Reshaping Access Patterns for Generating Sparse Codes*

Aart J.C. Bik, Peter M.W. Knijnenburg and Harry A.G. Wijshoff
High Performance Computing Division
Department of Computer Science
Leiden University
P.O. Box 9512, 2300 RA Leiden, the Netherlands
ajcbik@cs.leidenuniv.nl

Abstract

In a new approach to the development of sparse codes, the programmer defines a particular algorithm on dense matrices which are actually sparse. The sparsity of the matrices as indicated by the programmer is only dealt with at compile-time. The compiler selects appropriate compact data structure and automatically converts the algorithm into code that takes advantage of the sparsity of the matrices. In order to achieve efficient sparse codes, the compiler must be able to reshape some access patterns before a data structure is selected. In this paper, we discuss a reshaping method that is based on unimodular transformations.
Index Terms: Program Transformations, Restructuring Compilers, Sparse Matrices.

1 Introduction

Because of the inherent complexity of sparse codes, it is worthwhile to consider whether sparse codes can be generated automatically. In [7, 9] we have proposed an approach in which the algorithm is defined on dense matrices and automatically converted into sparse code. This implies that all operations can be defined on two dimensional arrays, which reduces the complexity of sparse codes development and maintenance, and enables more standard compiler optimizations [6]. Annotations are used to identify which of the declared dense data structures are actually sparse. Matrices which are actually sparse are referred to as **implicitly sparse matrices**, because the programmer does not have to deal with the sparsity explicitly. This burden is placed on the compiler which, in order to exploit the sparsity as much as possible, selects a compact data structure and generates corresponding sparse code. This approach can be used for sparse matrices with arbitrary nonzero structures, but if the matrices are available on file, compile-time analysis of these matrices [10] or some kind of annotation can be used to enable the compiler to take advantage of certain properties of the nonzero structures.

However, as alluded to in previous work, the application of standard program transformations is essential for the generation of efficient sparse codes. Because the input program operates on two-dimensional arrays that support direct access, the programmer is free to use all kind of access patterns through the arrays. In some cases, preference is given to a particular kind of access patterns. For instance, column-wise access is often used in FORTRAN to enable vectorization or to improve the spatial locality of the program. However, in general, all kind of access patterns through the implicitly sparse matrices can occur. Sparse data structures usually only support efficient generation of entries along one kind of access patterns. This is caused by the fact that these data structures do not only consists of primary storage to store numerical values, but also of overhead storage used to access these values and to reconstruct the structure of the matrix. Supporting several kind of access patterns would require data structures in which the savings in storage for numerical values is diminished by the required amount of overhead storage, or for which the maintenance overhead would increase the execution time considerably. Therefore, before an organization for the sparse data structure is selected, reshaping the access patterns can resolve conflicts between different kind of access patterns. This can improve the performance of the resulting sparse code considerably.

The outline of the rest of this paper is as follows.

*Support was provided by the Foundation for Computer Science (SION) of the Netherlands Organization for the Advancement of Pure Research (NWO) and the EC Esprit Agency DG XIII under Grant No. APPARC 6634 BRA III.

In section 2, we give a motivation for the necessity of reshaping techniques in order to generate efficient sparse codes. In section 3 we give an outline of unimodular transformations, which will be used for the reshaping techniques. These techniques are discussed in section 4. We give some examples in section 5. Finally, in section 6, conclusions and issues for future research are stated.

2 Importance of Reshaping

In this section, we illustrate that reshaping access patterns for implicitly sparse matrices is essential for the automatic generation of efficient code.

2.1 Definitions

Most occurrences of an implicitly sparse matrix A appear in a nested loop of degree n. We assume that the subscript functions of each occurrence can be represented by a single mapping $F_A(\vec{\mathtt{I}}) : \mathbf{Z}^n \rightarrow \mathbf{Z}^2$ of the form $F_A(\vec{\mathtt{I}}) = \vec{m} + M \cdot \vec{\mathtt{I}}$, where $\vec{\mathtt{I}}$ denotes the surrounding loop indices, that is $\vec{\mathtt{I}} = (\mathtt{I}_1, \ldots, \mathtt{I}_n)^T$:

$$F_A(\vec{\mathtt{I}}) = \begin{pmatrix} m_{10} \\ m_{20} \end{pmatrix} + \begin{pmatrix} m_{11} & \ldots & m_{1n} \\ m_{21} & \ldots & m_{2n} \end{pmatrix} \cdot \vec{\mathtt{I}}$$

Each occurrence induces **access patterns**, consisting of the index sets of the elements that are referenced in one execution of the innermost loop. The **direction** of an access pattern is defined as $\vec{d}_A = (m_{1n}, m_{2n})$. If either m_{1n} or m_{2n} is zero, the access patterns are called **row-** and **column-wise** respectively. All other nonzero directions are referred to as **diagonal-wise**. Access patterns for which $\vec{d}_A = \vec{0}$ holds are called **scalar-wise**. For the latter kind of access patterns, a direction is usually induced at a higher level i, for which m_{1i} or m_{2i} is nonzero. Two directions $\vec{d}$ and $\vec{d'}$ are called **linearly dependent** if $\vec{d} = \lambda \cdot \vec{d'}$ for some $\lambda \in \mathbf{R}$.

2.2 A Naive Approach

Consider, for example, the following fragment, where an operation $\vec{c} \leftarrow A \cdot \vec{b}$ is followed by an accumulation of particular elements in an implicitly sparse matrix:

```
      DO I = 1, M
        DO J = 1, N
S1:       C(I) += A(I,J) * B(J)
        ENDDO
      ENDDO
      DO J = 1, N / 2
        DO I = 1, M / 2
S2:       ACC += A(2*I,2*J)
        ENDDO
      ENDDO
```

In [9] we have shown that statements S_1 and S_2 only have to be executed for entries, i.e. elements that are stored explicitly in the compact data structure of A. If the organization of the selected data structure is consistent with the access pattern in these statements, a special construct can be generated that will iterate over all entries along a certain access pattern at run-time. Since S_1 and S_2 induce respectively row- and column-wise access patterns through A, one of the following fragments would result after selection of the data structure (see [7, 9] for details on the data structure and code generation):

```
row-wise storage of A:
      DO I = 1, M
        DO AD = ALOW(I), AHIGH(I)
          J = AIND(AD)
S1:       C(I) += AVAL(AD) * B(J)
        ENDDO
      ENDDO
      DO J = 1, N / 2
        DO I = 1, M / 2
          AD = LKP(AIND,ALOW(2*I),AHIGH(2*I),2*J)
S2:       IF (AD ≠ ⊥) ACC += VAL(AD)
        ENDDO
      ENDDO

column-wise storage of A:
      DO I = 1, M
        DO J = 1, N
          AD = LKP(AIND,ALOW(J),AHIGH(J),I)
S1:       IF (AD ≠ ⊥) C(I) += AVAL(AD) * B(J)
        ENDDO
      ENDDO
      DO J = 1, N / 2
        DO AD = ALOW(2*J), AHIGH(2*J)
          I = AIND(AD)
S2:       IF (MOD(I,2) = 0) ACC += AVAL(AD)
        ENDDO
      ENDDO
```

In the first fragment, a construct that iterates over entries can be used for S_1. This construct can also be used for S_2 in the second fragment, although a test is required because only the entries $a_{i,2*j}$ in a column for which $i \bmod 2 = 0$ holds, are operated on in the original dense fragment. If row-wise storage is selected, a lookup is required for S_2 to fetch the elements in the matrix. The value $\perp$ is returned if this element is not an entry. Consequently, we can skip the operations for these zero elements, although this test does not reduce the execution time [23]. Similarly, a lookup results for S_1 if column-wise storage is selected. These lookups are unwanted for two reasons. First, each lookup induces substantial overhead because in the worst case all entries in a whole row or column must be scanned in order to obtain the address of an entry, or to conclude that the element to be fetched is not an entry.

