Affine Transformations for Communication Minimal Parallelization and Locality Optimization of Arbitrarily Nested Loop Sequences

Uday Bondhugula\textsuperscript{1}, Muthu Baskaran\textsuperscript{1}, Sriram Krishnamoorthy\textsuperscript{1}, J. Ramanujam\textsuperscript{2}, Atanas Rountev\textsuperscript{1}, and P. Sadayappan\textsuperscript{1}

\textsuperscript{1}Dept. of Computer Science and Engg.\textsuperscript{2}Dept. of Electrical & Computer Engg.
The Ohio State UniversityLouisiana State University
2015 Neil Ave. Columbus, OH, USA Baton Rouge, LA
{bondhugu, baskaran, krishnsr, rountev, saday}@cse.ohio-state.edu jxr@ece.lsu.edu

OSU CSE Technical Report
OSU-CISRC-5/07-TR43
Abstract

The polytope model provides powerful abstractions to optimize loop nests with regular accesses for parallel execution. Affine transformations in the polytope model encompass compositions of loop permutation, skewing, reversal, relative shifting, and fusion. Though significant amount of research has dealt with affine scheduling and partitioning, the problem of finding good affine transforms for communication-minimal coarse-grained parallelization as well as locality optimization for the general case of arbitrarily-nested loop sequences has not been addressed. Also, many frameworks do not treat parallelization and locality optimization in an integrated manner, and/or optimize across a sequence of producer/consumer loops.

In this paper, we develop an algorithm for communication minimal and locality optimal tiling of arbitrarily nested loop sequences. The transformed loop nests are a hierarchy of fully permutable loop nest sets such that tiling those leads to minimal communication in the processor space as well as minimal reuse distances for local execution at each node. The approach also finds maximal fusion structures across a sequence of loop nests that have a producer/consumer relationship. Programs with one-dimensional and multi-dimensional schedules are all handled with the same algorithm. Synchronization-free parallelism, permutable loops or pipelined parallelism, and inner parallel loops can be detected. Examples are provided that demonstrate the effectiveness of the approach. The algorithm has been implemented into a tool to generate transformations from C/Fortran code in a fully automatic fashion.

1 Introduction and Motivation

Tiling and loop fusion are two key transformations in optimizing for parallelism and data locality. There has been a considerable amount of research into these two transformations, but very few studies have considered these two transformations in an integrated manner. Tiling has been studied from two perspectives - data locality optimization and parallelization. Tiling for data locality optimization requires grouping points in an iteration space to maximize data reuse. Tiling for parallelism fundamentally involves partitioning the iteration space into tiles that may be concurrently executed on different processors, with a reduced volume and frequency of inter-processor communication.

Loop fusion involves merging a sequence of two or more loops into a fused loop structure with multiple statements in the loop body. Sequences of producer/consumer loops are commonly encountered in applications, where a nested loop statement produces an array that is consumed in a subsequent loop nest. In this context, fusion can greatly reduce the number of cache misses when the arrays are large - instead of first writing all elements of the array in the producer loop (forcing capacity misses in the cache) and then reading them in
the consumer loop (incurring cache misses), fusion allows the production and consumption of elements of the array to be interleaved, thereby reducing the number of cache misses.

The seminal work of Feautrier [13, 12, 14] has led to many research efforts on loop transformation that use an affine scheduling framework [27, 26, 4, 18, 9]. However, we are unaware of any reported affine framework that addresses both of the following questions:

• What is the best set of tiling hyperplanes, to minimize the volume of communication between tiles (in processor space) as well as improve reuse at each processor?

• Given a sequence of nested loops with affine array references and loop bounds, what is (are) the maximal set(s) of fusible loops?

In this paper, we develop an approach to answering the above questions in an affine framework. The main contributions of our approach are two-fold: (1) we find good tiling hyperplanes that minimize communication volume for the general case of multiple iteration spaces with affine dependences, these hyperplanes also improve reuse at a single processor, (2) our algorithm enables detection of maximal fusion structures across a sequence of weakly connected components. We also outline how parallel code should be generated with such a transformation using existing approaches in the literature.

The rest of this paper is organized as follows. Section 2 covers the notation and mathematical background for affine transformations. In Section 3, we motivate and describe our approach in detail. Section 4 shows application of our approach through examples. Section 5 discusses related work and conclusions are presented in Section 7.

2 Background and Notation

This section provides background information on the polytope/polyhedral model, dependence abstraction, and affine machinery.

2.1 The polytope model

Definition 1 (Hyperplane) The set $X$ of all vectors $x \in \mathbb{Z}^n$ such that $\vec{h}.\vec{x} = k$, for $k \in \mathbb{Q}$, forms a hyperplane.

The set of parallel hyperplane instances corresponding to different values of $k$ is characterized by the vector $\vec{h}$ which is normal to the hyperplane. Each instance of a hyperplane is an $n-1$ dimensional subspace of the $n$-dimensional space. Two vectors $x_1$ and $x_2$ lie in the same hyperplane if $h.x_1 = h.x_2$. 

2
Definition 2 (Polyhedron, polytope) The set of all vectors $x \in \mathbb{Q}^n$ such that $Ax + b \geq 0$ defines a (convex) polyhedron. A polytope is a bounded polyhedron.

Each run-time instance of a statement $S$ is defined by its iteration vector $\mathbf{i}$ which contains values for the indices of the loops surrounding $S$, from outermost to innermost. A statement $S$ is associated with a polytope $D_S$ of dimensionality $m_S$. Each point in the polytope is an $m_S$-dimensional iteration vector, and the polytope is characterized by a set of bounding hyperplanes. This is true when the loop bounds are affine combinations of outer loop indices and structure parameters (typically, symbolic constants representing the problem side); our work is focused on programs with this property. Let the hyperplanes bounding the polytope of statement $S_k$ be given by:

$$a_{S,k} \begin{pmatrix} \mathbf{i} \\ \mathbf{n} \end{pmatrix} + b_{S,k} \geq 0, \quad k = 1, m_S$$

where $\mathbf{n}$ is a vector of the structure parameters. A well-known known result useful in the context of the polytope model is the affine form of the farkas lemma.

**Lemma 1 (Affine form of Farkas Lemma)** Let $D$ be a non-empty polyhedron defined by $p$ affine inequalities or faces

$$a_kx + b_k \geq 0, \quad k = 1, p$$

Then, an affine form $\psi$ is non-negative everywhere in $D$ iff it is a positive affine combination of the faces:

$$\psi(x) \equiv \lambda_0 + \sum_k \lambda_k(a_kx + b_k), \lambda \geq 0$$

The non-negative constants $\lambda_k$ are referred to as Farkas multipliers. Proof of the *if* part is obvious. For the *only if* part, see Schrijver [32].

