they prevent H from being a linear combination of the rows of T_1, which is allowed

by criterion (7), but which would prevent consecutivity. Unlike Bastoul and Feautrier (2004), the linear combination is part of an optimization criterion and not a hard constraint. In particular, a constraint is added that makes the schedule row equal to $G\boldsymbol{\lambda}$, with $\boldsymbol{\lambda}$ additional unconstrained variables. This equality constraint is encoded as a pair of inequality constraints that are only enforced if the corresponding decision variable is set. Note that while a one-shot scheduler is being used, it is called several times, each time fixing an additional row of the schedule.

Kong et al. (2013) have similar objectives to those of the present paper as they try and obtain stride-1 and stride-0 accesses. Being based on a one-shot scheduler (in their case only called once), this approach belongs to a different class of approaches. Moreover, it is exclusively based on which statement indices appear in index expressions and schedule rows and not on any linear combinations of those statement indices, thereby missing some opportunities, e.g., in case of an access `A[j][j - i]` with loop iterators `i` and `j`. Finally, the encoding in the ILP problem only seems to create favorable conditions where stride-1 or stride-0 accesses may appear rather than necessarily enforcing such accesses. See Verdoolaege and Isoard (2017, Section 2.11) for further details.

When using the `isl` scheduler, a potential approach would be to exploit proximity schedule constraints to try and bring accesses to consecutive elements close to each other. However, a naive implementation would have these proximity schedule constraints compete with those for temporal locality, where one group may drown out the other. They would continue to be enforced within a band even if outer members already prevent spatial locality, potentially steering the scheduler in the wrong direction. In case of temporal reuse in an array reference, a naive formulation would result in many interrelated instances, possibly causing infeasibility of the ILP problem (Verdoolaege and Janssens 2017, Section 6.6.3). Finally, proximity schedule constraints are not directional. That is, they only bring statement instances close to each other, but do not ensure that one appears before the other, which is required for consecutivity.

The approach of Zinenko et al. (2018) attempts to resolve some of these issues by introducing specialized *spatial* proximity schedule constraints, but focuses on general spatial locality and not specifically on consecutivity. The constraints are derived from pairs of statement instances that access adjacent elements of an array, with additional filtering to avoid some problematic cases. Due to this filtering, many of the resulting pairs have a strong correspondence with the intra-statement consecutivity schedule constraints introduced below, in the sense that a satisfied intra-statement consecutivity schedule constraint means that the corresponding spatial proximity schedule constraints will have a zero distance in the outer dimensions and a distance of one (in absolute value) in some inner dimension. However, since

```
void transpose(int N,
    __pencil_consecutive float A[N][N],
    __pencil_consecutive float C[N][N])
{
    float tmp[N][N];

    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
S:          tmp[i][j] = A[i][j];
T:          C[j][i] = tmp[i][j];
        }
}
```

Listing 1. Input file transpose-n.c

spatial proximity schedule constraints are not tailored to optimizing consecutivity, they do not distinguish between accessing elements in increasing order and accessing them in decreasing order. They also make no distinction between directions that should be in outer dimensions and directions that are independent, meaning they will favor putting the independent directions in non-innermost positions, while the handling of intra-statement consecutivity schedule constraints below does not imply such a preference. If a choice needs to be made, then this also means that they cannot tell the difference between allowing one of the independent directions to be innermost (while still achieving spatial locality) and allowing one of the outer index expression directions to be innermost (thereby failing to achieve spatial locality).

Spatial proximity schedule constraints derived from uniformly generated references are grouped together and the groups are sorted according to their expected influence. During the computation of a schedule row, the maximal schedule distance between pairs of elements in each group is minimized in turn. Each group with a non-zero distance is removed from consideration for any subsequent schedule rows. The group based minimization means that a sequence of additional variables in the ILP problem is added for each group. In contrast, the approach of this paper does not introduce any additional variables and does not involve any minimization, in particular of distances between pairs of instances that may in the end turn out to correspond to a failed spatial locality constraint. On the other hand, the approach of this paper may commit too eagerly to some constraints and miss other opportunities, although this effect is mitigated by the preprocessing of Section 3.4. It is therefore difficult to predict which approach will produce the best results with the least amount of effort. A detailed comparative experimental evaluation is left to future work.

3 Intra-Statement Consecutivity

Consider the matrix transpose code in Listing 1. The code assumes that arrays `A` and `C` are stored in external memory and would therefore benefit from consecutive accesses. This objective is expressed through the `__pencil_consecutive` annotation. The `tmp` array is stored locally and does not need to be considered for consecutivity. It is clear that in order to obtain consecutive accesses, the loop nest needs to be distributed and loop interchange needs to be applied to the second loop nest. This section describes the schedule constraints that allow information about desired consecutivity to be communicated to the scheduler as well as their handling by the scheduler. Note that `PPCG` has an optimization that groups chains of statements accessing the same memory elements into larger statements in the input to the scheduler (Verdoolaege and Janssens 2017, Section 7.4). This optimization is disabled if consecutivity is involved as it would group statements such as `S` and `T` in the example into a single statement based on their accesses to `tmp`, preventing any loop distribution.