Second, no reduction in the number of iterations is obtained. For example, S_1 is executed $\mathtt{M} \cdot \mathtt{N}$ times in the second fragment, but only NZ times in the first, where NZ indicates the total number of entries in A.

2.3 Reshaping for Consistency

The problems arising in the previous section are caused by the inconsistency of access patterns of occurrences of the same matrix, i.e. the access patterns induced by the occurrences of A in S_1 and S_2 overlap and have linearly independent directions $(0, 1)$ and $(2, 0)$ respectively. Since the organization of the data structure can only support the storage of entries along access patterns in one direction, lookups must be generated for all occurrences inducing access patterns that are inconsistent with this data structure organization.

However, in some cases, reshaping techniques can be used to obtain consistency. For the previous fragment, interchanging the loops that surround S_3 converts the direction of the access patterns through A into $(0, 2)$. Consequently, if sparse row-wise storage is selected for A, code in which a construct iterates over entries can be generated for both S_1 and S_2. In table 1, we present some timings of the two versions of the previous section and the reshaped variant on one CPU of a CRAY C98/4256 for some matrices of the Harwell-Boeing Sparse Matrix Collection [17] in the appropriate storage format. The row-wise variant is preferable over the column-wise variant, because the lookup is executed less frequently in the first fragment than in the second, namely $\frac{1}{4} \cdot \mathtt{M} \cdot \mathtt{N}$ and $\mathtt{M} \cdot \mathtt{N}$ times respectively. However, the reshaped version is clearly superior, due to the elimination of all lookups.

Although this experiment is rather simple, it illustrates the most important objective in the generation of sparse codes, namely that the number of operations performed must be kept proportional to the number of entries in the sparse matrix [14, 16, 23]. Skipping operations on zeros by means of conditionals is useless. Scanning sparse data structure to obtain an entry must be avoided as much as possible. For the automatic data structure selection and sparse code generation method this implies that it is very important to achieve consistency between all the access patterns through an implicitly sparse matrix. In this case the generation of constructs that limit the number of operations performed, become feasible.

Matrix	Row	Column	Reshaped
gre_1107	0.4	1.5	$2.1 \cdot 10^{-3}$
jagmesh1	0.3	1.1	$1.7 \cdot 10^{-3}$
orani678	2.2	8.8	$6.4 \cdot 10^{-3}$
steam2	0.1	0.5	$1.4 \cdot 10^{-3}$

Table 1: Execution Time in seconds (Cray C98/4256)

3 Unimodular Matrices

In this section we give an outline of the general approach to iteration-level loop transformations in terms of unimodular matrices. For an extensive overview of the theory, consult [4, 5, 15, 25, 26].

3.1 Loop Transformations

Every iteration-level loop transformation on n perfectly nested loops with stride 1 and regular loop bounds, consisting of a combination of loop interchanging, loop skewing, or loop reversal (see e.g. [1, 22, 24, 27, 28, 29]) can be modeled by a mapping between the **original** and **target iteration space**, namely a linear transformation that is represented by a unimodular matrix U. A **unimodular matrix** is an $n \times n$ integer matrix, i.e. all elements are integers, for which $|\det(U)| = 1$ holds. Each iteration $\vec{\imath}$ in the original iteration space is mapped to an iteration $\vec{\imath}' = U \cdot \vec{\imath}$ in the target iteration space. Because iterations in the target iteration space are also traversed in lexicographic order, application of a transformation effectively results in a new execution order on the instances.

Because each unimodular matrix can be decomposed into a number of such elementary loop transformations, and conversely, any combination of iteration level loop transformations is represented by a unimodular matrix, this approach offers more flexibility than the traditional step-wise application of loop transformations, where the usefulness and validity of each transformation is considered separately.

3.2 Application of a Transformation

Application of a unimodular transformation U is valid, if each data dependence in the original nesting is satisfied in the resulting nesting. Dependence distance vectors provide a convenient representation of data dependences. If iteration $\vec{\imath}'$ depends on iteration $\vec{\imath}$, then $\vec{\imath} + \vec{d} = \vec{\imath}'$ for some distance vector $\vec{d}$. Induced by the sequential semantics of DO-loops, iterations are executed in lexicographic order. Consequently, each distance vector of a loop-carried data dependence is **lexicographically positive**, denoted by $\vec{d} \succ \vec{0}$, i.e. its leading component (first nonzero component) is positive. Since U is a linear transformation, $U \cdot \vec{\imath}' - U \cdot \vec{\imath} = U \cdot (\vec{\imath}' - \vec{\imath})$. Consequently, application of a unimodular transformation U is valid if and only if $U \cdot \vec{d} \succ \vec{0}$ for each dependence distance $\vec{d} \neq \vec{0}$ in the original nest.

In [25, 26], a more abstract representation of data dependences, referred to as dependence directions, is incorporated in the validity test. Each component of a general dependence vector $\vec{d}$ can represent a possibly infinite range of integers. Directions '<', '>', '=', and '∗' correspond to the ranges $[1, \infty]$, $[-\infty, -1]$, $[0, 0]$, and $[-\infty, \infty]$ respectively. A distance component d is denoted by the degenerate range $[d, d]$. If all dependences are represented by lexicographically positive dependence vectors, i.e., each component is an interval $[l, r]$ with $l > 0$, dependence directions can also be handled by defining an arithmetic on these vectors. Addition of ranges and multiplication of a range by a scalar are defined as respectively $[l, r] + [l', r'] = [l + l', r + r']$ and $s \cdot [l, r] = [s \cdot l, s \cdot r]$, if $s \geq 0$, or range $[s \cdot r, s \cdot l]$ otherwise. Operations on ∞ are as expected (e.g. $0 \cdot \infty = 0$ and $-1 \cdot -\infty = \infty$). Using this new arithmetic, application of U is valid, if $U \cdot \vec{d} \succ \vec{0}$ for all dependence vectors $\vec{d}$. The converse implication, however, does not hold. If loop skewings are used in U, dependence information may be lost.

The application of a unimodular matrix to a loop nest is implemented by rewriting the loop-body and generating appropriate loop bounds. Since $\vec{\mathtt{I}} = U^{-1} \cdot \vec{\mathtt{I}}'$ holds, the new body is obtained by replacing each $\mathtt{I}_j$ in the original body accordingly. The new loop bounds are determined by application of Fourier-Motzkin elimination [2, 13, 19] to the system of inequalities that is obtained by substituting $U^{-1} \cdot \vec{\mathtt{I}}'$ for $\vec{\mathtt{I}}$ in the system defined by the original loop bounds. In [12], we present an implementation of Fourier-Motzkin elimination in which only integer arithmetic is involved and also present simplification methods to eliminate all redundant constraints from such a system, thereby improving the efficiency of the generated code.