### 2.2 Dependence Abstraction

Our dependence model is the same as the one used by Feautrier [14] and Lim et al. [27, 26]. Dependences are determined precisely through dataflow analysis [13]. The Generalized Dependence Graph (GDG) is a directed multi-graph represented by the tuple $(V, E, D, R)$, where $V$ is the set of vertices with each vertex representing a statement, $E$ is the set of edges where an edge from node $S_i$ to $S_j$ represents a dependence from an instance of $S_i$ to an instance of $S_j$. $D$ is a function from $V$ to the polytopes associated with the statements. $R$ is a function from $E$ to the corresponding dependence relations. There may be multiple edges between two statements, as well as a self-edge for a vertex. The notation we use is similar to that in [14].

For a dependence from $S_i$ to $S_j$ characterized by an edge $e \in E$, let $R_e$ be the corresponding dependence relation. Then, exact dependence analysis makes sure that the dependence relation $R_e$ can be expressed in a
minimal form. A polyhedron $P_e$ and an affine transformation $f_e$ are obtained such that

$$p \in D^{s_i}, \ q \in D^{s_j}, \ \langle p, q \rangle \in R_e \ \equiv \ (p = f_e(q) \land q \in P_e) \quad (3)$$

where $P_e = \{ \vec{q} \mid \vec{q} \in D^{s_j} \land f_e(\vec{q}) \in D^{s_i} \}$ and $f_e$ represents the affine relation between the depending iteration vectors $\vec{p}$ and $\vec{q}$. The right-hand side defines a dependence polyhedron. We have such a polyhedron for every dependence, $e \in E$. Let $P_e$ be defined by:

$$c_{e,k} \begin{pmatrix} i \\ i \\ \vec{i} \end{pmatrix} + d_{e,k} \geq 0, \ k = 1, m_e \quad (4)$$

3 Finding good affine transforms

Let a one-dimensional affine transform for $S_k$ be defined by:

$$\phi_{S_k} = \begin{bmatrix} c_1 & c_2 & \cdots & c_{m_s} \end{bmatrix} \begin{pmatrix} \vec{i} \\ \vec{i} \end{pmatrix} + c_0 \quad (5)$$

$$\phi_{S_k}(\vec{q}) - \phi_{S_i}(\vec{p}) \geq 0, \langle p, q \rangle \in R_e \quad (7)$$

where $h_{S_k} = [c_0, c_1, c_2, \ldots, c_{m_s}]$ with $c_1, c_2, \ldots, c_{m_s} \in \mathbb{Z}$. We do not include $\vec{n}$ in the mapping; implications of this are discussed later.

Let there be a dependence from statement instance $\vec{p}$ of $s_i$ to $\vec{q}$ of $S_j$ corresponding to an edge $e$ of the GDG of a program. After exact dependence analysis, we obtain,

$$\vec{p} = f_e(\vec{q}), \ \vec{q} \in P_e$$

where $P_e$ is as defined in (3), and $f_e$ is an affine function which is also called the $h$-transformation.

**Lemma 2** Let $\phi_{s_i}$ be a one-dimensional affine transform for $S_i$. For $\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_k}$, to be a coordinated valid tiling hyperplane for all statements, the following should hold for each edge $e$ from $S_i$ and $S_j$:

$$\phi_{s_j}(\vec{q}) - \phi_{s_i}(\vec{p}) \geq 0, \ \langle p, q \rangle \in R_e \quad (7)$$

**Proof.** Tiling defined by a set of tiling hyperplanes is said to be legal if each tile can be executed atomically and a valid total ordering of the tiles can be constructed. This implies that there exists no two tiles such that they both influence each other. Since a dependent iteration is mapped to the same hyperplane or a greater
hyperplane than the source, the set of all iterations that are outside of the tile and are influenced by it always lie in the forward direction along one of the independent tiling dimensions. Similarly, all outside iterations influencing a tile are either in that tile or in the backward direction along one or more of the tiling hyperplanes. Hence, an affine transform that satisfies the above constraint can be executed atomically with communication needed only before and after its execution. The above argument holds true for inter-statement dependences too. A dependence from $\vec{p}$ of $S_i$ to $\vec{q}$ of $S_j$ leads to $\vec{q}$ being mapped to a greater or a hyperplane with the same value, leading to an interleaved execution of tiles of iteration spaces of each statement (when code is generated from these mappings). Hence, in the presence of any number of dependences, tiles of $S_1, S_2, \ldots, S_k$ defined by $\phi_{s_1}, \phi_{s_2}, \ldots, \phi_{s_k}$ can be executed in an interleaved fashion atomically.

The above condition was well known for the case of a single-statement perfectly nested loop with constant dependences from the work on Irigoin and Triolet [21] (as $H.D \geq 0$). We have generalized it above for multiple iteration spaces with affine dependences (constant and non-constant) with possibly different dimensionalities. Clearly, this exists a strong relationship between tiling hyperplanes of each statement and they are found in a coordinated fashion. The interleaved execution also leads to the notion of fusion that we discuss later in this section.

In the rest of this paper, we use the term affine transform (with this property) and tiling hyperplane interchangeably, since after applying the transform, the target loop in the transformed iteration space can be blocked rectangularly as all dependences have positive components along that hyperplane.

Tiling hyperplanes are used for parallelization as well as for data locality optimization. Our goal is to find optimal hyperplanes for that purpose for all statements, i.e., the output of our algorithm should be, for each statement, as many hyperplanes as the dimensionality of its iteration space polytope, forming a spanning basis. Since each hyperplane either carries a dependence or puts it in its null space (due to $\geq 0$ in Eqn. 7), and since the set of target hyperplanes span the entire polytope of the statement, every dependence is carried at some level or the other. This approach of finding spanning bases is different from most works in this area that obtain good schedules as outer loops and then inner parallel loops. Parallel code generation in both cases is different; we also address this later.

Consider the perfectly nested version of 1-d Jacobi shown in Fig. 1(a) as an example. This discussion also applies to the imperfectly nested version, but for convenience we first look at the single-statement perfectly nested version. We first describe solutions obtained by existing state of the art approaches - Lim and Lam’s affine partitioning [27, 26] and Griebl’s space and time tiling with FCO placement [18]. Lim and Lam define legal time partitions which have the same property of tiling hyperplanes we described in the previous section. Their algorithm obtains affine partitions that minimize the order of communication while maximizing the degree of parallelism. Using the validity constraint in Eqn 7, we obtain the constraints: $(c_i \geq 0; c_i+c_j \geq 0; c_i-c_j \geq 0)$. 
for $t = 1, T$ do
  for $i = 2, N-1$ do
    $a_{t,i} = 0.33 * (a_{t-1,i} + a_{t-1,i-1} + a_{t-1,i+1})$
  end for
end for
(a) 1-d Jacobi: perfectly nested

for $t = 1$ to $T$ do
  for $i = 2$ to $N-1$ do
    $b_{i} = 0.33 * (a_{i-1} + a_{i} + a_{i+1})$
  end for
end for
S1: $b_{i} = 0.33 * (a_{i-1} + a_{i} + a_{i+1})$

for $i = 2$ to $N-1$ do
  $a_{i} = b_{i}$
end for
S2: $a_{i} = b_{i}$
(b) 1-d Jacobi: imperfectly nested