3.1 Objective

Given the linear part of affine index expression F, split into an outer part G and an inner part H as in (1), the objective is to obtain a transformed array access that has the innermost transformed loop iterators appear with coefficient 1 in the innermost index expressions (in the same order) and not in any outer index expressions. Furthermore, some additional transformed loop iterators may be mixed in with the aforementioned innermost transformed loop iterators as long as this additional set does not appear in any index expressions. That is, $F\,T^{-1}$ is of the form

$$\begin{matrix} \\ d-f \\ \\ \\ f \end{matrix}\!\!\left\{\begin{bmatrix} & 0 & & 0 & & \cdots & & 0 & \\ & \vdots & & \vdots & & & & \vdots & \\ A & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ & 1 & & 0 & & \cdots & & 0 & \\ & & & 1 & & \ddots & & \vdots & \\ & & L & & & \ddots & & 0 & \\ & & & & & & & 1 & \end{bmatrix}\right., \tag{8}$$

for some A and L. This is a minor generalization of criterion (6) of Kandemir, Ramanujam, and Choudhary (1999), where H may have a number of rows f that is greater than one, where the number of zero columns may be any number greater than or equal to zero and where both L and the lower part of A do not need to be all zeros. Allowing multiple rows in H is mostly of interest for combining multiple references as described in Section 3.4 below. Let the number of zero columns be t. They allow for t directions of temporal locality *in addition* to the f directions of consecutivity. Criterion (7), on the other hand, considers temporal locality as a special case of spatial locality. A straightforward extension to multiple inner index expressions would therefore try to achieve at most f levels of spatial locality. Note that criterion (8) above only requires a transformed loop iterator appearing with a coefficient 1 in some index expression to not appear in any *earlier* index expression. It can appear in *later* index expressions since the iterator may be considered to be fixed when evaluating the consecutivity of this later index expression.

Criterion (8) can only be satisfied if F satisfies some criteria. In particular, H needs to be linearly independent and H needs to be linearly independent of G. The matrix G itself does not need to be linearly independent, but it may be replaced by a basis of its rows without affecting consecutivity.

For the code in Listing 1,

$$F_S = \left[\begin{array}{cc} G_S \\ \hline H_S \end{array}\right] = \left[\begin{array}{cc} 1 & 0 \\ \hline 0 & 1 \end{array}\right] \quad \text{and} \quad F_T = \left[\begin{array}{cc} 0 & 1 \\ \hline 1 & 0 \end{array}\right], \tag{9}$$

and transformation matrices

$$T_S = \left[\begin{array}{cc} 1 & 0 \\ \hline 0 & 1 \end{array}\right] \quad \text{and} \quad T_T = \left[\begin{array}{cc} 0 & 1 \\ \hline 1 & 0 \end{array}\right] \tag{10}$$

can be found that both result in an identity $F\,T^{-1}$, where A consists of the first column and there are no zero columns.

3.2 Strategy

The strategy for trying to achieve objective (8) in a scheduler that computes the schedule row by row consists of two main phases. The first phase tries to obtain the zeros in the top right part of criterion (8), while the second phase tries to obtain the lower-triangular matrix in the bottom right. Once these goals have been obtained, consecutivity has been achieved and the scheduler is free to choose the remaining schedule rows based on other criteria. Let the linear part of the schedule of the corresponding statement be subdivided accordingly into rows T_1, T_2 and T_3. Note that T_0 still refers to the linear part of the schedule computed so far.

The first phase is essentially the same as the strategy of Vasilache et al. (2012), picking outer rows T_1 that have G as a linear combination, i.e., $\ker T_1 \subseteq \ker G$ (7), except that it also needs to take into account that it will be followed by a second phase that will try and pick T_2 to be equal to H (up to some linear combinations of earlier rows). This means that the rows of T_1 should be chosen to be linearly independent of H. In fact, since each schedule row needs to be linearly independent of all previous rows, it should be chosen to be linearly independent of H and T_0 combined. Let

$$r_1 = \operatorname{rank} T_0 \quad \text{and} \quad r_2 = \operatorname{rank} \begin{bmatrix} T_0 \\ G \end{bmatrix}, \tag{11}$$

then the first phase continues until r_1 becomes equal to r_2, at which point G is a linear combination of T_0.

The best way to make progress during the first phase is to pick schedule rows that are linear combinations of G. This

Constraints introduced when $r_1 < r_2$
$\left(C = X \begin{bmatrix} T_0 \\ G \end{bmatrix} \wedge C \neq Y \begin{bmatrix} T_0 \\ H \end{bmatrix}\right) \vee C \neq Y \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} \vee C = XT_0$
Constraints introduced when $r_1 = r_2 \wedge h < f$
$C = H_h + X \begin{bmatrix} T_1 \\ H_< \end{bmatrix} \quad \vee C \neq Y \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} \vee C = XT_0$
Constraints introduced when $h = f$
none

Table 2. Constraints on (the linear part of) the next schedule row C introduced at different stages of the consecutivity constraints handling process. $C = X\,M$ is short for C being a linear combination of the rows of M. $C \neq Y\,M$ is short for C being linearly independent of the rows of M. $H_<$ contains the rows of H with index smaller than h.

will increase r_1 without also increasing r_2. Each such schedule row may also include contributions from the previously computed schedule rows. To ensure that the row contains a non-zero contribution from G, it is explicitly enforced to be linearly independent of T_0 and H. Alternatively, it is also possible to pick a row that is linearly independent of the entire F (and T_0). This will increase both r_1 and r_2. Such a choice is only possible if the total rank of T_0 and F combined is smaller than the statement dimension n. Finally, for lower-dimensional statements that still satisfy condition (4), allowing the next schedule row to be linearly dependent on T_0, it is also possible to pick a linear combination of T_0. Such a choice increases neither r_1 nor r_2. It also does not prevent a successful application of the second phase because each row of T_0 is linearly independent of H at this stage. These different options are summarized in the top part of Table 2.