3.3 Example

Consider, for example, application of the unimodular matrix U to the following loop:

```
DO I1 = 0, 50
  DO I2 = 0, 50 - I1
    DO I3 = 0, 50
      L(I1,I2,I3)
    ENDDO
  ENDDO
ENDDO
```

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

$$U^{-1} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & -1 & -1 \end{pmatrix}$$

We assume that application is valid. The resulting loop-body is obtained by replacing $\vec{\mathtt{I}}$ according to equation $\vec{\mathtt{I}} = U^{-1} \cdot \vec{\mathtt{I}}'$. Application of Fourier-Motzkin elimination to the system of inequalities that is obtained by substituting $U^{-1} \cdot \vec{\mathtt{I}}'$ for $\vec{\mathtt{I}}$ in the original system is used to rewrite the system into a format that can be used to generate the resulting bounds. Replacing $\vec{\mathtt{I}}$ by $U^{-1} \cdot \vec{\mathtt{I}}'$ in the original system yields the following system in the new loop indices:

$$\begin{array}{rcccl} 0 & \leq & \mathtt{I}_2' & \leq & 50 \\ 0 & \leq & \mathtt{I}_3' & \leq & 50 - \mathtt{I}_2' \\ 0 & \leq & \mathtt{I}_1' - \mathtt{I}_2' - \mathtt{I}_3' & \leq & 50 \end{array}$$

The appropriate form is obtained by application of Fourier-Motzkin elimination to this system:

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{I}_3' & \leq & 50 - \mathtt{I}_2' & \mathtt{I}_3' & \leq & \mathtt{I}_1' - \mathtt{I}_2' \\ 0 & \leq & \mathtt{I}_3' & \mathtt{I}_1' - \mathtt{I}_2' - 50 & \leq & \mathtt{I}_3' \\ -\mathtt{I}_2' & \leq & 0 & \mathtt{I}_2' & \leq & 50 \end{array}$$

$\downarrow$ Eliminate $\mathtt{I}_3'$ $\downarrow$

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{I}_2' & \leq & 50 & \mathtt{I}_2' & \leq & \mathtt{I}_1' \\ 0 & \leq & \mathtt{I}_2' & \mathtt{I}_2' & \leq & 50 \\ \mathtt{I}_1' & \leq & 100 & 0 & \leq & 50 \end{array}$$

$\downarrow$ Eliminate $\mathtt{I}_2'$ $\downarrow$

$$\begin{array}{rcl@{\qquad}rcl} \mathtt{I}_1' & \leq & 100 & 0 & \leq & \mathtt{I}_1' \\ 0 & \leq & 50 & 0 & \leq & 50 \\ 0 & \leq & 50 & & & \end{array}$$

The following code is generated, after redundant inequalities have been eliminated:

```
DO I'1 = 0, 100
  DO I'2 = 0, MIN(50,I'1)
    DO I'3 = MAX(0,I'1-I'2-50), MIN(50-I'2,I'1-I'2)
      L(I'2,I'3,I'1-I'2-I'3)
    ENDDO
  ENDDO
ENDDO
```

The conversion of the original iteration space into the target iteration space is illustrated in figure 1. It can be easily determined which iterations in the original iteration space are executed in one iteration of the resulting loops. For example, since $(1, 1, 1)$ is the first row of U, all iterations in the original iteration space that lie in the plane $\mathtt{I}_1 + \mathtt{I}_2 + \mathtt{I}_3 = c_1$, for $c_1 \in \mathbf{Z}$, are executed in iteration $\mathtt{I}_1' = c_1$. Moreover, these planes are traversed in the direction $(1, 1, 1)$ in successive iterations of the outermost resulting loop, as illustrated in the first picture of figure 2. Because $(1, 0, 0)$ is the second row of U, all iterations in the intersection of the plane defined by the outermost

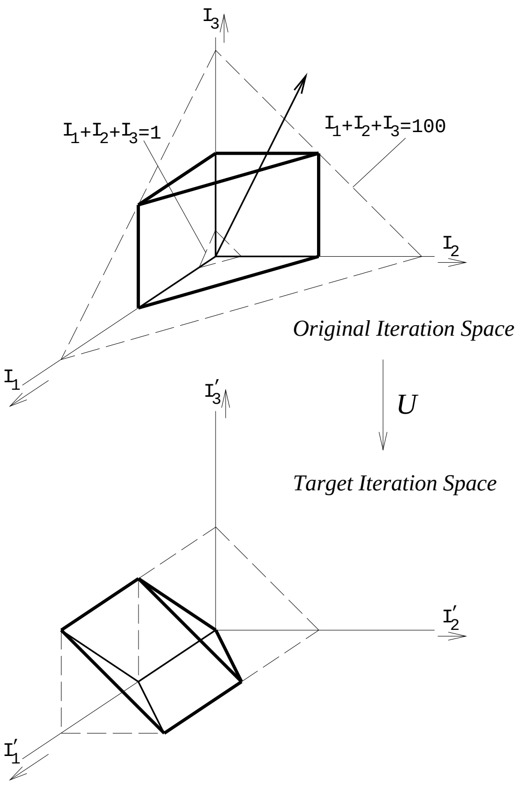

Figure 1: Application of U

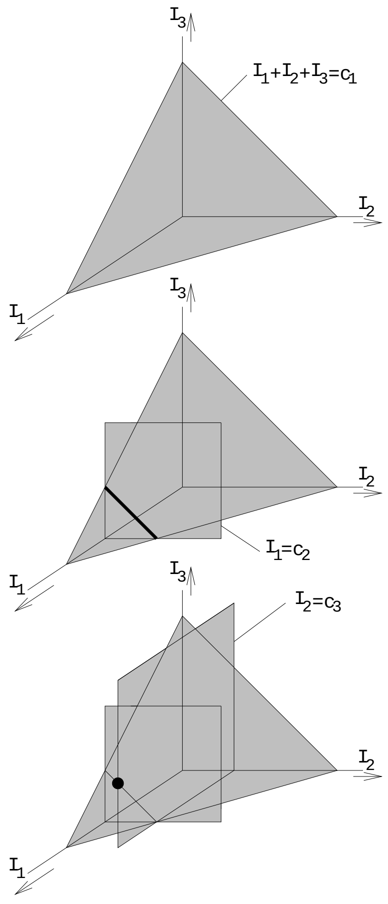

Figure 2: Original Iteration Space Traversal

loop and the plane $\mathtt{I}_1 = c_2$, for $c_2 \in \mathbf{Z}$, are executed in iteration $\mathtt{I}'_2 = c_2$. This line has direction $(0, 1, -1)$, as is shown in the second picture of figure 2. An iteration along this line that also lies in the plane of which the normal vector is defined by the last row of U, i.e. $\mathtt{I}_2 = c_3$, for $c_3 \in \mathbf{Z}$, is executed in the iteration $\mathtt{I}'_3 = c_3$, as illustrated in the last picture of figure 2. In general, the rows of an $n \times n$ unimodular matrix U, for $n \geq 2$, define the normal vectors of so-called hyperplanes in the n-dimensional iteration space. The intersection of such hyperplanes determine which original iteration is executed in a particular iteration $\vec{\mathtt{I}}' = \vec{c}$ of the resulting loop, according to equation $U \cdot \vec{\mathtt{I}} = \vec{c}$. Consequently, all iterations that are executed in a complete execution of the innermost resulting loop are along a line through the original iteration, formed by the intersection of hyperplanes defined by the first $n - 1$ rows of U. Moreover, since $\vec{\mathtt{I}} = U^{-1} \cdot \vec{\mathtt{I}}'$, the last column of U^{-1} defines the direction of this line. Along this line, original iterations are executed in successive iterations of the innermost resulting loop. For the previous transformation, for instance, this direction is $(0, 1, -1)$, i.e. the last column of U.