Figure 1. 1-d Jacobi

Figure 2. Communication volume with different valid hyperplanes for 1-d jacobi

There are infinitely many valid solutions with the same order complexity of synchronization, but with different communication volumes that may impact performance. Although it may seem that the volume may not effect performance considering the fact that communication startup time on modern interconnects dominates, for higher dimensional problems like $n$-d Jacobi, the ratio of communication to computation increases (proportional to tile size raised to $n-1$). Existing works on tiling [31, 30, 35] can find near communication-optimal tiles for perfectly nested loops with constant dependences, but cannot handle arbitrarily nested loops. For 1-d Jacobi, all solutions within the cone formed by the vectors $(1, 1)$ and $(1, -1)$ are valid tiling hyperplanes. For imperfectly nested Jacobi, Lim’s algorithm [27] finds two valid independent solutions without optimizing for any particular criterion. In particular, the solutions found by their algorithm (Algorithm A in [27]) are $(2, -1)$ and $(3, -1)$ which are clearly not the best tiling hyperplanes to minimize communication volume, though they do minimize the order of synchronization which is $O(N)$ (in this case any valid hyperplane has $O(N)$ synchronization). Figure 2 shows that the required communication increases as the hyperplane gets more and more oblique. For a hyperplane with normal $(k, 1)$, one would need $(k + 1)T$ values from the neighboring tile.

Using Griebl’s approach, we first find that only space tiling is enabled with Feautrier’s schedule being $\theta(t, i) = t$. With FCO placement along $(1,1)$, time tiling is enabled that can aggregate iterations into time

\[\text{\footnotesize\textsuperscript{1}}\text{For the imperfectly nested version of 1-d Jacobi, the valid cone is (2, 1) and (2, -1)}\]
tiles thus decreasing the frequency of communication. However, note that communication in the processor space occurs along (1,1), i.e., two lines of the array are required. However, using (1,0) and (1,1) as tiling hyperplanes with (1,0) as space and (1,1) as inner time and a tile space schedule of (2,1) leads to only one line of communication along (1,0). Our algorithm finds such a solution.

We now develop a cost metric for an affine transform that captures reuse distance and communication volume.

3.1 Cost metric in the affine framework

We define an affine form $\delta_e$:

$$
\delta_e(\vec{q}) = \phi_{s_i}(\vec{q}) - \phi_{s_j}(f_e(\vec{q})), \quad \vec{q} \in P_e
$$

The affine form $\delta_e(\vec{q})$ holds much significance. This function is also the number of hyperplanes the dependence $e$ traverses along the hyperplane normal. It gives us a measure of the reuse distance if the hyperplane is used as time, i.e., if the hyperplanes are executed sequentially. Also, this function is a rough measure of communication volume if the hyperplane is used to generate tiles for parallelization and used as a processor space dimension. An upper bound on this function would mean that the number of hyperplanes that would be communicated as a result of the dependence at the tile boundaries would not exceed this bound. We are particularly interested if this function can be reduced to a constant amount or zero by choosing a suitable direction for $\phi$: if this is possible, then that particular dependence leads to a constant or no communication for this hyperplane. Note that each $\delta_e$ is an affine function of the loop indices. The challenge is to use this function to obtain a suitable objective for optimization in the affine framework.

3.2 Challenges

The constraints obtained from Eqn 7 above only represent validity (permutability). We discuss below problems encountered when one tries to apply a performance factor to find a good tile shape out of the several possibilities.

The Farkas Lemma has been used by many approaches in the affine literature [14, 15, 27, 6, 18] to eliminate loop variables from constraints by getting equivalent linear inequalities. The affine form in the loop variables is represented as a positive linear combination of the faces of the dependence polyhedron. When this is done, the coefficients of the loop variables on the left and right hand side are equated to eliminate the constraints of variables. This is done for each of the dependences, and the constraints obtained are aggregated. The resulting constraints are entirely in the coefficients of the tile mappings and Farkas multipliers. All Farkas multipliers can be eliminated, some by Gaussian elimination and the rest by Fourier-Motzkin [32]. However, an attempt to
minimize communication volume ends up in an objective function involving both loop variables and hyperplane coefficients. For example, \( \phi(\vec{q}) - \phi(f_e(\vec{q})) \) could be \( c_1 i + (c_2 - c_3) j \), where \( 1 \leq i \leq N \land 1 \leq j \leq N \land i \leq j \).

One could possibly end up with such a form when one or more of the dependences are not uniform, making it infeasible to construct an objective function involving only the unknown hyperplane coefficients.

A possible approach touched upon by Feautrier is to visit all vertices of the polyhedron in the hyperplane coefficients space characterized by the constraints that express validity. It is likely that vertices will dominate all other points in the solution space. However, this procedure is not scalable beyond the smallest inputs. For example, for a sequence of two nested loops, each with a 3-d iteration space, the number of coefficients is at least 14. \( p \) unknowns could lead to exploration of up to \( 2^p \) vertices (hypercube) in the worst case.

It is also plausible that a positive spanning basis to the set of constraints obtained is better than other solutions. This is due to the fact that any valid tiling hyperplane can be expressed as a positive linear combination of the vectors in the positive spanning basis and that the basis represents the tight extreme vectors for the cone of solutions. This is indeed true for the perfectly nested 1-d Jacobi for which \((1,1)\) and \((1,-1)\) are good hyperplanes. However, we do not know whether this holds in the general case, but clearly they are sub-optimal when compared to \((1,0)\) and \((1,1)\) for perfectly nested 1-d Jacobi.

### 3.3 Approach

We first discuss a result that would take us closer to the solution.

**Lemma 3** If all iteration spaces are bounded, there exists at least one affine form \( v \) in the structure parameters \( \vec{n} \), that bounds \( \delta_e(\vec{q}) \) for every dependence edge \( e \), i.e., there exists

\[
v(\vec{n}) = u.\vec{n} + w
\]

such that

\[
v(\vec{n}) - (\phi_s(\vec{q}) - \phi_s(f_e(\vec{q}))) \geq 0, \; \vec{p} \in P_e, \forall e \in E
\]

\[
v(\vec{n}) - \delta_e(\vec{q}) \geq 0, \; \vec{p} \in P_e, \forall e \in E
\]

The idea behind the above is that even if \( \delta_e \) involves loop variables, one can find large enough constants in \( u \) that would be sufficient to bound \( \delta_e(\vec{p}) \). Note that the loop variables themselves are bounded by affine functions of the parameters, and hence the maximum value taken by \( \delta_e(\vec{p}) \) will be bounded by such an affine form. Also, since \( v(\vec{n}) \geq \delta_e(\vec{p}) \geq 0 \), \( v \) should increase with an increase in the structural parameters, i.e., the coordinates of \( u \) are positive. The reuse distance or communication volume for each dependence is bounded in this fashion by the same affine form.
Now, we apply the Farkas lemma to (10).

