A Singular Loop Transformation Framework Based on Non-singular Matrices

Wei Li

Keshav Pingali

Department of Computer Science

Cornell University

Ithaca, New York 14853

Abstract

In this paper, we discuss a loop transformation framework that is based on integer non-singular matrices. The transformations included in this framework are called Λ-transformations and include permutation, skewing and reversal, as well as a transformation called *loop scaling*. This framework is more general than existing ones; however, it is also more difficult to generate code in our framework. This paper shows how integer lattice theory can be used to generate efficient code. An added advantage of our framework over existing ones is that there is a simple completion algorithm which, given a partial transformation matrix, produces a full transformation matrix that satisfies all dependences. This completion procedure has applications in parallelization and in the generation of code for NUMA machines.

[1]This work was supported by the Cornell Theory Center, NSF Presidential Young Investigator award #CCR-8958543, by NSF grant #CCR-9008526, and by a grant from the Hewlett-Packard Corporation.

Keywords: Loop transformations, compound transformations, non-singular matrices, Hermite normal form, code generation.

1 Introduction

The importance of loop transformations in generating good code for vector and parallel machines is widely recognized [13, 1, 19]. A recent advance in this area is the use of *unimodular* matrices to model three important loop transformations — *permutation*, *skewing* and *reversal*. Unimodular matrices have integer entries and a determinant that is 1 or -1; therefore, they are closed under matrix product. It follows that any sequence of these loop transformations can also be represented as a unimodular matrix; conversely, any unimodular matrix can be interpreted as representing a sequence of permutation, skewing and reversal transformations. The main benefit of the unimodular abstraction is that it provides an approach to tackling the so-called 'phase-ordering problem' — for many problems where there is no obvious order in which the transformations should be performed, it is often possible to generate a unimodular matrix from which the desired order of loop transformations can be determined easily. Banerjee has used this framework to address the problem of generating parallel loops [3]; Wolf and Lam have used this framework extensively to address both this problem and that of promoting data reuse for improving cache performance [17, 18].

In this paper, we propose to use *non-singular* matrices, rather than unimodular matrices, as a foundation for modeling loop transformations. Non-singular matrices include unimodular matrices as a special case, and permit us to include a new transformation called *loop scaling* in this framework. Surprisingly, code generation is somewhat more intricate for non-singular matrices than for unimodular matrices, and it is the main concern of this paper.

Another advantage of our approach is that it is easier to generate non-singular matrices than it is to generate unimodular matrices. A typical algorithm that uses the matrix framework, such as the generation of parallel outermost loops [3] or the exploitation of locality in NUMA architectures [11], determines the first few rows of the matrix, and then 'pads out' the remaining rows to generate a matrix that represents a legal transformation. It is easier to generate a non-singular matrix than a unimodular matrix since there are fewer constraints to be satisfied, and in this paper, we give a completion procedure that produces a non-singular matrix, given the first few rows. This completion procedure is non-trivial since we must ensure that the result matrix respects the dependencies of the loop nest.

The rest of the paper is organized as follows. In Section 2, we define the problem formally, outline our loop restructuring framework and discuss the difficulties in generating the transformed loop nest. In Section 3, we sketch the code generation technique for the case of unimodular matrices, and discuss why it cannot be used directly for non-singular matrices. In Section 4, we solve the code generation problem for non-singular matrices. The key technical result is that a non-singular matrix can be decomposed into the product of a lower triangular matrix with positive diagonal elements and a unimodular matrix. Using these two matrices, we generate the transformed loop nest. In Section 5, we give a completion procedure for non-singular matrices. The last section discusses related work.

2 Linear Loop Transformations

In this section, we introduce integer lattices as a model of the iteration space of loops, and non-singular matrices as a model of loop transformations.

2.1 Iteration Spaces and Integer Lattices

Consider the loop nest in Figure 1(a) whose iteration space is shown in Figure 1(c). The points in the iteration space of this loop can be modeled as integer vectors in the two dimensional space $\mathbb{Z}^2$, where $\mathbb{Z}$ is the set of integers. For example, the iteration ($i = 2$, $j = 3$) can be represented by the vector (2, 3). In general, points in the iteration space of a loop nest of depth n can be represented by integer vectors from the space $\mathbb{Z}^n$. It is convenient to use the theory of *integer lattices* [5] and view the points in the iteration space as being generated by integral linear combinations of a set of basis vectors. For example, it is easy to see that the points in the iteration space shown in Figure 1(c) can be generated by integral linear combinations of two integer vectors $\begin{pmatrix} 0 \\ 1 \end{pmatrix}$ and $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Similarly, the iteration space in Figure 1(d) is generated by linear combinations of the integer vectors $\begin{pmatrix} -2 \\ 1 \end{pmatrix}$ and

$\begin{pmatrix} 4 \\ 1 \end{pmatrix}$.

For future reference, we define these concepts more precisely.

Definition 2.1 Let $a_1, a_2, ..., a_m$ be a set of linearly independent integer vectors. The set $\Lambda = \{\lambda_1 a_1 + \lambda_2 a_2 + ... + \lambda_m a_m \mid \lambda_1, ..., \lambda_m \in \mathbb{Z}\}$ is called an *integer lattice* generated by the *basis* $a_1, a_2, ..., a_m$.

We will call an integer matrix a *basis matrix*, if its columns are a basis.

The loop nest in Figure 1(a) has the property that every integer point within the loop bounds is a point in the iteration space of the loop nest. We will call this a *dense* iteration space. By contrast, Figure 1(d) shows a *sparse* iteration space because the integer point (2, 3), for example, is within the loop bounds but does not represent a point in the iteration space of the loop. The notion of *dense* and *sparse* can be formally defined as follows.

Definition 2.2 An iteration space is *dense*, if for any two integer vectors v_1 and v_2 representing loop iterations, any integer vector $v_3 = \lambda v_1 + (1 - \lambda) v_2$ for some $0 \leq \lambda \leq 1$ also represents a loop iteration. An iteration space is *sparse* if it is not dense.