4 Reshaping Access Patterns

In this section, we present the construction of a unimodular transformation which will be used for changing the direction of the access patterns of occurrences of implicitly sparse matrices.

4.1 Simple Method for Reshaping

Consider the following general framework, in which an occurrence of an implicitly sparse matrix A appears in a perfectly nested loop at nesting depth $n \geq 2$, and where the subscripts functions are represented by a mapping $F_A(\vec{\mathtt{I}}) = \vec{m} + M \cdot \vec{\mathtt{I}}$ as defined in section 2.1:

```
      DO $\mathtt{I}_1 \in V_1$
        ...
          DO $\mathtt{I}_n \in V_n$
$S_1$:        ... A( $F_A(\vec{\mathtt{I}})$ ) ...
          ENDDO
        ...
      ENDDO
```

In order to achieve that the access patterns of this occurrence lie along lines with a particular desired direction $\vec{v} = (v_1, v_2)$, i.e. the directions become linearly dependent, we require that all index pairs (x, y) in one access pattern are on a line of the form $(x, y) = (c_1, c_2) + t \cdot (v_1, v_2)$, as illustrated in figure 3:

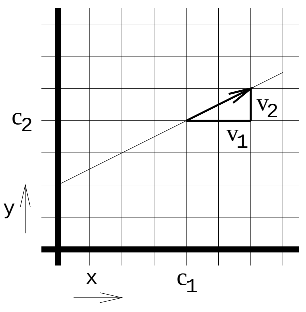

Figure 3: $(x, y) = (c_1, c_2) + t \cdot (v_1, v_2)$

An important observation is that for any point $(x, y) = (c_1 + t \cdot v_1, c_2 + t \cdot v_2)$ on this line, the equation $v_2 \cdot x - v_1 \cdot y = v_2 \cdot c_1 - v_1 \cdot c_2$ holds. Therefore, the expression $v_2 \cdot x - v_1 \cdot y$ remains constant along a line of the previous form. Consequently, one way to enforce a desired direction $\vec{v}$ is to require that the expression $v_2 \cdot x - v_1 \cdot y$ for $(x, y)^T = F_A(\vec{\mathtt{I}})$ is constant in one iteration of the *outermost* loop. Using the definition of section 2.1, x and y are defined as follows:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} m_{10} \\ m_{20} \end{pmatrix} + \begin{pmatrix} m_{11} & \dots & m_{1n} \\ m_{21} & \dots & m_{2n} \end{pmatrix} \cdot \vec{\mathtt{I}}$$

The constraint can be rewritten into the following form, where $k_i = v_2 \cdot m_{1i} - v_1 \cdot m_{2i}$ for $1 \leq i \leq n$:

$$(v_2 \cdot m_1 - v_1 \cdot m_2) + k_1 \cdot \mathtt{I}_1 + \dots + k_n \cdot \mathtt{I}_n = c'$$

Because only the variant part of this constraint is relevant, we simplify it as follows. In this expression, we define $\alpha_i = k_i / g$ for $g = \gcd(k_1, \dots, k_n)$, assuming that that $g \neq 0$:[1]

$$\alpha_1 \cdot \mathtt{I}_1 + \dots + \alpha_n \cdot \mathtt{I}_n = c \qquad (1)$$

A traversal of the original iteration space where in iteration $\mathtt{I}'_1 = c$, all iterations in the hyperplane $\alpha_1 \cdot \mathtt{I}_1 + \dots + \alpha_n \cdot \mathtt{I}_n = c$ are visited, is obtained by application of a linear transformation $\vec{\mathtt{I}}' = U \cdot \vec{\mathtt{I}}$, where the first row of U consists of $(\alpha_1, \dots, \alpha_n)$. In [21, 25], a completion method is presented that yields an $n \times n$ unimodular matrix of this form. In [11], we have extended this method to construct the inverse of this matrix efficiently by a simultaneous construction. This eliminates the need to explicitly compute the inverse afterwards. If no data dependences have to be accounted for, this completion method can be used to construct a loop transformation changing the direction of particular access patterns.

[1]For most conversion, $g \neq 0$ holds. Only if a single element or a one-dimensional part of the matrix is accessed that is parallel to the desired direction, the situation $g = 0$ can arise.

4.2 Double Loop Example

Suppose that regular diagonal-wise access patterns are desired in the following perfectly nested loop, in which access patterns with direction $(2, 4)$ occur:

```
DO I1 = 1, 6
  DO I2 = 1, 3
    ACC += A(I1+2*I2-2,4*I2-3)
  ENDDO
ENDDO
```

$$F_A(\vec{\mathtt{I}}) = \begin{pmatrix} -2 \\ -3 \end{pmatrix} + \begin{pmatrix} 1 & 2 \\ 0 & 4 \end{pmatrix} \cdot \vec{\mathtt{I}}$$

To achieve that the access patterns lie along lines with the desired direction $(1, 1)$, we require that expression $\mathtt{I}_1 - 2 \cdot \mathtt{I}_2$ is constant in one iteration of the outermost loop. If the dependence caused by the accumulation can be ignored, direct use of an automatically constructed matrix is possible:

$$U = \begin{pmatrix} 1 & -2 \\ 0 & 1 \end{pmatrix} \qquad U^{-1} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$

This reshaping, illustrated in figure 4, results in the following fragment in which access patterns with direction $(4, 4)$ occur:

```
DO I1' = -5, 4
  DO I2' = MAX(1,⌈(1-I1')/2⌉), MIN(3,⌊(6-I1')/2⌋)
    ACC += A(I1'+4*I2'-2,4*I2'-3)
  ENDDO
ENDDO
```

4.3 Validity of Reshaping

Application of a unimodular transformation U is valid if $\vec{d}' = U \cdot \vec{d} \succ \vec{0}$ holds for all dependence distances $\vec{d} \neq \vec{0}$ in the original loop [3, 5, 25]. This constraint is not always satisfied for a constructed transformation U. Fortunately, we can exploit the fact that several unimodular matrices can enforce traversal of hyperplanes that have the form (1), given in section 4.1. For double loops, the following proposition can be used:

Proposition 1 If there exists an $z_1 \in \{-1, +1\}$ such that $z_1 \cdot (\alpha_1, \alpha_2) \cdot \vec{d} \geq 0$ for all dependence distance vectors $\vec{d}$ in a double loop, then there exists an integer $z_2 \in \{-1, +1\}$ such that application of the

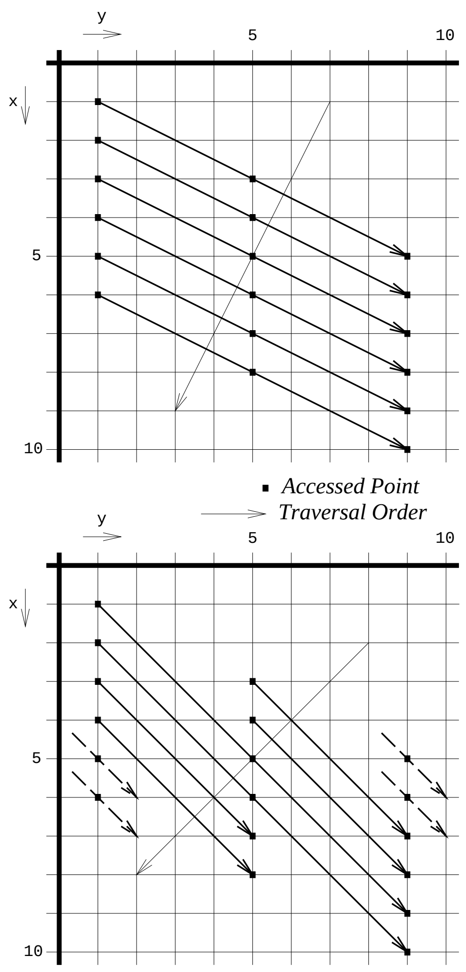

Figure 4: Conversion of $(2,4)$ into $(1,1)$

following transformation V, where U is a unimodular matrix with first row (α_1,α_2), is valid.