\[
v(\vec{n}) - \delta_e(q) \equiv \lambda_{e0} + \sum_{k=1}^{m} \lambda_{ek} \begin{pmatrix} \vec{i} \\ \vec{n} \end{pmatrix} + d_{ek}\tag{11}
\]

The above is an identity and the coefficients of each of the loop indices in \(\vec{i}\) and parameters in \(\vec{n}\) on the left and right hand side can be gathered and equated. We now get linear inequalities entirely in coefficients of the affine mappings for all statements, components of row vector \(\vec{u}\), and \(w\). The above inequalities can be at once be solved by finding a lexicographic minimal solution with \(\vec{u}\) and \(w\) in the leading position, and the other variables following in any order.

\[
\text{minimize}_< \{u_1, u_2, \ldots, u_k, w, \ldots, c'_i, s, \ldots\} \tag{12}
\]

Finding the lexicographic minimal solution is within the reach of the simplex algorithm and can be handled by the PIP software \([12]\). Since the structural parameters are quite large, we first want to minimize their coefficients. We do not lose the optimal solution since an optimal solution would have the smallest possible values for \(u\)’s. Note that the relative ordering of the structural parameters and their values at runtime may effect the solution, but considering this is beyond the scope of this approach.

The solution gives a hyperplane for each statement. Note that the application of the Farkas lemma to (10) is not required in all cases. When a dependence is uniform, the corresponding \(\delta_e\) is independent of any loop variables, and application of the Farkas lemma is not required. In such cases, we just have \(w \geq \delta_e\).

**Finding independent solutions.** Minimizing the objective function with the simplex algorithm gives us a single solution to the coefficients of the best mappings for each statement. We need at least as many independent solutions as the dimensionality of the polytope associated with each statement. Hence, once a solution is found, we run the simplex algorithm again to find the next solution with the constraints being augmented by new ones. The new constraints make sure of independence with solutions found so far. Let the rows of \(H_{s_i}\) represent the solutions found so far for a statement. Then, the sub-space orthogonal to \(H\) is given by:

\[
J = I - H^T (HH^T)^{-1} H \tag{13}
\]

Note that \(JH^T = 0\), i.e., the rows of \(H\) are orthogonal to those of \(J\). Any one of the inequalities given by, \(Jh_S^r \geq \vec{0}\) or \(Jh_S^r \leq \vec{0}\) gives the necessary constraint to be added for statement \(S\). The constraints are added for each statement. The mappings found are independent on a per-statement basis. When there are statements with different dimensionalities, the number of mappings found for each statement is equal to the
dimensionality of the statement with the deepest loop nest. The additional rows of statements with lower
dimensionalities give the necessary embedding into the fully permutable target loop nest. This is possible
when all loop nests can be fused into one fully permutable one – which is equivalent to the existence of a
one-dimensional schedule. In other cases, a hierarchy of fully permutable loop nests are found, and a lower
level in the hierarchy will not be obtained unless constraints corresponding to dependences that have been
carried by outer hyperplane sets have been removed.

3.4 Communication and locality optimization unified

From the algorithm described above, we show that both communication-free and pipelined parallelism
is found by the same algorithm. Note that the best possible solution to the communication minimization
constraint (12) is zero, and this happens when we find a hyperplane that has no dependence components
along its normal, which is a degree of synchronization-free parallelism or an outer parallel loop. Thus, in
each of the steps that we find a new independent hyperplane, we end up first finding all communication-free
hyperplanes; these are followed by a set of fully permutable hyperplanes that are tilable and pipelined parallel
with minimal communication. By bringing in the notion of communication volume and its minimization, we
avoid the need to separately solve the constraint $\phi(q) - \phi(p) = 0$ for space partitioning, and $\phi(q) - \phi(p) \geq 0$ for
time partitioning if the former fails, as was done by Lim and Lam [27, 26]. Just using the latter constraint with
communication volume minimization objective is sufficient to find the target iteration space that minimizes
communication if a loop is used as processor space and minimizes reuse distances if it is used as a time (serial)
loop.

3.5 Space and time in transformed iteration space.

By minimizing $\phi(q) - \phi(p)$ as we find hyperplanes from outermost to innermost, we push dependence
carrying to inner loops and also ensure that no loops have negative dependences components so that all target
loops can be blocked. Once this is done, if the outer loops are used as space (how many ever desired, say $k$),
and the rest are used as time (note that at least one time loop is required unless all loops are synchronization-
free parallel), communication in the processor space is optimized as the outer space loops are the $k$ best ones.
All loops can be tiled resulting in coarse-grained parallelism as well as better reuse within a tile. Hence, the
same set of hyperplanes are used to scan points in a tile, while a transformation is necessary in the outer tile
space loops to get a tile schedule for parallel code generation. This is addressed in Sec. 3.9.
3.6 Fusion in the affine framework

The same affine hyperplane partitioning algorithm described in the previous section can determine valid fusion structures for a sequence of loop nests. We extend affine partitioning to weakly connected components as well, in order to enable fusion across multiple iteration spaces that are weakly connected, as in sequences of producer/consumer loops. For weakly connected components in the GDG, a fictitious outer loop needs to be added that goes across all statements with a condition for each statement to allow the notion of a dependence from two statements.

Consider the sequence of two matrix-vector multiplies in Figure 3(a). Applying affine partitioning on it first gives us only one solution:

\[(c_i, c_j, c_i', c_j') = (1, 0, 0, 1)\]

This implies fusion of the \(i\) loop of \(S_1\) and the \(j\) loop of \(S_2\). Putting the orthogonality constraint now, we do not obtain any more solutions. Hence, now removing the dependence dismissed by it, and running affine partitioning again does not yield any solutions as the loops cannot be fused further. The remaining unfused loops are thus placed lexicographically inside as shown in Figure 3(b).

\[
\begin{align*}
\text{for } i = 1 \text{ to } N \text{ do} \\
\quad \text{for } j = 1 \text{ to } N \text{ do} \\
\quad \quad S1: x(i) = x(i) + a(i, j) \cdot y(j) \\
\quad \text{end for} \\
\text{end for} \\
\text{for } i' = 1 \text{ to } N \text{ do} \\
\quad \text{for } j' = 1 \text{ to } N \text{ do} \\
\quad \quad S2: y(i') = y(i') + a(i', j') \cdot x(j') \\
\quad \text{end for} \\
\text{end for}
\end{align*}
\]

(a) A sequence of matrix vector multiplies

\[
\begin{align*}
\text{for } i = 1 \text{ to } N \text{ do} \\
\quad \text{for } j = 1 \text{ to } N \text{ do} \\
\quad \quad S1: x(i) = x(i) + a(i, j) \cdot y(j) \\
\quad \text{end for} \\
\text{end for} \\
\text{for } i' = 1 \text{ to } N \text{ do} \\
\quad \text{for } j' = 1 \text{ to } N \text{ do} \\
\quad \quad S2: y(i') = y(i') + a(i', j') \cdot x(j') \\
\quad \text{end for} \\
\text{end for}
\end{align*}
\]

(b) Fused matrix vector multiply

Figure 3. Two matrix vector multiplies

Solving for hyperplanes for multiple statements leads to a schedule for each statement such that all statements in question are finely interleaved: this is indeed fusion with either a permutation, a skew, and/or a constant shift. In many cases, we find it to be a permutation (as in Figure 3) or a constant shift (shown for 1-D imperfectly nested Jacobi later). Hence, a common tiling hyperplane also represents a fused loop, and can be employed for components that are weakly connected to reduce reuse distances. Note that it is important to leave the structure parameter \(\tilde{n}\) out of our affine transform definition in 6 for the above to hold true. The set of valid independent hyperplanes that can be found from our algorithm when multiple statements are involved is the maximum number of loops that can be fused for those statements.