In the second phase, the successive schedule rows need to be made equal to successive rows of H. For each intra-statement consecutivity schedule constraint, the scheduler therefore keeps track of the number of schedule rows h that have been made equal to the first of the f rows of H. Note that the schedule rows do not need to be exactly the same as the rows of H, but may instead also have contributions from earlier schedule rows. Since these earlier rows are linearly independent of the current row of H, the contribution of this row is not canceled out and the resulting schedule row is linearly independent of both the outer schedule rows and of subsequent rows of H. As in the first phase, schedule rows that are linearly independent of the entire F, or linear combinations of earlier rows are also allowed, when applicable. However, such intermediate rows, which correspond to the zero columns in criterion (8), are not allowed in the linear combinations that may be added to subsequent rows made equal to rows in H in order to ensure that those columns are entirely zero. The corresponding constraint in Table 2 therefore refers to T_1 (which does not include these rows) rather than to T_0 (which does include these rows). As soon as all rows of H have been handled, i.e., $h = f$, consecutivity has been achieved and no more consecutivity based constraints on schedule coefficients are introduced during the computation of subsequent schedule rows.

For statement T in Listing 1, the first phase tries to find a suitable linear combination of $\begin{bmatrix} 0 & 1 \end{bmatrix}$ that is linearly independent of $\begin{bmatrix} 1 & 0 \end{bmatrix}$, say $\begin{bmatrix} 0 & 1 \end{bmatrix}$ itself, while the second phase tries to construct a schedule row that is equal to $\begin{bmatrix} 1 & 0 \end{bmatrix}$ plus some linear combination of the previous row $\begin{bmatrix} 0 & 1 \end{bmatrix}$.

As soon as any of the constraints imposed on the schedule coefficients fails to be satisfied by the solution, the corresponding intra-statement consecutivity schedule constraint is removed from consideration. Note that in the current implementation, these constraints are only imposed by the Pluto scheduler. In cases where a step of the Feautrier scheduler ends up getting performed, the computed schedule row may therefore not satisfy those constraints. If at any stage,

$$f - h > \operatorname{rank} \begin{bmatrix} T_0 \\ G \\ H \end{bmatrix} - r_2, \tag{12}$$

i.e., there are not enough linearly independent rows in H left, then the corresponding intra-statement consecutivity schedule constraint is also removed from consideration.

If multiple intra-statement consecutivity schedule constraints were specified for the same statement, then a constraint on the schedule coefficients is constructed for each intra-statement consecutivity schedule constraint according to the rules in Table 2. However, the solver is instructed to first try and satisfy the constraint on the schedule coefficients corresponding to the first intra-statement consecutivity schedule constraint on the statement and to only consider the one corresponding to a later intra-statement consecutivity schedule constraint when the one corresponding to the previous one cannot be satisfied. Disjuncts that also appear in the constraint on the schedule coefficients corresponding to previous intra-statement consecutivity schedule constraints are therefore dropped since they are already known to be unsatisfiable by the time they would be reconsidered. In particular, the linear dependence disjunct is independent of the intra-statement consecutivity schedule constraint and is therefore only considered for the first intra-statement consecutivity schedule constraint. If all disjuncts corresponding to an intra-statement consecutivity schedule constraint are duplicates of disjuncts corresponding to previous ones on the same statement, then the entire disjunction is dropped.

3.3 Solution

The elementary constraints in Table 2 on the preceding page are of a form that is similar to that of the linear independence constraint (3). They are also handled in a similar way. Some of these constraints are also linear independence constraints, but they involve more rows, meaning that the complement U has fewer rows and that therefore fewer cases need to be considered during backtracking. Some impose linear combinations, in which case the same orthogonal complement U is computed, but the (linear) constraints $UC^{\mathrm{t}} = \mathbf{0}$ are imposed, which do not require any backtracking. When both types are combined, i.e., $UC^{\mathrm{t}} = \mathbf{0}$ and $VC^{\mathrm{t}} \neq \mathbf{0}$, then V can be replaced by $V' = V \setminus U$. It is the rows of this V' that are normalized to favor schedules with zero values for later schedule coefficients and a positive value for the first schedule coefficient involved.