$$V=\begin{pmatrix} z_1 & 0 \\ 0 & z_2 \end{pmatrix}\cdot U \quad V^{-1}=U^{-1}\cdot\begin{pmatrix} z_1 & 0 \\ 0 & z_2 \end{pmatrix}$$

Proof Suppose there exists an $z_1\in\{-1,+1\}$ such that $z_1\cdot(\alpha_1,\alpha_2)\cdot\vec{d}\geq 0$ for all dependences distance vectors $\vec{d}$ in the original nest. Let $\vec{d}$ be an arbitrary dependence distance vector in the original nest. We distinguish two cases. If $z_1\cdot(\alpha_1,\alpha_2)\cdot\vec{d}>0$, then $V\cdot\vec{d}\succ\vec{0}$ and application of V is valid. If $z_1\cdot(\alpha_1,\alpha_2)\cdot\vec{d}=0$, then $(\alpha_1,\alpha_2)\cdot\vec{d}=0$, and, hence, $U\cdot\vec{d}=(0,t)^T$ for some $t\in\mathbf{Z}$. Therefore, we have $\vec{d}=U^{-1}\cdot(0,t)^T=t\cdot(-\alpha_2,\alpha_1)^T$. Since all dependences in the original nest are lexicographically positive we have, depending on the values of α_1 and α_2, *either* for all dependences $\vec{d}$ such that $(\alpha_1,\alpha_2)\cdot\vec{d}=0$, $\vec{d}$ is of the form $t\cdot(-\alpha_2,\alpha_1)^T$ for some $t\geq 0$, *or* for all dependences $\vec{d}$ such that $(\alpha_1,\alpha_2)\cdot\vec{d}=0$, $\vec{d}$ is of the form $t\cdot(-\alpha_2,\alpha_1)^T$ for some $t\leq 0$. Consequently, we can choose $z_2\in\{+1,-1\}$ such that for any $\vec{d}$ with $(\alpha_1,\alpha_2)\cdot\vec{d}=0$, we have $V\cdot\vec{d}=V\cdot U^{-1}(0,t)^T=(0,z_2\cdot t)^T\succ\vec{0}$. △

For example, making the direction through a matrix A and a desired direction $\vec{v}=(-1,1)$ linearly dependent by application of the following constructed transformation U would violate the original dependences with distance $\vec{d}=(1,-1)^T$ in the following fragment:

```
DO I1 = 2, 5
  DO I2 = 1, 5
    B(I1,I2) = A(I1,I2) * B(I1-1,I2+1)
  ENDDO
ENDDO
```

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

Negation of the second row in U, i.e. choosing $z_2=-1$, eliminates the dependence violation. The direction within each access pattern is reversed, as can be seen in the resulting code:

```
DO I1 = 3, 10
  DO I2 = MAX(2,I1'-5), MIN(5,I1'-1)
    B(I2',I1'-I2') = A(I2',I1'-I2') * B(I2'-1,I1'-I2'+1)
  ENDDO
ENDDO
```

The dependences within each access pattern are illustrated in figure 5.[2]

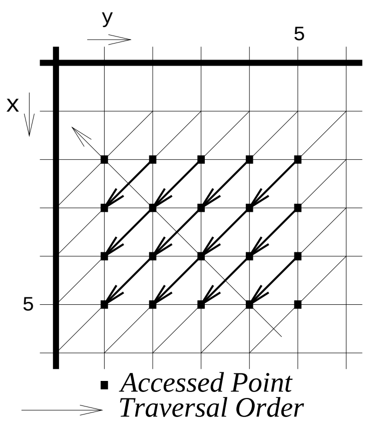

Figure 5: Dependences within one Access Pattern

The method fails if the first component of $U\cdot\vec{d}$ is positive for some dependences, but negative for others. This occurs, for example, in the conversion that

[2]Dependences are usually depicted in the iteration space. However, since x = I1 and y = I2 hold, a straightforward relation between accessed points and iterations hold.

makes the direction of the access patterns through A and $(1,1)$ linearly dependent for the following fragment with dependence distances $(0,1)^T$ and $(1,0)^T$:

```
DO I1 = 2, 5
  DO I2 = 2, 5
    B(I1,I2) = A(I1,I2) * B(I1,I2-1) * B(I1-1,I2)
  ENDDO
ENDDO
```

In this case, for the two dependence distances, we get the following equations:

$$\begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$
$$\begin{pmatrix} 1 & -1 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} +1 \\ 1 \end{pmatrix}$$

The reason for this problem is illustrated in figure 6. In diagonal-wise access patterns, all indices with $\mathtt{I}_1 - \mathtt{I}_2 = c$ are accessed before a next value of c is considered. However, the dependences imposes a cyclic traversal ordering on these access patterns.

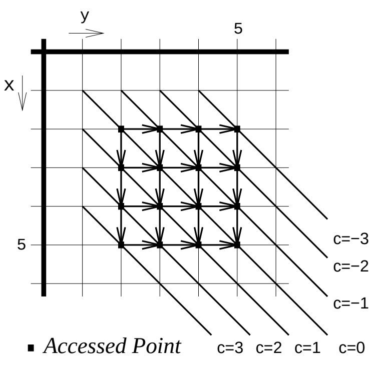

Figure 6: Cyclic Ordering on Access Patterns

4.4 Refinement of the Method

We have shown that the simple method of reshaping is successful for double loops, which follows from the fact that keeping expression (1) constant in one iteration of the *outermost* loop enforces a direction for the access patterns in the innermost loop, such that this direction and a desired direction $\vec{v}$ are linearly dependent. However, for general loops the basic method is too restrictive as can be seen in the following example. Enforcing row-wise access patterns requires that expression $\mathtt{I}_1 + 2 \cdot \mathtt{I}_3$ is constant in one iteration of the outermost loop, and results in the next conversion:

```
DO I1 = 1, 10
  DO I2 = 1, 10
    DO I3 = 1, 10
      ... A(I1+2*I3,I2) ...
    ENDDO
  ENDDO
ENDDO
```

$$\downarrow \quad U = \begin{pmatrix} 1 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \downarrow$$

```
DO I'1 = 3, 30
  DO I'2 = 1, 10
    DO I'3 = MAX(1,⌈(I'1-10)/2⌉), MIN(10,⌊(I'1-1)/2⌋)
      ... A(I'1,I'2) ...
    ENDDO
  ENDDO
ENDDO
```

In the resulting fragment, *all* iterations that reference elements in the same row are executed in one iteration of the outermost loop. Each set of iterations is within a plane with normal vector $(1,0,2)$, as illustrated in figure 7. Although this kind of transformations can be useful for sparse code generation, because it tends to move the overhead for accessing sparse rows to a higher level in the loop nest, it provides little flexibility. Especially, if we want to change the directions of the access pattern of *several* occurrences of implicitly sparse matrices in a particular loop, it is unlikely that identical hyperplanes must be traversed in the outermost loop.