Consider the sequence of three matrix vector multiplies shown below.
\[ y = Ax \]
\[ z = By \]
\[ w = Cz \]

Together, these three loop nests do not have a common surrounding loop. However, it is possible to fuse the first two or the last two. When our algorithm is run on all three components, no valid tiling hyperplane is found. Due to space constraints, we do not show the procedure for fusion examples in this section. There exists a valid tiling hyperplane when the algorithm is run for \( M_1 \) and \( M_2 \), or for \( M_2 \) and \( M_3 \), similar to that in Figure 3. With this motivation, we generalize the notion of fusion in the affine framework.

**Enumerating fusion structures.** Given the GDG, a set of nodes on a path can be tested for fusion using affine partitioning. Let there be a path of length \( n \) with a maximum nesting depth of \( k \) for some statement. If there exist \( k \) independent solutions to affine partitioning, the statements are fully fusible, i.e., they can be embedded into a perfectly nested loop. If no valid solution (as in the case of three matrix vector multiplies) exists, there is no common loop for all of the \( n \) statements. Subsequently, fusion for each of the two paths of length \( n - 1 \) are explored. In the general case, we keep proceeding until we find common loops (\( r \) paths each of length \( n - r \) at the \( r \)th step). Once a set of fusible loops are found at any level, the dependences dismissed by them are removed from consideration, and the process is repeated recursively until \( k \) tiling hyperplanes are found. This process allows us to exhaustively enumerate all possible fusion structures. The algorithm described in the previous section can thus also find maximal fusion at each level and enumerate valid fusion structures whenever there is a choice. Our approach currently does not provide a way to choose the best possible structure.

**Multi-dimensional schedules** Code for which other techniques in the literature come up with multidimensional schedules (i.e., code that does not accept one-dimensional schedules), are handled with our algorithm without any special treatment.

\[
\begin{align*}
\text{DO } i &= 1, N \\
&\quad \text{DO } j = 1, N \\
&\quad \quad s = s + a[i,j] \\
&\quad \text{END DO} \\
&\text{END DO}
\end{align*}
\]
Consider the above example from the literature [15]. Using our algorithm, (1,0) is the first and the only hyperplane found that satisfies the permutability constraint for all dependences. Unless dependences carried by it are removed, we cannot find another solution. When this is done, the dependence \((i,j) \rightarrow (i',j')\) where 
\[i' = i + 1 \land j = N \land j' = 1\] is removed; subsequently, \((0,1)\) is found as the next hyperplane.

A loop nest is not amenable to any parallelization if a single solution to the mapping coefficients is obtained repeatedly at every level, and the last loop also carries a dependence. This corresponds to a completely sequential loop nest or multiple loop nests that have only one common surrounding loop (carrying a dependence) at any level.

### 3.7 Summary

We can broadly conclude the following from this section. For affine partitioning on a path in the GDG, the set of independent solutions obtained from our algorithm have the following properties.

If \(k\) independent solutions are found at any level by our algorithm, the following are equivalent:

1. There exist \(k\) fully permutable loops at that level
2. There exist at least \(k - 1\) degrees of parallelism
3. There exist \(k\) tilable loops at that level
4. There exist \(k\) common (fused) surrounding loops at that level

### 3.8 Limitations

**Accuracy of metric and refinement.** The metric we presented here can be refined to any degree as long as the problem is kept within ILP. The motivation behind taking a max is to avoid multiple counting of the same set of points that need to be communicated for different dependences. This happens when all dependences originate from the same data space. Using the sum of max’es on a per-array basis is a more accurate metric. Also, even for a single array, sets of points with very less overlap or no overlap may have to be communicated for different dependences. This is can be inferred from the access functions. Currently, other than the dependences, our approach does not care about the access functions of the dependent iterations and the implications it has. However, for the limited programs tested, taking the max across all dependences works well and is simple. A refined metric can easily be incorporated into our algorithm. Different dependences may be associated with different kinds and degrees of reuse – constant or higher order. Priorities can be given to minimize such dependence components in the transformed space (by making sure they they dominate in \(\delta_e\)). Also, our metric does not consider the amount of computation in the tile per communication. Considering this makes
the problem non-linear and interferes with the stepwise completion procedure for the tiling transformation our
algorithm obtains; also, it yields no better solution in almost all cases.

**Trade-off between fusion and parallelization.** Consider the sequence of matrix vector multiplies shown
in Fig. 3. Fusing it allows better reuse, however it leads to loss of parallelism. Both loop nests can be
parallelized in a synchronization-free fashion when each of them is treated separately, and a synchronization
is needed between them. However, after fusion we only get an inner level of parallelism from the inner loop of
S2. Our approach cannot select the better of these two. It would always fuse if it is legal.

**Concurrent start.** The choice of space loops in the transformed space need not be just made based on
communication volume. When there is a pipelined start-up processor space, the pipelined start-up can be
eliminated in some cases [25]. For 1-D Jacobi, it happens that concurrent start can be enabled by split or
overlapped tiling along (1,1), but not along (1,0); using (1,1) as space with split tiling is shown to provide
better performance [25].

### 3.9 Parallel code generation

Code generation under multiple affine mappings was first addressed by Kelly et al. [24]. Advances were
made by Quilleré et al [29] and Bastoul [7] more recently and have been implemented into a freely available
tool, CLooG [1]. The transformations we produce can be readily given to CLooG as scattering functions.
Unlike most other works, since we find sets of fully permutable loops, there may not be a single loop in the
transformed space that carries all dependences (even if the code admits a one dimensional schedule). The
outer loops we find are space loops and care has to be taken in the parallel code generation when these loops
are pipelined parallel. Our approach to coarse-grained (tiled) parallel code generation is as follows.

1. At any level we have a set of fully permutable loops in the transformed space. Tile all loops in each such
set.

2. Perform the following unimodular transformation on only the outer tile loops that step through the tile
space (for each set):

\[(l_1, l_2, \ldots, l_k) \rightarrow (l_1 + l_2 + \cdots + l_k, l_2, \ldots, l_k)\].