The significance of this classification of iterations spaces is that it is considerably more difficult to generate code for a loop nest if the iteration space is sparse, than if it is dense, as we will show in Section 3.

Let e_i be an n-dimensional vector with 1 in the ith entry and 0 elsewhere.

Theorem 2.1 *The integer vectors from a n-dimensional dense iteration space form an integer lattice with the basis e_1, e_2,... e_n.*

Proof: Obvious. □

2.2 Loop Transformations

In this paper, we will focus on transformations that can be represented by linear, one-to-one mappings from the iteration space of the source program to the iteration space of the target program. This class of transformations includes permutation, skewing and reversal, as well as a new transformation called *scaling*. Examples of these transformations are shown in Figure 2. These transformations are standard except for scaling which corresponds to replacing a loop iteration variable by an integer multiple of it. Loop scaling gives the ability to transform sparse iteration spaces, which is important in iteration space tiling. For example, if a two dimensional iteration space is partitioned into 2×2 tiles, each tile can be represented by its bottom-left corner. The space of these representatives is a sparse space, and it can be viewed as the result of loop scaling by a factor of 2. Loop scaling is also useful in *access normalization* [11] which is a loop transformation for improving data locality.

Linear, one-to-one mappings between iteration spaces can be modeled using *integer, non-singular matrices*. The reader can verify that the matrices shown in Figure 2 perform the desired mappings from the source iteration space to the target iteration space, and are integer and non-singular. Similarly, in Figure 1(d), the points in the target iteration space are the image of the source iteration space points under the integer, non-singular matrix $T = \begin{pmatrix} -2 & 4 \\ 1 & 1 \end{pmatrix}$.

Definition 2.3 A loop transformation is called a Λ-*transformation* if it can be modeled by an integer non-singular matrix.

Performing a sequence of transformations corresponds to composing the mappings between iteration spaces, which, in turn, can be modeled as the product of the matrices representing the individual transformations. Since the product of any number of integer, non-singular matrices is integer and non-singular, it follows that the set of Λ-*transformations* is closed under composition.

Conversely, we can show that the transformation represented by any integer non-singular matrix can be viewed as a composition of the four basic transformations. More precisely, we have the following result.

Theorem 2.2 *The transformation represented by any integer non-singular matrix can be viewed as a composition of permutation, skewing, reversal and scaling.*

Proof: By applying the appropriate elementary row and column operations, an integer non-singular can be reduced to a diagonal matrix [16]. The elementary operations can be represented by multiplying interchange, reversal and skewing matrices on the left hand side and on the right hand side of the non-singular matrix. The diagonal matrix can be further reduced to a product of scaling matrices. □

As mentioned earlier, unimodular transformations are the subset of Λ-transformations without loop scaling. An important property of unimodular transformations is the following.

Theorem 2.3 *Unimodular transformations map a dense (sparse) iteration space to another dense (sparse) iteration space.*

Proof: The lattice remains the same, since only the basis is changed [16]. □

2.3 Legality of Λ-transformations

Not every Λ-transformation is valid with respect to the data dependencies in the original loop nest. Data dependencies can be represented by *distance* or *direction* vectors that are lexicographically positive. For example, a distance vector $d = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ means that the iteration (i, j) depends on the iteration $(i-3, j-2)$. Then Td is the dependence vector in the new iteration space, since Λ-transformations are linear. A Λ-transformation T is legal if and only if Td is lexicographically positive.

2.4 Generating Code

To generate code for the target loop nest, we must generate DO-loops that scan the points of the target iteration space in lexicographic order, and replace occurences in the loop body of the old loop indices with the new loop indices. The first problem is non-trivial and is discussed in Sections 3 and 4. On the other hand, the problem of transforming the loop body is relatively straight-forward and we sketch a solution here for completeness. If vectors S_i and S_j represent the source and target iteration variables, notice that $S_i = T^{-1}S_j$. This is just a set of equations expressing the old subscripts in terms of the new ones, and it can be used to eliminate occurrences of the source iteration variables in the body of the loop in favor of the new ones. For our running example, this set of equations is the following:

$$\begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} -1/6 & 4/6 \\ 1/6 & 2/6 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}$$

The transformed loop body is shown in Figure 1(b).

Note that $T^{-1}S_j$ will always be an integer point even though T^{-1} may be a rational matrix; therefore, expressions like $u/2$ and $(u+2v)/6$ in Figure 1(b) should be strength reduced.

3 Difficulties in Generating Code

The difficulty in generating DO-loops to scan the target iteration space is that a Λ-transformation, in general, does not preserve lexicographic order (two iterations may be performed in one order in the source loop nest but in a different order in the target loop nest), so there is no obvious way to use the source loop nest to generate code. As a first attempt, we can find the image of the original bounds (the four inclined lines in Figure 1(d) for the running example), and then generate a loop nest that visits in lexicographical order all the integer points in the area bounded by the image. In this section, we show that this approach works well when the target iteration space is dense; sparse iteration spaces will require additional machinery.

3.1 Computing Image of Bounds

There are many ways to compute the image of the original bounds; here, we describe a simple method that uses Fourier-Motzkin elimination.

Given a non-singular matrix representing the transformation, the image bounds for the target loop can be computed using the inverse of the transformation. Let $S_i = (i_1, .., i_n)^T$ and $S_j = (j_1, .., j_n)^T$ be source and target loop indices respectively under the non-singular transformation T. Let the loop bounds for loop i_k be an affine function of loop indices

$i_1, ..i_{(k-1)}$. Each lower bound is in the form of $a_{j1}i_1 + .. + a_{j(k-1)}i_{(k-1)} + b_j \leq i_k$. There may be many such lower bounds whose maximum is the lower bound for i_k. Similarly for upper bounds, there may be many affine bounds whose minimum is the upper bound for i_k. The bounds in the loop nest can be written in the following matrix form:

$$L_b S_i + b_l \leq I_l S_i \quad \text{and} \quad I_u S_i \leq U_b S_i + b_u$$

where $L_b(U_b)$ is an $m_l \times n$ ($m_u \times n$) matrix, $b_l(b_u)$ is a vector of length $m_l(m_u)$. $I_l(I_u)$ is an identity matrix with some of its rows replicated to an $m_l \times n$ ($m_u \times n$) matrix. Each row of L_b (U_b) plus the corresponding row from $b_l(b_u)$ form one lower(upper) bound. Note that m_l and m_u are different because the number of expressions in the piece-wise linear lower and upper bounds are different in general.