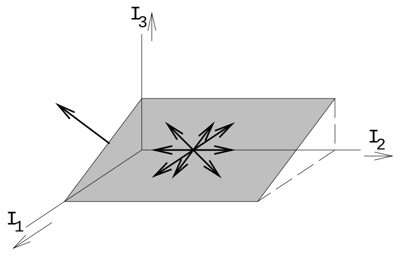

Figure 7: Plane $\mathtt{I}_1 + 2 \cdot \mathtt{I}_3 = c$

More flexibility is obtained if we observe that iterations along *any* line within the previous plane reference (possibly identical) elements in the same row. Consequently, any transformation that induces a loop in which all iterations along a line in this plane are performed in one iterations of the resulting innermost loop can be used to enforce row-wise (or possibly scalar-wise) access patterns. In general, the direction of an access patterns and a desired direction $\vec{v}$ are linearly dependent, if the iterations that are executed in one iteration of the resulting *innermost* loop are on a line within a hyperplane that is given by equation (1). Since a line can only be within such a hyperplane if the direction $\vec{u}$ of that line is perpendicular to the normal vector $(\alpha_1, \ldots, \alpha_n)$ of that plane, i.e. $(u_1, \ldots, u_n) \perp (\alpha_1, \ldots, \alpha_n)$, the following constraint is imposed on $\vec{u}$:

$$u_1 \cdot \alpha_1 + \ldots + u_n \cdot \alpha_n = 0 \qquad (2)$$

The general solution of diophantine equation (2) has the form $\vec{u} = T^{-1} \cdot (0, t_2, \ldots, t_n)^T$ for arbitrary $t_i \in \mathbf{Z}$ for a matrix T with $\alpha_1, \ldots, \alpha_n$ as first row [3]. Matrix T^{-1} can be obtained during the construction of T as described in [11]. For instance, the direction $\vec{u}$ of a line through the plane $\mathtt{I}_1 + 2 \cdot \mathtt{I}_3 = c$ of the previous example has the following general form:

$$\vec{u} = \begin{pmatrix} 1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ t_2 \\ t_3 \end{pmatrix} = \begin{pmatrix} -2 \cdot t_3 \\ t_2 \\ t_3 \end{pmatrix} \qquad (3)$$

Consequently, as observed at the end of section 3.3, any unimodular transformation U, for which the intersection of the hyperplanes defined by the first $n-1$ rows form a line with a direction satisfying constraint (2), can be used to enforce a particular direction on the access patterns. For instance, since direction $(0,1,0)^T$ is an instance of (3), and the intersection of planes $\mathtt{I}_1 = c_1$ and $\mathtt{I}_3 = c_2$ is precisely a line in this direction, a unimodular matrix where the first and second row correspond to these planes can also be used to enforce row-wise access patterns in the previous example. The following matrix, modeling a single loop interchange, satisfies that form. Note that the last column of U^{-1} is identical to this direction:

$$U = \left(\begin{array}{ccc} 1 & 0 & 0 \\ 0 & 0 & 1 \\ \hline 0 & 1 & 0 \end{array}\right) \quad U^{-1} = \left(\begin{array}{cc|c} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{array}\right)$$

Indeed, interchanging the $\mathtt{I}_2$- and $\mathtt{I}_3$-loop in the previous example is another, more obvious, way to obtain row-wise access patterns through matrix A. In some cases, a penalty must be paid for this increased flexibility. For example, because $(-2,0,1)^T$ is also an instance of (3), application of the following matrix is also allowed, because the intersection of planes $\mathtt{I}_2 = c_1$ and $\mathtt{I}_1 + 2 \cdot \mathtt{I}_3 = c_2$ is a line in this direction:

$$U = \left(\begin{array}{ccc} 0 & 1 & 0 \\ 1 & 0 & 2 \\ \hline 0 & 0 & 1 \end{array}\right) \quad U^{-1} = \left(\begin{array}{cc|c} 0 & 1 & -2 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{array}\right)$$

This effectively reverses the $\mathtt{I}'_1$- and $\mathtt{I}'_2$-loop in the code that results after application of the first transformation discussed in this section, and scalar-wise access patterns result. An illustration of the kind of sparse sparse code that can be generated automatically if row-wise storage is selected by the compiler is shown below (for non-entries an address $\perp$ for which `AVAL(`$\perp$`)=0.0` results, see [9] for details):

```
DO I = 1, 10
  DO J = 3, 30
   AD = LKP(AIND,ALOW(J),AHIGH(J),I)
   DO K = MAX(1,CEIL(J-10,2)), MAX(10,FLOOR(J-1,2))
     ... AVAL(AD) ...
   ENDDO
  ENDDO
ENDDO
```

Because the same element is referenced in a complete execution of the innermost non-controlling loop, lookup overhead in this row is amortized over several iterations. Therefore, scalar-wise access patterns are considered acceptable, which is reflected in the fact that the scalar-wise direction $(0,0)$ and any other direction are linearly dependent. However, solutions in which row-wise access patterns occur are preferred.

4.5 General Method for Reshaping

In this section, we present a five-step method for obtaining the solution of the following problem: given p subscript functions $F^i(\vec{\mathtt{I}})$ of several occurrences of implicitly sparse matrices in a perfectly nested loop of depth n, and desired directions $\vec{v}^{\,i}$, for $1 \leq i \leq p$, construct an $n \times n$ unimodular matrix U such that application of U to this loop nest is valid and achieves that the direction of the access patterns induced by each resulting subscript function and the desired direction $\vec{v}^{\,i}$ become linearly dependent.

4.5.1 Construction of Hyperplanes

In the first step, p hyperplanes of the form (1) in which expression $v^i_2 \cdot x - v^i_1 \cdot y$ where $(x,y)^T = F^i(\vec{\mathtt{I}})$, remains constant, are constructed for each subscript function and desired direction:

$$\left\{ \begin{array}{c} \alpha^1_1 \cdot \mathtt{I}_1 + \cdots + \alpha^1_n \cdot \mathtt{I}_n = c_1 \\ \vdots \\ \alpha^p_1 \cdot \mathtt{I}_1 + \cdots + \alpha^p_n \cdot \mathtt{I}_n = c_p \end{array} \right.$$

Occurrences for which this description of the hyperplane cannot be constructed because all coefficients are zero are ignored because any transformation will obtain an appropriate direction for the access patterns of such occurrences.

4.5.2 Construction of the Intersection

In this step, the general form for the direction $\vec{u}$ of a line that lies within the intersection of these hyperplanes is determined. Since the direction of this line must be perpendicular to all vectors $(\alpha_1^i, \ldots, \alpha_n^i)$, a general form of the solution of the following homogeneous system of linear diophantine equations must be obtained:

$$\left\{ \begin{array}{c} \alpha_1^1 \cdot u_1 + \ldots + \alpha_n^1 \cdot u_n = 0 \\ \vdots \\ \alpha_1^p \cdot u_1 + \ldots + \alpha_n^p \cdot u_n = 0 \end{array} \right. \quad (4)$$

Fortunately, finding a solution of this system has been well-studied in the context of data dependence analysis. In [3, 5] it is shown, that if system (4) is represented by $(u_1, \ldots, u_n) \cdot A = \vec{0}$, where the $\alpha_1^i, \ldots, \alpha_n^i$ constitutes the ith column of A, and a unimodular $n \times n$ matrix R represents the operations to reduce the $n \times p$ matrix A to echelon form, i.e. $R \cdot A = S$ for an $n \times p$ echelon matrix S, then all solutions are given by $\vec{u} = (t_1, \ldots, t_n) \cdot R$ for arbitrary $t_i \in \mathbf{Z}$ such that $(t_1, \ldots, t_n) \cdot S = \vec{0}$. An algorithm to reduce a matrix to echelon form, which means that nonzeros rows are ordered first and the nonzeros appear in staircase fashion, is given in [5]. If $r = \text{rank}(A)$, the first r rows of S are nonzero. Because a homogeneous system is considered, this implies that the general solution has the following form, for some $t_i \in \mathbf{Z}$:

$$\vec{u} = (\underbrace{0, \ldots, 0}_{r}, t_{n-r+1}, \ldots, t_n) \cdot R \quad (5)$$

Since the intersection of the p hyperplanes given in (4) is constructed, $\vec{u} = \vec{0}$ is the only solution if $\text{rank}(A) = n$. In that case, a compromise must be found. Possible solutions are to ignore certain occurrences or to adapt a number of desired directions so that a nonzero solution exists. Ideally, such decisions are made automatically, but user interaction can be used to simplify this process.