The remaining constraint is of the form $C = A + XM$, with A linearly independent of M. Let U be the orthogonal complement of $[M; A]$. Then $UC^{\mathrm{t}} = \mathbf{0}$. Moreover, if U' is the orthogonal complement of M, then the constraint also implies $U'C^{\mathrm{t}} = U'A^{\mathrm{t}}$. Since $UC^{\mathrm{t}} = \mathbf{0}$, the matrix U' in this last condition can be replaced by $U'' = U' \setminus U$. The combined constraint on the schedule coefficients C is therefore

$$\begin{bmatrix} U \\ U'' \end{bmatrix} C^{\mathrm{t}} = \begin{bmatrix} \mathbf{0} \\ U''A^{\mathrm{t}} \end{bmatrix}. \tag{13}$$

For example. during the second phase of the computation for statement `T` in Listing 1, $A = \begin{bmatrix} 1 & 0 \end{bmatrix}$ and $M = \begin{bmatrix} 0 & 1 \end{bmatrix}$, meaning that U has zero rows and $U'' = U' = \begin{bmatrix} 0 & 1 \end{bmatrix}$. The constraint therefore specializes to

$$\begin{bmatrix} 1 & 0 \end{bmatrix} C^{\mathrm{t}} = \begin{bmatrix} 1 \end{bmatrix}, \tag{14}$$

allowing any solution of the form $C = \begin{bmatrix} 1 & x \end{bmatrix}$.

As in the case of the linear independence constraint (3), a constraint on the schedule coefficients is only enforced if it is not already satisfied by the current ILP solution. However, the backtracking search is modified in several ways.

First, in contrast to constraint (3), which is required to produce a schedule with linearly independent rows, the constraints on schedule coefficients derived from intra-statement consecutivity schedule constraints are *optional*. That is, a schedule not satisfying such a constraint is still a valid, if suboptimal, schedule. Besides the $2n$ cases of the form (5), the search therefore also needs to consider the case where the constraint is not imposed, but is *disabled* instead. The constraint needs to be disabled to avoid the constraint being considered at nested levels in the search. It is re-enabled when backtracking out of the level that disabled the constraint. Optional constraints are considered before required constraints, i.e., those that are required for linear independence of the schedule. Note that the optional constraints that involve some linear independence subsume the required linear independence constraints on the same statement. That is, when this part of the optional constraint is being enforced, the corresponding required constraint will not be triggered.

Second, the new types of constraints have a *fixed* part that is enforced in all the linear independence cases (5), but not in the case where the constraint is disabled. If the constraint does not involve a linear independence part, then there are two states, one where the fixed part is enforced and one where the constraint is disabled.

Third, the constraint may be *disjunctive*, in which case it is only triggered when all of the disjuncts are violated by the current ILP solution. When it is triggered, the first disjunct that has not been disabled at previous levels of the backtracking search is enforced. If this does not result in a solution, then the disjunct will be disabled and the next disjunct will be enforced until all disjuncts have been considered.

Finally, a (possibly disjunctive) constraint may be *conditional* on the previous (possibly disjunctive) constraint. In this case, the entire disjunctive constraint is ignored until the final disjunct of the previous constraint has been disabled.

Whenever a solution has been found, all optional constraints satisfied by the solution are turned into required constraints, ensuring that any improved solution has at least the same satisfied intra-statement consecutivity schedule constraints. This could be further refined to enforce that the *number* of satisfied intra-statement consecutivity schedule constraints does not decrease. Note, in particular, that the backtracking search is currently not continued for the purpose of increasing the number of satisfied intra-statement consecutivity schedule constraints, but only for obtaining a "significantly better" solution, as described in Section 2.2.

3.4 Multiple References

If a statement has multiple accesses to arrays marked consecutive, then intra-statement consecutivity schedule constraints can be constructed for each access individually. However, for each statement, the `isl` scheduler will only try to optimize one of them, as it expects the user to construct combined intra-statement consecutivity schedule constraints that cover multiple such accesses. A prototype implementation is available in `PPCG`.

For the purpose of constructing consecutivity constraints, `PPCG` first prunes references that access multiple array elements per statement instance, that are not purely affine or that have an innermost index expression that is a linear combination of the outer index expressions, e.g., `A[i][i]`. If any reference to an array is pruned, then any other reference to the same array from the same statement is also pruned as it is impossible to achieve consecutivity for such arrays. If there are multiple references to the same array from the statement, then they are first combined into a single reference using the procedure described below. If this fails to produce a singe reference that covers all original references to the array, then these references are pruned as well.

```
void matmul(int N, int M, int K,
    __pencil_consecutive float A[N][K],
    __pencil_consecutive float B[K][M],
    __pencil_consecutive float C[N][M])
{
  __builtin_assume(K > 0);
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j) {
S:    C[i][j] = 0;
      for (int k = 0; k < K; ++k)
T:      C[i][j] += A[i][k] * B[k][j];
    }
}
```

Listing 3. Input file matmul-n.c

Multiple references in the same statement (to the same array in a first phase and to distinct arrays in a second phase) are combined into one or more composite references by successively combining pairs of intra-statement consecutivity schedule constraints (G_1, H_1) and (G_2, H_2), with H_i linearly independent and also linearly independent of G_i, into a single intra-statement consecutivity schedule constraint that satisfies the same properties (Verdoolaege and Isoard 2017, Section 3.2.1). In particular,

1. if $H_1 = H_2$, then the two constraints can be combined by setting
$$G = \begin{bmatrix} G_1 \\ G_2 \end{bmatrix} \qquad H = H_1, \tag{15}$$
provided $H_1 = H_2$ is linearly independent of G.
2. If H_2 is linearly independent of $[F_1; G_2]$, then the two constraints can be combined by setting
$$G = \begin{bmatrix} G_1 \\ G_2 \setminus F_1 \end{bmatrix} \qquad H = \begin{bmatrix} H_1 \\ H_2 \end{bmatrix}. \tag{16}$$

The final result is a list of possibly composite references with those that cover more original references appearing first.