This gives us an outer loop that is a valid tile schedule

3. Place a barrier at the end of the tile schedule loop (inside it) - one barrier for each set of permutable
loops found.
Since each target loop has a non-negative component for every dependence and since all dependence are carried at some level or the other, the sum of all \( \phi \)'s satisfies is a valid tile schedule. Note that communication still happens along boundaries of \( l_1, l_2, \ldots, l_s \), and the same old hyperplanes \( l_1, l_2, \ldots, l_k \) are used to scan a tile. Hence, reuse distances are minimized while scanning a tile and communication in the processor space is minimal. Note that obtaining an affine schedule and then enabling time tiling would still lead to communication along a non-optimal hyperplane. Fig. 4 shows this for a simple example with tiling hyperplanes \((1,0)\) and \((0,1)\).

\[
\text{DO } t=2, 2n, B \\
\text{DOALL } p=\max(1,t-n), \min(t-1,n), B \text{ do} \\
\text{DO } i=1, N \text{ do} \\
\text{DO } j=1, N \text{ do} \\
a[i,j] = a[i-1,j] + a[i,j-1] \\
\text{END DO} \\
\text{END DO} \\
\text{END DOALL} \\
\text{barrier ()} \\
\text{END DO} \\
\]

(b) Coarse-grained parallel barrier

Fig. 4. Shared memory parallel code generation example

4 Examples

In this section, we apply our algorithm on different examples to show the power of the approach.

4.1 Example 1: Non-Constant Dependences

\[
\text{for } i = 1, N \text{ do} \\
\text{for } j = 2, N \text{ do} \\
a[i,j] = a[j,i] + a[i,j-1] \\
\text{end for} \\
\text{end for} \\
\]

(a) Code with non-constant dependences

(b) Pipelined parallelism with \((1,1)\) and \((1,0)\)

Fig. 5. Example 1: Non-constant dependences
Figure 5 shows an example from the literature [10] with affine non-constant dependences. We exclude the constant $c_0$ from the inequalities as we have a single statement. Dependence analysis produces the following dependence polyhedra:

flow: $a(i', j') \rightarrow a(i, j - 1)$ \quad $i' = i; \; j' = j - 1 \land j \geq 2 \land i \geq 1 \land i \leq N \land j \leq N$

flow: $a(i', j') \rightarrow a(j, i)$ \quad $i' = j; \; j' = i$ \quad $j \geq 2 \land i \geq 1 \land i < j \land i \leq N \land j \leq N$

anti: $a(j', i') \rightarrow a(i, j)$ \quad $j' = i; \; i' = j$ \quad $j \geq 2 \land i \geq 1 \land i > j \land i \leq N \land j \leq N$

Permutability constraints lead to

$$c_j \geq 0; \quad c_i - c_j \geq 0$$

The volume bounding constraints for each of the two non-constant dependences are the same (due to the symmetry with respect to $i$ and $j$). On applying Farkas lemma to it:

$$uN + w - (c_i j + c_j i - c_i - c_j j) \equiv \lambda_0 + \lambda_1 (N - i) + \lambda (N - j) + \lambda_0 (j - i - 1) + \lambda_1 (i - 1) + \lambda_2 (i - 1) + \lambda_3 (j - 1)$$

Eliminating as many Farkas multipliers as possible, we obtain:

$$u - \lambda_2 \geq 0$$

$$c_j - c_i - \lambda_3 + \lambda_2 \geq 0$$

$$c_i - c_j + u - \lambda_2 + \lambda_3 \geq 0$$

$$\lambda_2 \geq 0$$

$$\lambda_3 \geq 0$$

$$w \geq 0$$

Minimize $\langle u, w, c_i, c_j, \lambda_1, \ldots \rangle$

$$\text{(15)}$$

The lexicographic minimal solution for the vector $(u, w, c_i, c_j, \lambda_1, \lambda_2, \lambda_3, \lambda_4, \lambda_5) = (0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0)$. Hence, we get $c_i = c_j = 1$. Note that $c_i = 1$ and $c_j = 0$ is not obtained even though it is a valid tiling hyperplane as it involves more communication: it requires $u$ to be positive.

The next solution is forced to have a positive component in the subspace orthogonal to $(1, 1)$ given by (13). This adds the constraint $c_i - c_j > 0$ or $c_j - c_i > 0$ to the existing formulation. We try each to find a solution.
Adding \( c_i - c_j \geq 1 \) to (14), the lexicographic minimal solution is \((1, 0, 1, 0)\), i.e., \( u = 1, w = 0, c_t = 1, c_j = 0 \) \((u = 0 \) is no longer valid). Hence, \((1, 1)\) and \((1, 0)\) are the best tiling hyperplanes. \((1,1)\) is used as space with one line of communication between processors, and the hyperplane \((1,0)\) is used as time in a tile. The outer tile schedule is \((2,1) = (1,0) + (1,1)\).

This is in contrast to other approaches based on schedules which obtain a schedule and then the rest of the transformation matrix. The greedy heuristic gives the schedule \( \theta(i, j) = 2i + j - 3 \) which carries all dependences. However, using this as either space or time does not lead to communication or locality optimization. The \((2,1)\) hyperplane has non-constant communication along it. In fact, the only hyperplane that has constant communication along its normal is the \((1,1)\) hyperplane. This is the best hyperplane to be used as a space loop if the nest is to be parallelized, and is the first solution that our algorithm finds. The \((1,0)\) hyperplane is used as time leading to a solution with one degree of pipelined parallelism with one line per tile of near-neighbor communication (along \((1,1)\)) as shown in Fig. 5(b). Hence, a good schedule that tries to carry all dependences (or as many as possible) is not necessarily a good loop for the transformed iteration space.

4.2 Example 2: Imperfectly Nested 1-d Jacobi

Consider the code in Figure 1(b). The affine dependences and the dependence polyhedra are as follows:

\[
\begin{align*}
(S_1, b[i]) &\rightarrow (S_2, b[j]) & t = t' \land j = i & C_c = c_0 \\
(S_2, b[j]) &\rightarrow (S_1, b[i]) & t = t' + 1 \land j = i & C_c = c_t - c_0 \\
(S_2, a[j]) &\rightarrow (S_1, a[i]) & t = t' + 1 \land j = i & C_c = c_t - c_0 \\
(S_1, a[i]) &\rightarrow (S_2, a[j]) & t = t' \land j = i & C_c = c_0 \\
(S_1, a[i+1]) &\rightarrow (S_2, b[j]) & t = t' \land j = i + 1 & C_c = c_0 + c_j \\
(S_1, a[i-1]) &\rightarrow (S_2, b[j]) & t = t' \land j = i - 1 & C_c = c_0 - c_j \\
(S_2, a[j]) &\rightarrow (S_1, a[i+1]) & t = t' + 1 \land j = i + 1 & C_c = c_t - c_0 + c_j \\
(S_2, a[j]) &\rightarrow (S_1, a[i-1]) & t = t' + 1 \land j = i - 1 & C_c = c_t - c_0 + c_j
\end{align*}
\]

Our algorithm obtains \((c_t, c_i) = (1, 0)\) with \(c_0 = 0\), followed by \((c_t, c_i) = (2,1)\) with \(c_0 = 1\). Note that \(c'_t = c_t\) and \(c'_i = c_i\). The solution is thus given by:

\[
\begin{align*}
\theta_{s_1} &= \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} t \\ i \end{pmatrix} \\
\theta_{s_2} &= \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} t' \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}
\end{align*}
\]
Both iteration spaces have the same hyperplanes, with \((2,1)\) hyperplane of \(S_2\) having a constant shift; the resulting transformation is equivalent to a constant shift of \(S_2\) relative to \(S_1\), fusion and skewing the \(i\) loop with respect to the \(t\) loop by a factor of 2. The \((1,0)\) hyperplane has the least communication: no dependence crosses more than one hyperplane instance along it.