Then the source iteration space is bounded by the following inequalities:

$$AS_i \leq b, \quad \text{where} \quad A = \begin{pmatrix} L_b - I_l \\ I_u - U_b \end{pmatrix}, b = \begin{pmatrix} -b_l \\ b_u \end{pmatrix}$$

The bounds for S_j are found by replacing S_i by $T^{-1}S_j$.

$$AT^{-1}S_j \leq b \tag{1}$$

These inequalities cannot be used directly as loop bounds, since the bounds for a loop can only be a function of outer loop indices. We use the Fourier-Motzkin elimination algorithm [6] suggested in [2] to compute the suitable bounds. The Fourier-Motzkin algorithm may introduce redundant constraints, but these may be eliminated [2].

The Fourier-Motzkin algorithm is quite simple. Consider a system of linear inequalities

$$\textstyle\sum_{j=1}^{n} a_{ij}x_j \leq b_j, \qquad i = (1, .., m).$$

This system can be partitioned into three sets of inequalities according to the sign of the coefficient of x_n.

$$\begin{aligned} x_n &\leq D_i(\overline{x}), \quad i = (1, .., p) \\ x_n &\geq E_j(\overline{x}), \quad j = (1, .., q) \\ 0 &\leq F_k(\overline{x}), \quad k = (1, .., r) \end{aligned}$$

where D_i, E_j and F_k are linear functions of $\overline{x} = (x_1, ..x_{(n-1)})$.

Now, we can eliminate x_n from the system to get the following *reduced system.*

$$\begin{aligned} E_j(\overline{x}) &\leq D_i(\overline{x}), \quad i = (1, .., p), \quad j = (1, .., q) \\ 0 &\leq F_k(\overline{x}), \quad k = (1, .., r) \end{aligned}$$

This process can be repeated until there is exactly one variable left. The bounds for this variable can be determined from inspection of the reduced system of equations.

We remark in passing that our use of the Fourier-Motzkin algorithm is justified because the source iteration space is convex and the transformation is a linear, one-to-one mapping to the target iteration space. Therefore, the target iteration space is convex and has the same number of dimensions as the source iteration space.

Going back to our problem, consider the system of inequalities for S_j. The loop bounds for j_n can be computed by solving the inequalities for j_n. The bounds for j_k can be computed by first eliminating $j_{(k+1)}, ..j_n$ from the system using Fourier-Motzkin elimination, then solving for j_k etc.

Consider the working example. The iteration space (Figure 1(c)) is represented by the integer vectors bounded by the system of linear inequalities in Figure 1(e). By computing

i, j in terms of u, v, replacing i, j by u, v in the inequalities and using the Fourier-Motzkin elimination, we have the image of the source bounds (Figure 1(f)). Unfortunately, we cannot use these inequalities directly to generate code for the target loop nest. There are two problems. First, the lower and upper bounds may not even be integers — for example, when $u = 4$, the lower bound for v is $\frac{5}{2}$. Furthermore, even though the source iteration space is dense, the target iteration space is sparse. This means that we must find some way to skip over points (like (2,3) in our example) that are not in the iteration space of the target loop nest.

3.2 Dense Spaces

For the special case when the target iteration space is dense (such as when a unimodular matrix is used to transform a loop nest with a dense iteration space (Theorem 2.3)), both these problems can be solved easily. If the target iteration space is dense, there is no need to skip over points that are not in the iteration space of the target loop nest. Furthermore, we can use floor and ceiling operations to get the nearest integers within the image bounds.

For example, consider the unimodular transformation $U = \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}$ on the working example.

$$\begin{pmatrix} p \\ q \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix}$$

For the source bounds in Figure 1(e), we can compute the image bounds shown in Figure 3(a). Since the target space is dense, we can use the ceiling and floor operations to compute the exact bounds shown in Figure 3(b).

3.3 Discussion

For sparse iterations spaces, the ceiling and floor operations cannot solve the problem. For the example in Figure 1(d), (4, 3) is the closest integer point to the boundary of v when

$u = 4$, but the starting point of the target loop nest is (4, 4). One possibility is to use conditional tests in the loop body to avoid executing the loop body at points that do not correspond to points in the target iteration space. This approach has been used by other researchers [12], but it involves visiting integer points that are not necessary; moreover, the conditional tests are expensive.

4 Algorithm for Code Generation

The key insight to solving the general problem is that an integer non-singular matrix T can be decomposed into the product of a lower triangular matrix H with positive diagonal elements and a unimodular matrix U. This decomposition is related to the *Hermite normal form* of the transformation matrix, which has been used in the context of integer programming to solve systems of integer equations [16], and in the context of optimizing data transfers in memory hierarchies to determine data transfer volumes [8]. We show that if U is used to transform the program, the resulting program executes iterations in the same lexicographic order as the program obtained by using T as the transformation matrix. We also show that the diagonal elements of H correspond to loop step sizes. Putting these observations together gives an algorithm that generates efficient code for the general case of non-singular matrices.