Consider the code in Listing 3, where the assumption on K only serves to simplify the output code in Listing 4. Statement T contains three accesses to arrays that should all be accessed consecutively. The constraints for the individual accesses are

$$F_A = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right], F_B = \left[\begin{array}{ccc} 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right], F_C = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 1 & 0 \end{array}\right]. \tag{17}$$

Since $H_B = H_C$ is linearly independent of the combination of G_B and G_C, the first form of combination can be applied, resulting in

$$F_{BC} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}. \tag{18}$$

```
for (int c0 = 0; c0 < N; c0 += 1)
  for (int c1 = 0; c1 < K; c1 += 1)
    for (int c2 = 0; c2 < M; c2 += 1) {
      if (c1 == 0)
        C[c0][c2] = 0;
      C[c0][c2] += A[c0][c1] * B[c1][c2];
    }
```

Listing 4. Transformed code for the input in Listing 3

Now, H_{BC} is linearly independent of

$$\left[\begin{array}{c} F_A \\ \hline G_{BC} \end{array}\right] = \left[\begin{array}{ccc} 1 & 0 & 0 \\ \hline 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{array}\right] \tag{19}$$

and so the second form of combination can be applied. In this case, G_{BC} is a linear transformation of F_A and so $G_{BC} \setminus F_A$ has zero rows. The result of the combination is therefore

$$F_{ABC} = \left[\begin{array}{ccc} 1 & 0 & 0 \\ \hline 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right]. \tag{20}$$

Satisfaction of this single constraint ensures consecutivity for all three accesses and corresponds to the loop order (i, k, j). The transformed code satisfying the constraint is shown in Listing 4. Note that for the accesses to B and C, the innermost loop iterator only appears in the last index expression and does so with coefficient one. For the access to A, the innermost iterator does not appear at all in the index expressions, while the second innermost iterator only appears in the last index expression and does so with coefficient one.

This mechanism for combining information from different array references is similar to the way Kandemir, Ramanujam, and Choudhary (1999) take multiple array references into account. However, they derive additional elements of the inverse transformation matrix directly by examining each array reference in turn, while the mechanism described in this section first collects information from multiple array references into one or more composite array references that are then later used as a whole during the schedule construction. Note that the combined constraint may end up being discarded by the scheduler if it conflicts with the validity or coincidence constraints. PPCG therefore also imposes consecutivity constraints that only cover some or even one array, with those that cover most arrays placed first.

3.5 Incremental Scheduling

As mentioned in Section 2.2, the isl scheduler first computes a band schedule in each component separately. If any of the components (partially) satisfies some intra-statement consecutivity schedule constraints, then the scheduler needs to take care not to violate these intra-statement consecutivity

schedule constraints when combining the components. In particular, if $h > 0$ for some statement in a component, i.e., if at least one schedule row has been set equal to a row in H, then an intra-statement consecutivity schedule constraint is introduced on the component that ensures that this row and all subsequent rows are unaffected (apart from possibly mixing in earlier rows). Let p be the position of the schedule row in the component schedule that corresponds to the first row of H. If this row belongs to an outer band, then set $p = 0$. Let v be the dimension of the component schedule. Then an intra-statement consecutivity schedule constraint is introduced with identity $F = I$ and f set to $v - p$.

Consider once more the code shown in Listing 3. The combined intra-statement consecutivity schedule constraint for statement T has been derived before (20). For S, the intra-statement consecutivity schedule constraint is

$$F_{\mathtt{S}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{21}$$

The two statements are first scheduled individually, resulting in the schedules

$$\{\, \mathtt{S}[i,j] \to \mathtt{C0}[i,j] \,\} \quad \text{and} \quad \{\, \mathtt{T}[i,j,k] \to \mathtt{C1}[i,k,j] \,\}. \tag{22}$$

In both cases, the second schedule dimension corresponds to the first row of the respective H matrices. The intra-statement consecutivity schedule constraints on the clusters C0 and C1 are therefore

$$F_{\mathtt{C0}} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \quad \text{and} \quad F_{\mathtt{C1}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}. \tag{23}$$

The proximity schedule constraints between the clusters determine how the clusters should be combined, resulting in the cluster schedule

$$\{\, \mathtt{C0}[a,b] \to [a,0,b]; \mathtt{C1}[a,b,c] \to [a,b,c] \,\}, \tag{24}$$

satisfying the cluster intra-statement consecutivity schedule constraints. Combined with the individual schedules (22), the final result is

$$\{\, \mathtt{S}[i,j] \to [i,0,j]; \mathtt{T}[i,j,k] \to [i,k,j] \,\}. \tag{25}$$

For the code in Listing 1, the cluster intra-statement consecutivity schedule constraints steer the scheduler towards a schedule that places the second cluster at an offset N in the outer dimension, effectively causing the entire second statement to be executed after the first statement. Since this cluster schedule does not optimize any proximity schedule constraints, it is rejected. This preserves the two separate bands, which are then also scheduled one after the other, resulting in loop distribution.