5 Related work

Iteration space tiling [21, 34, 33, 30] is a standard approach for aggregating a set of loop iterations into tiles, with each tile being executed atomically. In addition, researchers have considered the problem of selecting tile shape to minimize communication or improve locality [31, 30, 8, 35, 19]. These works on tile shape optimization are restricted to a single perfectly nested loop with uniform dependences. Tile size directly impacts the trade-off between parallelism and communication, and has been investigated [11, 20, 19].

Kelly and Pugh’s algorithm finds one dimension of parallelism for programs with arbitrary nesting and sequences of loops [23, 22]. Their program transforms include loop permutations and reversals, but not loop skewing. The exclusion of loop skewing enables them to enumerate all the possible transformation choices and select the best one based on communication cost.

Scheduling with affine functions using faces of the polytope by application of the Farkas algorithm was first proposed by Feautrier [14]. Feautrier explored various possible approaches to obtain good affine schedules that minimize latency. The one-dimensional schedules (wherever they can be found) carry all dependences and so all the inner loops are parallel. Using reasonable heuristics usually yields good solutions. However, transforming to permutable loops that are amenable to tiling or detecting outer parallel loops is not addressed. As discussed and shown in Sec. 3, using this schedule as one of the loops for parallel code generation does not necessarily optimize communication or locality. Hence, schedules need not be good hyperplanes for tiling.

Several works [17, 18, 9, 28] make use of such schedules. Though this yields inner parallel loops, they cannot tile the time loop unless communication in the space loops is in the forward direction [18] (dependences have positive components along in all dimensions). We force a \(\geq 0\) constraint for each dependence while finding each hyperplane. This allows the target loop nest to be fully permutable as well as allows us to put the minimization objective for synchronization and reuse distances. Our algorithm ends up finding inner parallel loops only if permutable loops are not available.

Lim and Lam [27, 26] use the same dependence model as us and propose an affine framework that identifies outer parallel loops (communication-free space partitions) and permutable loops (pipelined parallel or tilable loops) with the goal of minimizing the order of synchronization. Their approach subsumed previous work done on unimodular and affine transformations. However, several (infinitely many) solutions equivalent in terms of the criterion they optimize for result from their algorithm, and these significantly differ in communication
cost; no metric is provided to differentiate between these solutions. Also, tiling for locality is not handled in an integrated way with parallelization, i.e., parallel SPMD code generated from their affine transforms does not guarantee that the code on each node is optimized for locality. Fusion across a sequence of weakly connected components is not handled. This is important to optimize a sequence of producer/consumer loops. Our solution addresses all of these aspects.

Ahmed et al. [5, 6] propose a framework for data locality optimization of imperfectly nested loops for sequential execution on a uniprocessor. The approach determines the embedding for each statement into a product space, which is then optimized for locality. It maximizes the number of loops that can be tiled, and improves locality by minimizing a metric based on the reuse distance. The technique used for minimizing reuse distances again has the same shortcoming as Lim’s solution: no approach is provided to find the best affine transform to convert the product space to a fully permutable nest with minimal reuse distances (when non-uniform dependences exist).

Griebl [17, 18] presents an integrated framework for optimizing data locality and parallelism with tiling. Though Griebl’s approach enables time tiling by using a forward communication only placement with an existing schedule, it does not necessarily minimize communication. As mentioned earlier (Sec. 3), using schedules as time loops may not lead to communication-optimized solutions. Also, fusion of a sequence of producing-consuming nested loops is not addressed.

Recent work by Cohen et al., Girbal et al., Pouchet et al. [9, 16, 28] has focused on searching the space of affine transformations to find good ones through iterative optimization by employing performance counters. This is because the space of valid affine transforms for loop nests in an entire program is very large. Our approach directly obtains good solutions without search. However, in some cases empirical and iterative optimization is required to choose transforms that work best in practice. This is true, for example, when we need to choose among several different fusion structures and our algorithm cannot differentiate between them, or when there is a tradeoff between fusion and parallelization (Sec. 3.6). Also, effective determination of tile sizes and unroll factors for transformed whole-programs is only possible through empirical search through performance measurement. A combination of our affine transformation algorithm and empirical search in a smaller space is an interesting approach to pursue. Alternatively, more powerful cost models can be employed once solutions in a smaller space can be enumerated.

6 Implementation

We have implemented our transformation framework using PipLib 1.3.3 [12] and Polylib 5.22.3 [3]. Our tool takes as input dependence information (dependence polyhedra and h-transformations) from LooPo’s [2] dependence tester and generates statement-wise affine transformations. Flow, anti and output dependences
are considered for legality as well as the minimization objective. The transforms generated by our tool are
provided to CLooG [7] as scattering functions. Fig. 6 shows the flow. The goal is to get tiled shared memory parallel code.

Figure 6. Implementation status

Following are transformations obtained by our tool automatically from C/Fortran source code. The transformed code was generated using CLooG 0.14.0 [1]. The original code is shown on the left while the transformed code with the transformation matrices is on the right. Note that the space and time dimensions are not explicitly marked (but have been inferred), and the tiled code is not shown (but all loop nests are transformed to a tree of fully permutable nests).

6.1 2-d imperfectly nested Jacobi

```c
CONSTANT n;
DO t=1, n
  DO i=2,n-1
    DO j=2,n-1
      b[i,j] = a[i-1,j]+a[i,j]+a[i+1,j] + a[i,j-1]+a[i,j+1];
    END DO
  END DO
  END DO
  DO k=2,n-1
    DO l=2,n-1
      a[k,l] = b[k,l];
    END DO
  END DO
END DO
```

Figure 7. Imperfectly nested 2-d Jacobi

Fig. 7 shows the code and the transformation. The transformation implies shifting the i and j loop of statement S2 by one iteration each, fusion with S1, skewing of the fused i and j loops with respect to the time
loop by two.

\[
\text{CONSTANT } n; \\
\text{DO } k = 1, n \\
\quad \text{DO } j = k + 1, n \\
\quad \quad S1: a[k,j] = a[k,j] / a[k,k]; \\
\quad \text{END DO} \\
\text{DO } i = k + 1, n \\
\quad \text{DO } j = k + 1, n \\
\quad \quad S2: a[i,j] = a[i,j] - a[i,k] * a[k,j]; \\
\quad \text{END DO} \\
\text{END DO} \\
\text{END DO}
\]

\[
\text{DO } c1 = 1, M - 1 \\
\text{DO } c2 = c1 + 1, M \\
\quad S1(k = c1, j = c2) \\
\quad \text{DO } c3 = c1 + 1, M \\
\quad \quad S2(k = c1, i = c3, j = c2) \\
\quad \text{END DO} \\
\text{END DO} \\
\text{END DO}
\]

\[
\begin{array}{c|cccc}
 k & j & \text{const} & k & i & j & \text{const} \\
\hline
 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

**Figure 8. LU decomposition**

### 6.2 LU decomposition

Fig. 8 shows the code and transformation. The last row of S1 gives the necessary condition \((k = i)\) to embed S1 in the innermost loop of S2 in the target loops.