4.1 Auxiliary Iteration Space

By applying column operations to an integer non-singular matrix T, we can reduce it to an integer lower triangular matrix with positive diagonal elements. This lower triangular matrix is related to the *Hermite normal form* [16] of the matrix T. It follows that T can be written as the product of a lower triangular matrix H with positive diagonal elements and a unimodular matrix U that represents the composition of the column operations. This decomposition is not unique, but for our purpose, any such decomposition is adequate; to avoid being pedantic, we will abuse terminology and refer to any such H as the Hermite form of the transformation matrix T. Figure 4 shows how to compute H and U.

Let $T = HU$, and let the source space be S_i, and the target space be S_j. Define $S_k = US_i$. Then,

$$S_j = TS_i = HUS_i = HS_k$$

Definition 4.1 The iteration space S_k is called the *auxiliary iteration space* of S_i with respect to the decomposition HU.

Theorem 4.1 *The auxiliary iteration space is a dense space if the source space is dense.*

Proof: Follows from Theorem 2.3 since U is unimodular. $\square$

Therefore the exact loop bounds of the auxiliary space can be computed using the algorithm in Section 3.

An important property of the auxiliary space is that it executes iterations in the same lexicographic order as the target iteration space. To see this, consider our running example.

$$T = \begin{pmatrix} -2 & 4 \\ 1 & 1 \end{pmatrix} \quad H = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \quad U = \begin{pmatrix} -1 & 2 \\ 0 & 1 \end{pmatrix}$$

H is lower triangular with positive diagonal elements, and U is unimodular. Consider using U to transform the source program.

$$\begin{pmatrix} p \\ q \end{pmatrix} = U \begin{pmatrix} i \\ j \end{pmatrix}$$

This is the unimodular transformation considered in Figure 3 in Section 3, and the bounds of the auxiliary iteration space are shown in Figure 3(b). To develop the readers insight into the relationship between the source, auxiliary and target iteration spaces, the mappings between these spaces are shown below — notice that both (p, q) and (u, v) are traversed in the same lexicographical order, but that this order is different from that of the source. This can also be seen by comparing the iteration space diagrams in Figures 1(d) and 3.

$$
\begin{array}{ccccc}
(i, j) & \rightarrow & (p, q) & \rightarrow & (u, v) \\
(3, 1) & \rightarrow & (-1, 1) & \rightarrow & (-2, 4) \\
(2, 1) & \rightarrow & (0, 1) & \rightarrow & (0, 3) \\
(1, 1) & \rightarrow & (1, 1) & \rightarrow & (2, 2) \\
(3, 2) & \rightarrow & (1, 2) & \rightarrow & (2, 5) \\
(2, 2) & \rightarrow & (2, 2) & \rightarrow & (4, 4) \\
(1, 2) & \rightarrow & (3, 2) & \rightarrow & (6, 3) \\
(3, 3) & \rightarrow & (3, 3) & \rightarrow & (6, 6) \\
(2, 3) & \rightarrow & (4, 3) & \rightarrow & (8, 5) \\
(1, 3) & \rightarrow & (5, 3) & \rightarrow & (10, 4)
\end{array}
$$

To show that this property is true in general, let $\prec$ be the lexicographical order.

Theorem 4.2 *If the auxiliary iteration space is traversed in the lexicographical order, then the target iteration space is also traversed in the lexicographical order.*

Proof: $S_j = HS_k$, where H is a lower triangular matrix with positive diagonal. Let $\vec{k_1} \prec \vec{k_2}$ be two iterations in the auxiliary iteration space, and $\vec{d_1} = \vec{k_2} - \vec{k_1}$ be the distance of the two vectors. Clearly $\vec{d_1} \succ 0$. To see that the lexicographical order is preserved, consider the new distance $\vec{d_2}$.

$$\vec{d_2} = \vec{j_2} - \vec{j_1} = H\vec{k_2} - H\vec{k_1} = H\vec{d_1}$$

If $\vec{d_1}(i)$, the ith element of $\vec{d_1}$, is the leading nonzero, $\vec{d_1}(i)$ must be positive, since $\vec{d_1} \succ 0$. Then the leading nonzero of $\vec{d_2}$ is $h_{ii}\vec{d_1}(i)$, which is also positive. Therefore $\vec{d_2} \succ 0$, and $\vec{j_1} \prec \vec{j_2}$. $\square$

This result yields a technique for code generation - decompose T into HU, generate the DO-loops for traversing the auxiliary space using the technique of Section 3 (or any other technique that works for unimodular matrices) and compute the target iteration space variables in the loop body. Using the bounds for the auxiliary space computed earlier, the target code for our running example is the following:

for p = -1, 5

  for q = max(1, $\lceil\frac{p+1}{2}\rceil$*), min(3,* $\lfloor\frac{p+3}{2}\rfloor$*)*

    $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$

    A[u+3,v] = (u+2v)/6;

Although this code avoids making conditional tests, it can be improved considerably. Notice that the computation of u is invariant in the inner loop; moreover, u is a linear function of the outer loop index and it can be strength reduced. Similarly, v is a linear function of p and q and it can be strength reduced. Although such optimizations can be left to a later optimization phase, it is preferable to use the induction variables u and v directly as the loop control variables instead of p and q. We show how to do this next.

4.2 Target Iteration Space

Since H is lower triangular, it is easy to convert the bounds in the auxiliary space into bounds in the target space. For our example, the relation between these two spaces is given by the following equation:

$$\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ -1 & 3 \end{pmatrix} \begin{pmatrix} p \\ q \end{pmatrix}$$

From the first equation, it follows that the bounds for u are the bounds of p multiplied by 2. Therefore, the bounds for u are the following:

$$-2 \leq u \leq 10$$

The bounds for v are the bounds of q multiplied by 3 with the offset $-p$ which is $-\frac{u}{2}$. Therefore, the bounds for v are the following:

$$-\frac{u}{2} + 3max(1, \lceil\frac{\frac{u}{2}+1}{2}\rceil) \leq v \leq -\frac{u}{2} + 3min(3, \lfloor\frac{\frac{u}{2}+3}{2}\rfloor)$$

These bounds on u are constant, and the bounds on v depend only on u. Therefore, these bounds can be used directly to construct the loop nest, as is shown in Figure 1(b). The general algorithm is given in Figure 5.