4 Inter-Statement Consecutivity

The previous section only considered consecutivity within a statement. It can also be useful to try and bring instances of distinct statements that access consecutive array elements close to each other. Consider, for example, the code in Listing 5, where only B is marked consecutive. Different statements access different slices of the array and the goal is to obtain code as in Listing 6, with loop interchange and an appropriate interleaving of the statement instances.

```
void unroll(int N, int M, float A[N][M],
    __pencil_consecutive float B[M][N])
{
  __builtin_assume(N%2 == 0 && M%2 == 0);
  for (int i = 0; i < N; i += 2) {
    for (int j = 0; j < M; j += 2) {
S00:  B[j + 0][i + 0] = A[i + 0][j + 0];
S01:  B[j + 1][i + 0] = A[i + 0][j + 1];
S10:  B[j + 0][i + 1] = A[i + 1][j + 0];
S11:  B[j + 1][i + 1] = A[i + 1][j + 1];
    }
  }
}
```

Listing 5. Input file unrolled-n.c

```
for (int c0 = 0; c0 < M - 1; c0 += 2) {
  for (int c1 = 0; c1 < N - 1; c1 += 2) {
    B[c0][c1] = A[c1][c0];
    B[c0][c1 + 1] = A[c1 + 1][c0];
  }
  for (int c1 = 0; c1 < N - 1; c1 += 2) {
    B[c0 + 1][c1] = A[c1][c0 + 1];
    B[c0 + 1][c1 + 1] = A[c1 + 1][c0 + 1];
  }
}
```

Listing 6. Transformed code for the input in Listing 5

4.1 Objective

An inter-statement consecutivity schedule constraint consists of a pair of intra-statement consecutivity schedule constraints for two distinct statements S_1 and S_2, with reference matrices F_1 and F_2 with an equal number of final rows f, along with a binary relation between the two statements, specifying instances that access consecutive elements at the f-th innermost index expression. In the PPCG implementation, such pairs are only constructed for references that access elements once, only between pairs of reads or pairs of writes and only if there is no intermediate kill. See Verdoolaege and Isoard (2017, Section 3.2.2) for details. The objective is for these pairs of instances to be executed at a distance of 1 along the schedule dimension that is aligned with the f-th innermost index expression. That is, the schedule T mapping S_1 and S_2 to a common space should have

linear parts T_1 for S_1 and T_2 for S_2 that both satisfy the corresponding intra-statement consecutivity schedule constraint with the same number of zero columns t and for each pair of instances $(\mathbf{x}_1, \mathbf{x}_2) \in R$, there should be some $\mathbf{z} \in \mathbb{Z}^{f+t-1}$ such that

$$T(\mathbf{x}_2) - T(\mathbf{x}_1) = \begin{bmatrix} \mathbf{0} & 1 & \mathbf{z} \end{bmatrix}^{\mathrm{t}}. \tag{26}$$

4.2 Strategy

The two referenced intra-statement consecutivity schedule constraints are treated like any other intra-statement consecutivity schedule constraint. Additional constraints are added depending on the state of handling these intra-statement consecutivity schedule constraints. In particular, if either of the two intra-statement consecutivity schedule constraints has failed, then the inter-statement consecutivity schedule constraint is considered to have failed as well. Furthermore, additional constraints are only added when the number h_i of schedule rows made equal to H_i (plus some linear combination) is still zero for both intra-statement consecutivity schedule constraints. Specifically, as long as either of the intra-statement consecutivity schedule constraints is still in the process of constructing T_1, the schedule distance of the constructed schedule row is set to 0, i.e.,

$$\forall \mathbf{a} \to \mathbf{b} \in R : f(\mathbf{b}) - f(\mathbf{a}) = 0. \tag{27}$$

As soon as both have completed the linear schedule to cover G_i, the distance is set to 1, i.e.,

$$\forall \mathbf{a} \to \mathbf{b} \in R : f(\mathbf{b}) - f(\mathbf{a}) = 1. \tag{28}$$

In accordance with (26), this distance-1 constraint is only applied for a single schedule row. The inter-statement consecutivity schedule constraint is ignored for any subsequent schedule rows.

Any two statements connected by an inter-statement consecutivity schedule constraint are considered to belong to the same component for the purpose of incremental scheduling. This ensures that there are no inter-statement consecutivity schedule constraints across components, but reduces the advantage of the incremental scheduling. No special treatment is required to preserve the satisfied inter-statement consecutivity schedule constraints inside the resulting components since the relative positions are not modified and the referenced intra-statement consecutivity schedule constraints are already preserved, ensuring that a distance-1 direction does not get mixed in with distance-0 directions.

4.3 Solution

Constraints (27) and (28) are imposed by requiring the schedule coefficients (minus one) to be a linear combination of the affine hull of R, in particular by adding an (optional) constraint on these coefficients to be orthogonal to the orthogonal complement of the coefficients of this affine hull. See Verdoolaege and Isoard (2017, Section 3.4.3) for details.

Just like any other optional schedule coefficient constraint with a fixed value and no linear independence, this constraint can be in two states after it has been activated: the fixed value is enforced, or it has been disabled. In principle, the fixed value derived from an inter-statement consecutivity schedule constraint should only be enforced if the constraints of the corresponding intra-statement consecutivity schedule constraints are satisfied, but the current implementation does not explicitly check for this condition as it was not needed in the limited experiments performed with this feature.