4.5.3 Selection of a Suitable Line

The result of the previous step is a general description of the direction $\vec{u}$ of lines lie within the hyperplanes constructed in the first step. In principle, an arbitrary nonzero instance of (5) can be taken. However, it is preferable to select an instance in which nonzero components appear at the position of controlling loops in the first, second, or any dimension of subscript functions for which column-, row- or diagonal-wise access patterns are desired, in order to avoid the situation discussed at the end of section 4.4. Moreover, $\vec{u}$ is chosen such that $\gcd(u_1, \ldots, u_n) = 1$.

4.5.4 Construction of a Transformation

In this step, a unimodular matrix must be constructed such that iterations in the original iteration space are executed along lines with direction $\vec{u}$. In earlier examples, it was shown that any unimodular transformation for which $\vec{u}$ constitutes the last column of its inverse can be used for this purpose. Either a new completion method can be used to obtain such a matrix, or the completion method of [11] can be adapted. First, a unimodular matrix T is constructed with $\vec{u}$ as first row. This is possible because the components of $\vec{u}$ are relatively prime. Subsequently, we may define matrices U^{-1} and U as:

$$U^{-1} = (P \cdot T)^T \text{ and } U = \left(T^{-1} \cdot P^T\right)^T$$

$$\text{where } P = \begin{pmatrix} 0 & 1 & & \\ \vdots & & \ddots & \\ 0 & & & 1 \\ 1 & 0 & \ldots & 0 \end{pmatrix}$$

4.5.5 Making the Transformation Valid

The unimodular transformation constructed in the previous step is used directly if there are no dependences in the loop nest. Otherwise, we can use the fact that the following matrix V is also unimodular and that the last column of V^{-1} is either $\vec{u}$ or $-\vec{u}$, for any unimodular $(n-1) \times (n-1)$ matrix Z and integer $z \in \{-1, +1\}$:

$$V = \begin{pmatrix} & & & 0 \\ & Z & & \vdots \\ & & & 0 \\ 0 & \ldots & 0 & z \end{pmatrix} \cdot U$$

$$V^{-1} = U^{-1} \cdot \begin{pmatrix} & & & 0 \\ & Z^{-1} & & \vdots \\ & & & 0 \\ 0 & \ldots & 0 & z \end{pmatrix}$$

The following construction method will be used to enforce that for all dependence vectors $\vec{d}$ that represent loop-carried data dependences in the original loop nest, $\vec{d'} = V \cdot \vec{d} \succ \vec{0}$ holds. Starting with $Z = I$ and $z = 1$, the following steps are applied for $k = 1, \ldots, d-1$. At each step k, we select an index i with $k \le i < n$, such that for all dependence vectors either all components $d'_i \le 0$ or all components $d'_i \ge 0$. Subsequently, rows i and k in Z are interchanged followed by a reversal of row i if the components were non-positive. Dependence vectors for which the ith component become strictly positive do not have to be considered in following steps. Finally, z is chosen such that the nth component of all remaining dependence vectors is non-negative. Because Z is constructed by a permutation and reversals, Z^{-1} is equal to Z^T.

If this construction breaks down, the reshaping method fails. This will occur, for instance, in situations that are similar to the one depicted in figure 6. However, in general, we have more flexibility in selection of iteration space traversal than with the simple method of section 4.1.

4.6 An Example of Reshaping

Consider the following fragment with occurrences of implicitly sparse matrices A, B and C:

```
DO I1 = 10, 15
  DO I2 = 1, 3
    DO I3 = 1, 50
      ... A(I1+3*I2+I3,I1+I3) ...
      ... B(2*I1+I3,2*I2)     ...
      ... C(I1-3*I2,I3)       ...
    ENDDO
  ENDDO
ENDDO
```

$$F_A(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & 3 & 1 \\ 1 & 0 & 1 \end{pmatrix} \cdot \vec{\mathtt{I}}$$

$$F_B(\vec{\mathtt{I}}) = \begin{pmatrix} 2 & 0 & 1 \\ 0 & 2 & 0 \end{pmatrix} \cdot \vec{\mathtt{I}}$$

$$F_C(\vec{\mathtt{I}}) = \begin{pmatrix} 1 & -3 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \vec{\mathtt{I}}$$

Suppose that row-wise storage is desired for all matrices, and that there are data dependences with distances $(0,0,1)^T$, $(0,1,0)^T$ and $(1,0,0)^T$. Reshaping the access patterns accordingly seems to be a non-trivial task at first sight. Because the desired direction for the access patterns of the three occurrences is $(0,1)$, the following planes result in the first step:

$$\begin{cases} \mathtt{I}_1 + 3\cdot\mathtt{I}_2 + \mathtt{I}_3 & = & c_1 \\ 2\cdot\mathtt{I}_1 + \mathtt{I}_3 & = & c_2 \\ \mathtt{I}_1 - 3\cdot\mathtt{I}_2 & = & c_3 \end{cases}$$

In step 2, the general form for the direction $\vec{u}$ of line within these planes is determined. Echelon reduction yields the following form for $R\cdot A = S$, where $\mathrm{rank}(S) = 2$:

$$\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 3 & 1 & -6 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 3 & 0 & -3 \\ 1 & 1 & 0 \end{pmatrix} = S$$

Direction $\vec{u}$ is described for arbitrary $t_3 \in \mathbf{Z}$ as $(0,0,t_3)\cdot R = (3\cdot t_3, t_3, -6\cdot t_3)$, indicating the direction of the line that is formed by the intersection of the previous planes. Vector $(3,1,-6)$ is an instance of this description for which the components are relatively prime, and can be selected at step 3. Moreover, because all components are nonzero, the associated transformation will induce real row-wise access patterns. In the fourth step, matrices U^{-1} and U are constructed:

$$U = \begin{pmatrix} -1 & 3 & 0 \\ 0 & 6 & 1 \\ 0 & 1 & 0 \end{pmatrix} \quad U^{-1} = \begin{pmatrix} -1 & 0 & 3 \\ 0 & 0 & 1 \\ 0 & 1 & -6 \end{pmatrix}$$

Computation of $U\cdot\vec{d}$ yields $(0,1,0)^T$, $(3,6,1)^T$ and $(-1,0,0)^T$ respectively. Since all second components are non-negative, rows 1 and 2 in U are interchanged. Since only the resulting distance $(0,-1,0)^T$ must be considered, reversal of the the second row suffices to make all resulting dependences lexicographically positive. This gives rise to the following transformations:

$$V = \begin{pmatrix} 0 & 6 & 1 \\ 1 & -3 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad V^{-1} = \begin{pmatrix} 0 & 1 & 3 \\ 0 & 0 & 1 \\ 1 & 0 & -6 \end{pmatrix}$$

Therefore, this transformation affects the fragment as illustrated below. Indeed, row-wise access patterns result:

```
DO I1' = 7, 68
  DO I2' = MAX(1,⌈(21-I1')/2⌉), MIN(12,⌊80-I1')/2⌋)
    DO I3' = MAX(1,⌈(10-I2')/3⌉,⌈(I1'-50)/6⌉),
             MIN(3,⌊(15-I2')/3⌋,⌊(I1'-1)/6⌋)
      ... A(I1'+I2',I1'+I2'-3*I3') ...
      ... B(I1'+2*I2',2*I3')       ...
      ... C(I2',I1'-6*I3')         ...
    ENDDO
  ENDDO
ENDDO
```

5 Data Structure Selection

The reshaping method presented in the previous section enables us to take a systematic approach to data structure selection, as illustrated below with an example.