The proof of correctness of this algorithm depends on the following lemma and the fact that the diagonal elements of H are positive, and is omitted.

Lemma 4.1 *Let P1 = $(j_1, ..., j_n)$ be a point in the target iteration space that is the image of a point P2 = $(k_1, ..., k_n)$ in the auxiliary space. Any co-ordinate j_i can be written as a function of $k_1, ..., k_i$; similarly, k_i can be written as a function of $j_1, ..., j_i$.*

Proof: Follows from the observation that any leading principal sub-matrix $H[1:i, 1:i]$ of H is lower triangular and non-singular, and that the inverse of a lower triangular matrix is also lower triangular [9]. □

To complete the generation of code, we need to skip over points within the bounds that are not in the target iteration space. This would be difficult to do if these points appeared in some irregular pattern within the loop nest bounds; fortunately, we can show that this is not the case. In fact, we show that it suffices to use DO-loops with constant step sizes, and that these step sizes are the integers in the diagonal of the Hermite form.

For the working example, H has the diagonal $[2, 3]$, which means that the loop step is 2 for the outer loop, and 3 for the inner loop. More generally, we have the following theorem.

Theorem 4.3 *The positive integers on the diagonal of the Hermite form are the gaps in each dimension.*

Proof: Consider two points P1 = $(k_1, k_2, ...k_i, k_{(i+1)}...k_n)$ and P2 = $(k_1, k_2, ...k_i{+}1, a_{(i+1)}...a_n)$ in the auxiliary space which have the same co-ordinates in the first $i-1$ dimensions. Let P3 and P4 be their images in the target space. From Lemma 4.1, it follows that P3 and P4 have the same co-ordinate for the first $(i-1)$ dimensions; moreover, their difference in the ith dimension is h_{ii} since H is lower triangular. □

Hence a loop nest can be constructed to traverse the target space directly. For our running example, the target program is the following:

for u = -2, 10 step 2

for v $= -\frac{u}{2} + 3max(1, \lceil\frac{\frac{u}{2}+1}{2}\rceil), -\frac{u}{2} + 3min(3, \lfloor\frac{\frac{u}{2}+3}{2}\rfloor)$ *step 3*

A[u+3,v] = (u+2v)/6;

Notice the auxiliary space is used only to compute the bounds for the target space.

4.3 Sparse Source Iteration Space

So far, we have considered only the case when the source iteration space is dense. Our technique also works when the source iteration S_i is sparse as long as the source space is *regular*. A *regular* sparse space is one that can be represented by an integer lattice Λ_i. The *base* space S_b is always dense. The lattice basis Λ_i can be thought of as a Λ-transformation from S_b to S_i. The bounds for S_b can be computed by the Fourier-Motzkin elimination. Then any Λ-transformation T on S_i can be considered as a new Λ-trasnformation $T\Lambda_i$ on the base space S_b, since Λ-transformations are closed under composition.

This observation lets us handle source loops in which lower bounds are affine functions of loop indices, upper bounds are piece-wise affine functions of loop indices and step sizes are constant, as shown below.

Theorem 4.4 *The following loop nest with constant steps is regular.*

for $i_1 = 0$ to ub_1 step s_1

for $i_2 = a_{21}i_1$ to ub_2 step s_2

...

for $i_n = a_{n1}i_1 + a_{n2}i_2 + ... + a_{n(n-1)}i_{n-1}$ to ub_n step s_n

Proof: Without loss of generality, the loop nest can be *shifted* so that $\vec{0}$ is a loop point. This just changes the *origin* of the lattice. It is easy to show that the triangular matrix S

whose jth row comes from the coefficients of the lower bound and the loop step of the jth loop forms a basis for the lattice of the loop points.

$$S = \begin{pmatrix} s_1 & 0 & . & 0 \\ a_{21}s_1 & s_2 & . & 0 \\ . & . & . & . \\ a_{n1}s_1 & a_{n2}s_2 & . & s_n \end{pmatrix}$$

5 Completion Procedure

One advantage of using integer non-singular matrices is that there is a simple *completion procedure* that takes the first few rows of a desired transformation matrix and generates a complete transformation matrix that respects dependences. As an example that illustrates the need for a completion procedure, consider parallelizing the following program for a MIMD machine [3].

for i = 4, 8
for j = 3, 8
A[i, j] = A[i-3, j-2] + 1;

The dependence matrix for this program is $D = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$

The outermost loop is parallel if and only if it does not carry any dependences; that is, the first entry of every dependence vector is 0. In our example, the outermost loop is not a parallel loop, since iteration i depends on iteration $i-3$. We can transform the loop nest into one in which the outermost loop is parallel if we can find a transformation T such that every entry in the first row of TD is 0. Therefore, the condition that must be satisfied for transformation $T = \begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix}$ to achieve the goal is that $\begin{pmatrix} t_{11} & t_{12} \end{pmatrix}\begin{pmatrix} 3 \\ 2 \end{pmatrix} = 0$. The condition can be satisfied by choosing $t_{11} = 2$ and $t_{12} = -3$. This determines the first row of the transformation matrix, and now we must add additional row(s) to get a non-singular matrix that respects all the dependences of the loop nest.

A second example that illustrates the need for a completion procedure is the generation of code for a NUMA architecture starting from a language like FORTRAN-D with user-specified data distributions. This problem is discussed at length in a companion paper; here, we will simply show an example [11]. Consider the program of Figure 6(a). Assume that a two dimensional array B has a wrapped column distribution, i.e the columns of an array are distributed in a round-robin manner to the processors: if N is the number of processors, then processor 0 gets columns *0, N, 2N* and so on, while processor 1 gets columns *1, N+1, 2N+1,* etc.