5 Local Rescheduling

Since consecutivity schedule constraints take precedence over proximity schedule constraints, setting the former during the computation of a global schedule may cause some pairs of statement instances to be moved apart. For example, they cause the loop in Listing 1 to be fully distributed, resulting in higher memory requirements for `tmp` (after memory requirement optimization, Darte et al. 2005). The prototype `PPCG` implementation therefore also allows consecutivity schedule constraints to only be considered in a rescheduling of the point band after tiling. This causes the statements to be distributed inside the tile, resulting in modest memory requirements for `tmp` (the size of a tile) and consecutivity within a tile. Verdoolaege and Isoard (2017, Section 4) describe the rescheduling support in `isl`.

6 Conclusion and Future Work

Consecutivity of accesses can be exploited in performance-improving burst accesses. This paper describes how to steer the scheduler towards consecutivity both within and across statements within a particular approach for polyhedral compilation, involving both specific consecutivity support in the core scheduler and an appropriate use of this support by the polyhedral compiler. In the current implementation, priority is given to coincidence over consecutivity, meaning that this support is mostly useful in cases where the innermost loops end up being fully parallel and where, without consecutivity constraints, the scheduler would not have good criteria to schedule those inner loops.

Given that the current scheduler tries to place parallel schedule rows outermost, while consecutive rows are placed innermost, further modifications may be required for vectorization. The scheduler would have to either allow consecutivity schedule constraints to overrule a choice for an (outer) parallel row, or to compute the innermost rows (that are both parallel and consecutive) first. Allowing consecutivity schedule constraints to take priority over coincidence schedule constraints could be achieved by not adding the schedule coefficient constraints corresponding to the latter directly to the ILP, but instead to add them as a single optional schedule coefficient constraint after those corresponding to consecutivity schedule constraints. This is left for future work.

```
Initialize empty band
Coincidence ← true
while band not full-dimensional do
    Set up ILP
    Solve ILP (Algorithm 2)
    if no solution then
        if Coincidence then
            Coincidence ← false
            continue
        else
            break
    Add ILP solution to current band
return current band
```

Algorithm 1: Compute band schedule for component

```
Add optional schedule coefficient constraints (Table 2)
Add linear independence schedule coefficient
  constraints
Compute constrained lexmin (Algorithm 3)
Update consecutivity data
```

Algorithm 2: Solve ILP

Acknowledgments

This research was supported by Xilinx.

A Sketch of `isl` scheduling algorithm

This appendix provides a sketch of the core `isl` scheduling algorithm. In particular, Algorithm 1 shows how a schedule band is computed for a strongly connected component in the statement-level schedule constraint graph. This algorithm calls Algorithm 2 to solve the ILP problem it creates, which in turn calls Algorithm 3 to compute a constrained lexicographically minimal solution. Some details such as how constraints are removed during backtracking are not shown to avoid clutter. Algorithm 1 also abstracts away the support for live-range reordering (Verdoolaege and Cohen 2016). The highlighted lines are specific to the handling of consecutivity constraints. The step "Find first violated constraint" of Algorithm 3 deals with disjunctive and conditional schedule coefficient constraints.

```
while level ≥ 0 do
    if entering level then
        if empty then
            Backtrack
        Find first violated constraint
        if no violated constraint then
            Update best solution
            if solution is optimal then
                return best solution
            Backtrack
        Set fixed part of constraint
    if all cases handled then
        if constraint disabled or not optional then
            Backtrack
        Disable constraint
    Force better solution than current best
    Move to next case ((5) or disabled)
    Enter next level
return best solution
```

Algorithm 3: Compute constrained lexmin

References

Anderson, Jennifer M., Saman P. Amarasinghe, and Monica S. Lam (1995). "Data and Computation Transformations for Multiprocessors". In: *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPOPP '95. Santa Barbara, California, USA: ACM, pp. 166–178. DOI: 10.1145/209936.209954.

Bastoul, Cédric and Paul Feautrier (Aug. 2004). "More Legal Transformations for Locality". In: *Euro-Par'10 International Euro-Par conference*. Vol. 3149. Lecture Notes in Computer Science. Pisa, pp. 272–283. DOI: 10.1007/978-3-540-27866-5_36.

Bielecki, Wlodzimierz, Marek Palkowski, and Piotr Skotnicki (Oct. 2017). "Generation of parallel synchronization-free tiled code". In: *Computing*. DOI: 10.1007/s00607-017-0576-3.

Bondhugula, Uday, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan (Apr. 2008). "Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model". In: *International Conference on Compiler Construction (ETAPS CC)*. DOI: 10.1007/978-3-540-78791-4_9.

Bondhugula, Uday, Albert Hartono, J. Ramanujam, and P. Sadayappan (2008). "A practical automatic polyhedral parallelizer and locality optimizer". In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '08. Tucson, AZ, USA: ACM, pp. 101–113. DOI: 10.1145/1375581.1375595.

Darte, Alain, Robert Schreiber, and Gilles Villard (2005). "Lattice-Based Memory Allocation". In: *IEEE Trans. Comput.* 54.10, pp. 1242–1257. DOI: 10.1109/TC.2005.167.