### 6.3 Sequence of Matrix-Matrix multiplies

\[
\text{CONSTANT } n; \\
\text{DO } i = 1, n \\
\quad \text{DO } j = 1, n \\
\quad \text{DO } k = 1, n \\
\quad \quad S1: C[i,j] = C[i,j] + A[i,k] * B[k,j] \\
\quad \text{END DO} \\
\quad \text{END DO} \\
\text{END DO} \\
\text{DO } i = 1, n \\
\quad \text{DO } j = 1, n \\
\quad \text{DO } k = 1, n \\
\quad \quad S2: D[i,j] = D[i,j] + E[i,k] * C[k,j] \\
\quad \text{END DO} \\
\quad \text{END DO} \\
\text{END DO}
\]

\[
\text{DO } c1 = 1, n \\
\text{DO } c2 = 1, n \\
\quad \text{DO } c4 = 1, n \\
\quad \quad S1(i = c1, j = c2, k = c4) \\
\quad \text{END DO} \\
\quad \text{DO } c4 = 1, n \\
\quad \quad S2(i = c4, j = c2, k = c1) \\
\quad \text{END DO} \\
\text{END DO}
\]

\[
\begin{array}{c|cccc}
 i & j & k & \text{const} & i & j & k & \text{const} \\
\hline
 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\]

**Figure 9. Sequence of MMs**

A horizontal line in the solution matrix separates sets of permutable loops. Note that each of the original loop nests had two degrees of synchronization-free parallelism, but a synchronization is needed after the first
loop nest is executed. The transformed loop nest has one outer parallel loop \((c1)\), but reuse is improved as each element of matrix \(C\) is consumed immediately after it is produced, and \(C\) can be contracted to a single scalar.

\[
\begin{align*}
&\text{DO } i = 2, n-1 \\
&\quad a1[i] = a0[i-1] + a0[i] + a0[i+1];
\end{align*}
\]

\[
\begin{align*}
&\text{END DO} \\
&\text{DO } i = 2, n-1 \\
&\quad a2[i] = a1[i-1] + a1[i] + a1[i+1];
\end{align*}
\]

\[
\begin{align*}
&\text{END DO} \\
&\text{DO } i = 2, n-1 \\
&\quad a3[i] = a2[i-1] + a2[i] + a2[i+1];
\end{align*}
\]

\[
\begin{align*}
&\text{END DO} \\
&\text{DO } i = 2, n-1 \\
&\quad a4[i] = a3[i-1] + a3[i] + a3[i+1];
\end{align*}
\]

\[
\begin{align*}
&\text{END DO} \\
&\text{DO } i = 2, n-1 \\
&\quad a5[i] = a4[i-1] + a4[i] + a4[i+1];
\end{align*}
\]

\[
\begin{align*}
&\text{END DO}
\end{align*}
\]

\[
\begin{align*}
&\text{DO } c1=2, M+3 \\
&\quad \text{IF } (c1 <= M-1) \text{ THEN} \\
&\quad \quad S1(i = c1) \\
&\quad \text{END IF} \\
&\quad \text{IF } ((c1 >= 3) \text{ .AND. } (c1 <= M)) \text{ THEN} \\
&\quad \quad S2(i = c1-1) \\
&\quad \text{END IF} \\
&\quad \text{IF } ((c1 >= 4) \text{ .AND. } (c1 <= M+1)) \text{ THEN} \\
&\quad \quad S3(i = c1-2) \\
&\quad \text{END IF} \\
&\quad \text{IF } ((c1 >= 5) \text{ .AND. } (c1 <= M+2)) \text{ THEN} \\
&\quad \quad S4(i = c1-3) \\
&\quad \text{END IF} \\
&\quad \text{IF } (c1 >= 6) \text{ THEN} \\
&\quad \quad S5(i = c1-4) \\
&\quad \text{END IF}
\end{align*}
\]

\[
\begin{array}{cccc|cccc}
S1 & S2 & S3 & S4 & S5 \\
\hline
i & const & i & const & i & const & i & const \\
1 & 0 & 1 & 1 & 1 & 2 & 1 & 3 \\
\end{array}
\]

\textbf{Figure 10. Multi statement stencil}

6.4 Multiple statement stencils

This code (Fig. 10) is representative of multimedia applications. The transformed code enables immediate reuse of data produced by each statement at the next statement.

6.5 TCE four-index transform

This is a sequence of four loops, each of depth five occurring in Tensor Contraction Expressions. Our tool transforms the code to that shown in Fig. 11.

7 Conclusions

We have presented a single framework that addresses key aspects of automatic parallelization and data locality optimization using affine transforms. We proposed an algorithm to find near communication-optimal tiling hyperplanes for parallelization of a sequence of arbitrarily nested loops. The same hyperplanes also minimize reuse distances and improve data locality. The approach also enables fusion across a sequence of
producer-consumer loops. We are integrating our framework into LooPo – a polyhedral loop parallelizer and compiler infrastructure.

References

   http://icps.u-strasbg.fr/polylib/.


DO a = 1, N
  DO q = 1, N
    DO r = 1, N
      DO s = 1, N
        T1[a,q,r,s] = T1[a,q,r,s] + A[p,q,r,s] \times C4[p,a]
      END DO
    END DO
  END DO
END DO

DO a = 1, N
  DO q = 1, N
    DO r = 1, N
      DO s = 1, N
        T2[a,b,r,s] = T2[a,b,r,s] + T1[a,q,r,s] \times C3[q,b]
      END DO
    END DO
  END DO
END DO

DO a = 1, N
  DO b = 1, N
    DO c = 1, N
      DO s = 1, N
        T3[a,b,c,s] = T3[a,b,c,s] + T2[a,b,r,s] \times C2[r,c]
      END DO
    END DO
  END DO
END DO

DO a = 1, N
  DO b = 1, N
    DO c = 1, N
      DO d = 1, N
        B[a,b,c,d] = B[a,b,c,d] + T3[a,b,c,s] \times C1[s,d]
      END DO
    END DO
  END DO
END DO

(a) Original code

(b) Transformed code

Figure 11. TCE 4-index transform