Distributing iterations of the outer loop among the processors (Figure 6(a)) results in processor p executing iterations p, $p + N$, etc. Consider accesses to elements of array B. There will be many remote memory accesses. Now, consider the transformed program of Figure 6(b). If we distribute the outermost loop among the processors as before, there are no remote accesses to B.

For this particular goal, we need to transform the loop nest so that the subscript in the column index of B is the new outer loop index. This means that if the transformation is $T = \begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix}$ the first row of the transformation must be (2 -1). Again, we need to invoke a completion procedure to add additional row(s) to get the full matrix.

We will refer to this as the *completion* of a *partial transformation.*

5.1 Completion Procedure

Our completion procedure requires that the following precondition be satisfied.

Precondition: The partial transformation must have full row rank[1], and should not violate dependences.

These conditions are reasonable: if a row of the transformation matrix is linearly de-

[1] This means that each row is linearly independent of the others.

pendent on the others, it is clearly impossible to generate a non-singular matrix by adding additional rows. Similarly, if some row of the partial transformation violates one or more dependences, this cannot be rectified by extending the matrix with additional rows.

First, we delete all dependence vectors that are carried by the loops corresponding to the rows of the partial transformation, since they do not have to be considered when filling in the rest of the matrix. The completion procedure works by finding a vector that is independent of the existing row vectors in the partial transformation and within 90 degrees of each dependence vector. This vector is appended a new row to the partial transformation and all dependencies carried by the loop corresponding to this row are dropped from consideration. This technique is repeatedly applied until there are no further dependencies to be satisfied, at which point we can apply standard linear algebra techniques to complete the generation of a non-singular matrix.

To find the desired rows, we enforce the following invariant:

Invariant: The dependence vectors are in the orthogonal complement of the subspace spanned by the rows of the partial transformation.

This means that if d is a dependence vector and r is a row of the partial transformation, then $r^T d$ is zero.

To perform the completion, we find a vector that is within 90 degrees of every dependence vector and strictly within 90 degrees with at least one dependence vector. This ensures that the partial transformation obtained by appending the new row will not violate dependences. This vector can be found by looking for the first row of the dependence matrix with nonzero entries. Let k be that row index.

Lemma 5.1 *e_k is within 90 degrees of every dependence vector, and strictly within 90 degrees of at least one dependence vector.*

Proof: It is easy to check that e_k satisfy the condition, since for any dependence vector d, $e_k^T d \geq 0$, and $e_k^T d > 0$ for at least one d. $\square$

But e_k is not necessarily linearly independent of the rows in the partial transformation. Therefore, we project e_k to the orthogonal complement of the subspace spanned by the rows of the partial transformation so that the projected vector is linearly independent of the existing rows. If P^T is the partial transformation, it is easy to see that the projector is $Q = (I - P(P^T P)^{-1} P^T)$. The projected vector is $y = Qe_k$. Let $x = cy$ for some positive scaling number c that makes all of the entries integers and relative primes. For $P^T = (2\ \text{-}3)$ and $D = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$ the projection is shown in Figure 7.

Theorem 5.1 *The projected vector y is linearly independent of the rows and within 90 degrees of every dependence vector.*

Proof:

- Linear independence:

  We can prove a even stronger result, i.e. y is orthogonal to the rows in P^T.

$$P^T y = P^T (I - P(P^T P)^{-1} P^T) e_k = 0$$

- Within 90 degrees:

  For any dependence vector d, $y^T d = e_k^T Q^T d$. Q is symmetric and d is already in the orthogonal complement by the invariant. This means $Q^T d = Qd = d$. Then $y^T d = e_k^T d$. Hence y is within 90 degrees of every dependence vector by Lemma 5.1.

□

The complete algorithm is presented in Figure 8 and Figure 9.

Theorem 5.2 *The algorithm Completion generates a legal Λ-transformation.*

Proof: immediately follows from Theorem 5.1. □

We now apply the algorithm to the examples in the previous section. The first example can be satisfied by choosing $t_{21} = 3$ and $t_{22} = 2$. The new dependence vector is $\begin{pmatrix} 0 \\ 13 \end{pmatrix}$, which means that the new outer loop is a parallel loop. The matrix $\begin{pmatrix} 2 & -3 \\ 3 & 2 \end{pmatrix}$ satisfies the conditions. For the second example, there is no dependence, so the full transformation is $\begin{pmatrix} 2 & -1 \\ 0 & 1 \end{pmatrix}$.

5.2 Discussion

The completion technique discussed here works even when dependences are represented using direction vectors. There is considerable flexibility in the choice of the projector, and we have shown just one possibility; the choice of the most desirable projector will depend on the application. A completion algorithm to generate a *unimodular* matrix for the special case of a 2-level nested loop was given by Banerjee (3); however, it is unclear to us that this algorithm can be extended in any simple way to handle general loop nests.

6 Related Work

The use of compound loop transformations to parallelize loop nests goes back to Lamport's hyperplane method (10). This method used distance vectors to generate a transformed loop nest for exploiting parallelism. The connection with unimodular matrices was made by Dowling who showed how to use such matrices to parallelize loop nests (7). Banerjee related the unimodular matrix approach to compositions of standard loop transformations like permutation, skewing and reversal (3). The extension to direction vectors was made by Wolf and Lam, who also used the unimodular framework to restructure loop nests for parallelism and for cache locality (17, 18). At a more abstract level, Ancourt and Irigoin have developed algorithms for scanning polyhedra in iteration spaces using loop nests (2). There exist a number of other extensions to the unimodular framework. Lu uses general injective functions from the source to the destination iteration space, but the code he generates suffers from conditionals in the body of the loop (12). Other approaches to extending the unimodular framework are in Ramanujam (14), Barnett and Lengauer (4), and Sarkar amd Thekkath (15). Hermite decomposition was used by Gallivan, Jalby and Gannon to estimate data transfer volumes in memory hierarchies [8]. We are not aware of any prior work on general completion procedures.