5.1 Sparse Matrix Multiplication

Consider, for example, the following formulation of the operation $C \leftarrow C + A\cdot B$, where A, B and C are implicitly sparse matrices:

```
DO I_1 = 1, 10
  DO I_2 = 1, 10
    DO I_3 = 1, 10
      C(I_1,I_2) += A(I_1,I_3) * B(I_3,I_2)
    ENDDO
  ENDDO
ENDDO
```

$$F_A(\vec{I}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \vec{I}$$

$$F_B(\vec{I}) = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \vec{I}$$

$$F_C(\vec{I}) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \cdot \vec{I}$$

The reshaping method of section 4.5 enables us to explore all possible data structures. For any combination of e.g. row-, column- and the regular diagonal-wise storage of the three implicitly sparse matrices, the unimodular transformation that changes the direction of the access patterns accordingly can be determined. The result of this approach is shown in figure 8:

For example, in case row-, column- and row-wise access patterns are desired for matrices A, B and C respectively, $\vec{u} = (0,0,1)$ results which, not surprisingly, gives rise to the unimodular transformation $U = I$. Consequently, a tile is placed on the row-column-row combination. Note that, for example, all combination with row- and column-wise access patterns for A and B respectively are possible for $\vec{u} = (0,0,1)$ because the resulting scalar-wise access patterns for C matches all three directions. As another example, because reshaping for diagonal-wise access pattern for B and C, while keeping the access patterns of A row-wise, results in the following echelon reduction $R \cdot A = S$, where $\text{rank}(S) = 3$, the method fails and no tile appears at the corresponding position:

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

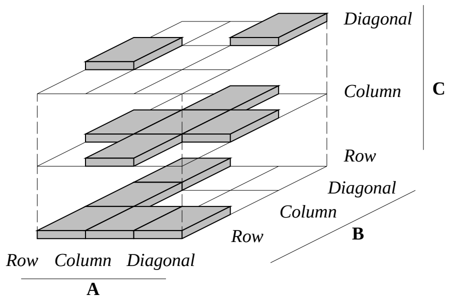

Figure 8: Matrix Multiplication

Discussions of the implementations of some of these variants can be found in e.g. [9, 18, 20, 23].

6 Future Research

In this paper we have presented a reshaping method which, for different occurrences in a perfectly nested loop, constructs a unimodular transformation that changes a direction of the access patterns of these occurrences into a desired direction. This reshaping method is necessary for the selection of compact data structures for implicitly sparse matrices that enables the generation of efficient sparse codes. For a simple fragment, the method can be used for an exhaustive search of the possible data structures for the different occurrences. However, since such an approach would induce an unacceptable increase of compile-time for real programs, a strategy to control the reshaping in combination with data structure selection must be developed.

Presently, we are studying the possibility to use the theory of this paper, that has been studied in the context of automatic conversion of dense programs into sparse code, to solve other kinds of problems. For instance, since for vector computers it is often desirable to have stride-1 accesses in the innermost loop, the method can be extended so that all accesses of several multi-dimensional arrays are along the first dimension.

The ideas of this paper have been incorporated in a prototype restructuring compiler MT1 [8]. Apart from a reshaping method, the compiler must also be able to isolate particular regions of the index set of the matrix in order to exploit characteristics of the nonzero structure of the matrix. In a forthcoming paper [12], we will present an execution set partitioning transformation that can be used for this purpose.

References

[1] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, Volume 9:491–542, 1987.

[2] Corinne Ancourt and Francois Irigoin. Scanning polyhedra with do loops. In *Proceedings of Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, 1989.

[3] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, 1988.

[4] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of Third Workshop on Languages and Compilers for Parallel Computing*, 1990.

[5] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, Boston, 1993.

[6] Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. In *Proceedings of Supercomputing 93*, pages 430–439, 1993.

[7] Aart J.C. Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the International Conference on Supercomputing*, pages 416–424, 1993.

[8] Aart J.C. Bik and Harry A.G. Wijshoff. MT1: A prototype restructuring compiler. Technical Report no. 93-32, Dept. of Computer Science, Leiden University, 1993.

[9] Aart J.C. Bik and Harry A.G. Wijshoff. On automatic data structure selection and code generation for sparse computations. In *Proceedings of the Sixth International Workshop on Languages and Compilers for Parallel Computing*, pages 57–75, 1993. Lecture Notes in Computer Science, No. 768.

[10] Aart J.C. Bik and Harry A.G. Wijshoff. Nonzero structure analysis. In *Proceedings of the International Conference on Supercomputing*, 1994. To appear.

[11] Aart J.C. Bik and Harry A.G. Wijshoff. On a completion method for unimodular matrices. Technical Report no. 94-14, Dept. of Computer Science, Leiden University, 1994.

[12] Aart J.C. Bik and Harry A.G. Wijshoff. On strategies for generating sparse codes. Technical Report In Progress, Dept. of Computer Science, Leiden University, 1994.

[13] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory*, Volume 14:288–297, 1973.

[14] David S. Dodson, Roger G. Grimes, and John G. Lewis. Algorithm 692: Model implementation and test package for the sparse linear algebra subprograms. *ACM Transactions on Mathematical Software*, Volume 17:264–272, 1991.

[15] Michael L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, Volume 16:157–171, 1990.

[16] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices.* Oxford Science Publications, 1990.

[17] I.S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, Volume 15:1–14, 1989.

[18] Fred G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, Volume 4:250–269, 1978.

[19] Wei Li and Keshav Pingali. A singular loop transformation framework based on non-singular matrices. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, 1992.

[20] John Michael McNamee. Algorithm 408: A sparse matrix package. *Communications of the ACM*, pages 265–273, 1971.

[21] Morris Newman. *Integral Matrices.* Academic Press, New York, 1972. Pure and Applied Mathematics, Volume 45.

[22] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, pages 1184–1201, 1986.

[23] Sergio Pissanetsky. *Sparse Matrix Technology.* Academic Press, London, 1984.

[24] C.D. Polychronoupolos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, 1988.

[25] Michael E. Wolf and Monica S. Lam. A data locality optimizing algorithm. In *Proceedings ACM SIGPLAN 91 Conference on Programming Languages Design and Implementation*, pages 30–44, 1991.

[26] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Algorithms*, pages 452–471, 1991.

[27] Michael J. Wolfe. Loop skewing: The wavefront method revisited. *International Journal of Parallel Programming*, Volume 15:279–293, 1986.

[28] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London, 1989.

[29] H. Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.

[30] Zahari Zlatev. *Computational Methods for General Sparse Matrices.* Kluwer Academic Publishers, 1991.