Feautrier, Paul (Oct. 1992a). "Some Efficient Solutions to the Affine Scheduling Problem. Part I. One-dimensional Time". In: *International Journal of Parallel Programming* 21.5, pp. 313–348. DOI: 10.1007/BF01407835.

Feautrier, Paul (Dec. 1992b). "Some Efficient Solutions to the Affine Scheduling Problem. Part II. Multidimensional Time". In: *International Journal of Parallel Programming* 21.6, pp. 389–420. DOI: 10.1007/BF01379404.

Grosser, Tobias, Armin Größlinger, and Christian Lengauer (2012). "Polly - Performing polyhedral optimizations on a low-level intermediate representation". In: *Parallel Processing Letters* 22.04. DOI: 10.1142/S0129626412500107.

Kandemir, Mahmut T., J. Ramanujam, and Alok N. Choudhary (Feb. 1999). "Improving Cache Locality by a Combination of Loop and Data Transformation". In: *IEEE Transactions on Computers* 48.2, pp. 159–167. DOI: 10.1109/12.752657.

Kandemir, Mahmut T., J. Ramanujam, Alok N. Choudhary, and Prithviraj Banerjee (Dec. 2001). "A Layout-Conscious Iteration Space Transformation Technique". In: *IEEE Transactions on Computers* 50.12, pp. 1321–1335. DOI: 10.1109/TC.2001.970571.

Kong, Martin, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and P. Sadayappan (2013). "When polyhedral transformations meet SIMD code generation". In: *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. PLDI '13. Seattle, Washington, USA: ACM, pp. 127–138. DOI: 10.1145/2491956.2462187.

Pouchet, Louis-Noël, Uday Bondhugula, Cédric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache (Jan. 2011). "Loop Transformations: Convexity, Pruning and Optimization". In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. Austin, TX, pp. 549–562. DOI: 10.1145/1926385.1926449.

Schrijver, Alexander (1986). *Theory of Linear and Integer Programming*. John Wiley & Sons.

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

Trifunovic, Konrad, Dorit Nuzman, Albert Cohen, Ayal Zaks, and Ira Rosen (2009). "Polyhedral-model guided loop-nest auto-vectorization". In: *Parallel Architectures and Compilation Techniques, 2009. PACT'09. 18th International Conference on*. IEEE, pp. 327–337. DOI: 10.1109/PACT.2009.18.

Vasilache, Nicolas (Sept. 2007). "Scalable Program Optimization Techniques in the Polyhedral Model". PhD thesis. Université Paris Sud XI, Orsay.

Vasilache, Nicolas, Benoît Meister, Muthu Baskaran, and Richard Lethin (Jan. 2012). "Joint Scheduling and Layout Optimization to Enable Multi-Level Vectorization". In: *IMPACT-2: 2nd International Workshop on Polyhedral Compilation Techniques*. Paris, France.

Verdoolaege, Sven (2010). "isl: An Integer Set Library for the Polyhedral Model". In: *Mathematical Software - ICMS 2010*. Ed. by Komei Fukuda, Joris Hoeven, Michael Joswig, and Nobuki Takayama. Vol. 6327. Lecture Notes in Computer Science. Springer, pp. 299–302. DOI: 10.1007/978-3-642-15582-6_49.

Verdoolaege, Sven and Albert Cohen (Jan. 2016). "Live-Range Reordering". In: *Proceedings of the sixth International Workshop on Polyhedral Compilation Techniques*. Prague, Czech Republic. DOI: 10.13140/RG.2.1.3272.9680.

Verdoolaege, Sven and Alexandre Isoard (Nov. 2017). *Consecutivity in the isl Polyhedral Scheduler*. Report CW 709. Leuven, Belgium: Department of Computer Science, KU Leuven. DOI: 10.13140/RG.2.2.15009.10082.

Verdoolaege, Sven and Gerda Janssens (June 2017). *Scheduling for PPCG*. Report CW 706. Leuven, Belgium: Department of Computer Science, KU Leuven. DOI: 10.13140/RG.2.2.28998.68169.

Verdoolaege, Sven, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor (2013). "Polyhedral parallel code generation for CUDA". In: *ACM Trans. Archit. Code Optim.* 9.4, p. 54. DOI: 10.1145/2400682.2400713.

Wolf, Michael E. and Monica S. Lam (June 1991a). "A Data Locality Optimizing Algorithm". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '91. Toronto, Ontario, Canada: ACM, pp. 30–44. DOI: 10.1145/113445.113449.

Wolf, Michael E. and Monica S. Lam (Oct. 1991b). "A Loop Transformation Theory and an Algorithm to Maximize Parallelism". In: *IEEE Transactions on Parallel and Distributed Systems* 2.4, pp. 452–471. DOI: 10.1109/71.97902.

Wolfe, Michael Joseph (1996). *High Performance Compilers for Parallel Computing*. Redwood City, CA: Addison Wesley.

Xilinx (Mar. 2017). Available from https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1207-sdaccel-optimization-guide.pdf. UG1207 (v2016.4).

Zinenko, Oleksandr, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen (2018). "Modeling the Conflicting Demands of Multi-Level Parallelism and Temporal/Spatial Locality in Affine Scheduling". In: *Proceedings of the 27th International Conference on Compiler Construction*. CC 2018. accepted.