7 Conclusions

We have introduced a loop transformation framework called Λ-transformations based on integer non-singular matrices. Efficient code can be generated for target loop nests using integer lattice theory. We have also presented a simple completion algorithm that generates correct transformations from partial transformations.

8 Acknowledgments

We would like to thank Mark Charney, Tom Coleman, Shmuel Onn, Danny Ralph, Paul Stodghill, Eva Tardos, Mike Todd, Charles Van Loan, Zhijun Wu for listening to the ideas and for their comments.

References

[1] R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Progamming Languages and Systems*, 9(4):491–542, October 1987.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Third ACM Symposium on Principles and Practice of Parallel Programming*, pages 39–50, April 1991.

[3] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[4] M. Barnett and C. Lengauer. Loop parallelization and unimodularity. Technical Report ECS-LFCS-92-197, University of Edinburgh, 1992.

[5] J. W. S. Cassels. *An introduction to the geometry of numbers.* Berlin, Springer, 1959.

[6] G. B. Dantzig and B. C. Eaves. Fourier-motzkin elimination and its dual. *Journal of Combinatorial Theory(A)*, 14:288–297, 1973.

[7] M. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, 1990.

[8] K. Gallivan, W. Jalby, and D. Gannon. On the problem of optimizing data transfers for complex memory systems. *International Conference on Supercomputing*, pages 238–253, 1988.

[9] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins University Press, 1989.

[10] L. Lamport. The parallel execution of do loops. *Communications of the ACM*, pages 83–93, February 1974.

[11] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. In *Proc. 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[12] L. Lu. A unified framework for systematic loop transformations. In *3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 28–38, April 1991.

[13] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of ACM*, 29(12):1184–1201, December 1986.

[14] J. Ramanujam. Non-unimodular transformations of nested loops. In *Proc. of Supercomputing*, 1992.

[15] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations. In *Proc. ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, 1992.

[16] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1986.

[17] M. Wolf and M. Lam. A data locality optimizing algorithm. In *Proc. ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 30–44, June 1991.

[18] M. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, October 1991.

[19] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London, 1989.

for i = 1, 3

for j = 1, 3

A[4j-2i+3, i+j] = j;

(a) The original code

for u = -2, 10 step 2

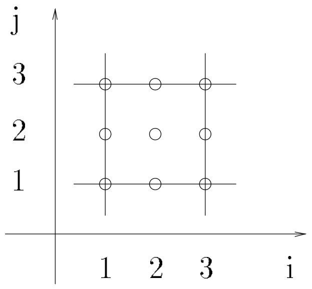
for v = $-\frac{u}{2} + 3max(1, \lceil \frac{\frac{u}{2}+1}{2} \rceil),$

$-\frac{u}{2} + 3min(3, \lfloor \frac{\frac{u}{2}+3}{2} \rfloor)$

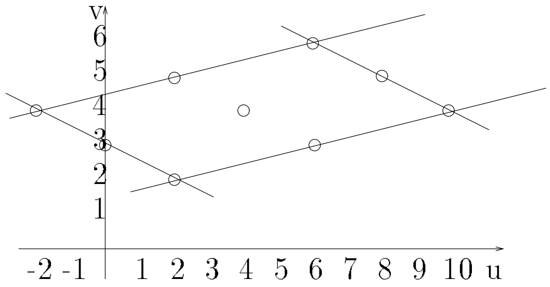
step 3

A[u+3, v] = (u + 2v)/6;

(b) The target code



(c) The original iteration space



(d) The target iteration space

$$1 \leq \quad i \quad \leq 3$$
$$1 \leq \quad j \quad \leq 3$$

(e) Loop Bounds

$$-2 \leq u \leq 10$$
$$max((u+6)/4, (6-u)/2) \leq v$$
$$v \leq min((u+18)/4, (18-u)/2)$$

(f) Image of Bounds

Figure 1: The working example

for i = 1, 3

for j = 1, 3

A[i, 2j] = j

The original loop nest

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

for u = 1, 3

for v = 1, 3

A[v, 2u] = u

(a) loop interchange

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

for u = 1, 3

for v = -3, -1

A[u, -2v] = -v

(b) loop reversal

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

for u = 1, 3

for v = u+1, u+3

A[u, 2(v-u)] = v-u

(c) loop skewing

$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

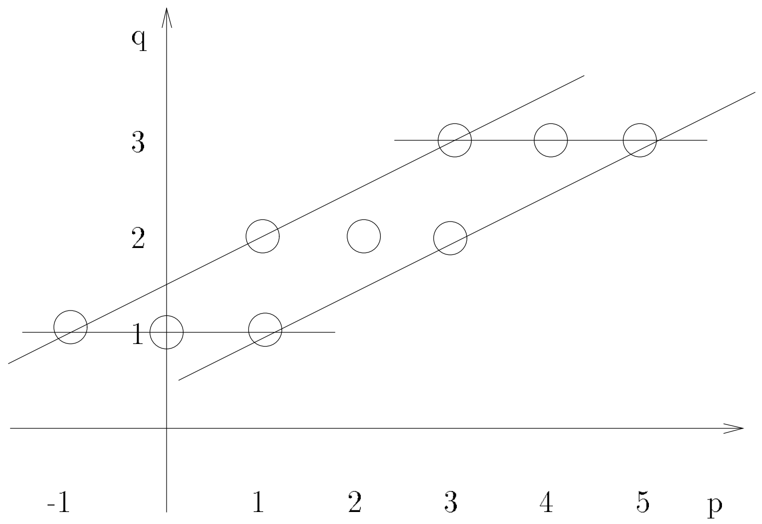
for u = 1, 3

for v = 2, 6, 2

A[u, v] = v/2

(d) loop scaling

Figure 2: Primitive Transformations



$$
\begin{aligned}
-1 \leq \quad & p \quad \leq 5 \\
max(1, \frac{p+1}{2}) \leq \quad & q \quad \leq min(3, \frac{p+3}{2})
\end{aligned}
\qquad
\begin{aligned}
-1 \leq \quad & p \quad \leq 5 \\
max(1, \lceil\frac{p+1}{2}\rceil) \leq \quad & q \quad \leq min(3, \lfloor\frac{p+3}{2}\rfloor)
\end{aligned}
$$

(a) Image of Bounds (b) Exact Bounds

Figure 3: Dense Iteration Space

Input: *An $n \times n$ integer matrix T.*

Output: *The Hermite form H of T and a unimodular matrix U.*

Algorithm Hermite

begin

  $U = I$, where I is an $n \times n$ identity matrix.

  For i = 1 to n do

    / Consider the submatrix T[i:n, i:n] */*

      While T[i, i+1:n] $\neq \vec{0}$ do

        Apply elementary column operations U_e to make T[i, i]

          positive and T[i, i+1:n] zero.

        $U = U_e^{-1} U$

      End-While

  End-For

  $H = T$

end

Figure 4: Computing the Hermite form

Input: *The Hermite form H, and the bounds of the auxiliary space S_k*

Output: *The bounds of the target space S_j.*

Algorithm Bounds(l^k, u^k) : (l^j, u^j)

begin

$S_k = H^{-1}S_j$

For i = 1 to n do

/* *Compute the offset by replacing* k_1, .., $k_{(i-1)}$ *by* j_1, .. $j_{(i-1)}$ */

$v_i = h_{i1}k_1 + .. + h_{i(i-1)}k_{(i-1)} = f_i(j_1, .., j_{(i-1)})$

/* *Compute lower bound with* k_1, .., $k_{(i-1)}$ *in* l_i^k

replaced by j_1, .. $j_{(i-1)}$ *using* $S_k = H^{-1}S_j$. */

$l_i^j = v_i + h_{ii}l_i^k$

/* *Compute upper bound with* k_1, .., $k_{(i-1)}$ *in* u_i^k

replaced by j_1, .. $j_{(i-1)}$ *using* $S_k = H^{-1}S_j$. */

$u_i^j = v_i + h_{ii}u_i^k$

End-For

end

Figure 5: Computing the loop bounds

for $i = l_i,\ u_i$

  for $j = l_j,\ u_j$

    B[j, 2i-j] = 0;

(a)

for $u = l_u,\ u_u$

  for $v = l_v,\ u_v$

    B[expr, u] = 0;

(b)

Figure 6: Transformation for Data Locality

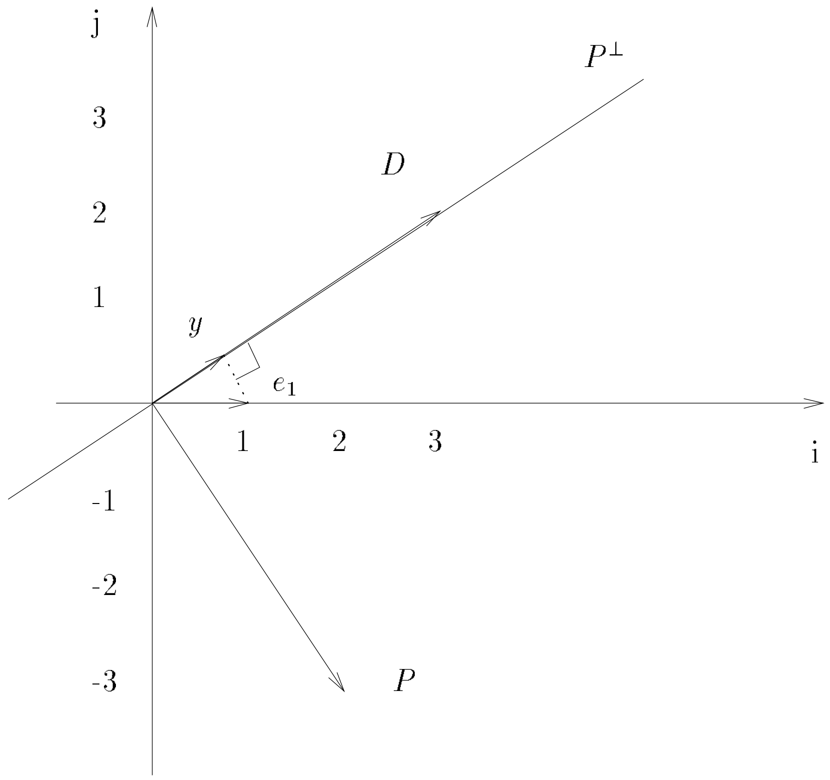


Figure 7: Extending Partial Transformation by Projection

Input: *An* $m \times n$ *partial transformation matrix* P^T

and a dependence matrix D*.*

Output: *An* $n \times n$ *legal transformation matrix* T*.*

Algorithm Completion(P^T*, D) : Matrix*

begin

/* *Let* P_i^T *be row* i *of* P^T*, and* d_i *be column* i *of* D */

For i = *1,* m

$f^T = P_i^T D$

$D = D - d_j$*, where* $f[j] > 0$

End-For

$r = m + 1;$

While D *is not empty do*

let k *be the first nonzero row of* D*;*

$x = c(I - P(P^T P)^{-1} P^T) e_k;$

where c *is a positive number that makes* x

an integer vector and the entries relative primes.

$D = D - d$*, if* d *is a dependence vector and* $d[k] > 0$

$P_r^T = x^T;$

$r = r + 1;$

End-While

Figure 8: Computing a Legal Full Transformation (part 1)

```
  R = I, where I is an n × n identity matrix.
  For i = 1, r-1 do
    /* Consider the submatrix P^T[i:m, i:n] */
    Apply the elementary column operations to make P^T[i, i] nonzero
      and P^T[i, i+1:n] zero.
    If columns i and j have been exchanged Then
      exchange rows i and j of R
    End-If
  End-For
  return(append(P^T, R[r:n, 1:n]));
end
```

Figure 9: Computing a Legal Full Transformation (part 2)