Sparse code generation for imperfectly nested loops with dependences

Vladimir Kotlyar   Keshav Pingali
Department of Computer Science
Cornell University, Ithaca, NY 14853
{*vladimir,pingali*}*@cs.cornell.edu*

Abstract

Standard restructuring compiler tools are based on polyhedral algebra and cannot be used to analyze or restructure sparse matrix codes. We have recently shown that tools based on relational algebra can be used to generate an efficient sparse matrix program from the corresponding dense matrix program and a specification of the sparse matrix format. This work was restricted to DO-ALL loops and loops with reductions. In this paper, we extend this approach to loops with dependences. Although our results are restricted to Compressed Hyperplane Storage formats, they apply to both perfectly nested loops and imperfectly nested loops.

1 INTRODUCTION

Although sparse matrix computations are ubiquitous in computational science, research in restructuring compilers has focused almost exclusively on *dense* matrix programs. This is because the tools used in restructuring compilers are based on the algebra of polyhedra, and can be used only when array subscripts are affine functions of loop index variables. Sparse matrices are represented using compressed formats to avoid storing zeroes, so array subscripts in sparse matrix programs are often complicated expressions involving indirection arrays. Therefore, tools based on polyhedral algebra cannot be used to analyze or to restructure sparse matrix programs.

One possibility is to express the algorithm as a dense matrix program, but declare to the compiler that certain matrices are actually sparse. This approach restores the compiler's ability to analyze the program, but it makes the compiler responsible for choosing the format used to represent the sparse matrices in the program, and for generating code in which computations involving zeros are eliminated. This idea was explored by Bik and Wijshoff in a series of papers [1–6]. However, the task of choosing a good representation for sparse matrices is a somewhat delicate one, and doing it right requires that careful attention be paid to the data flow of the algorithm, the structure of the non-zeros in sparse matrices, and properties of the target architecture. Compressed Row/Column Storage is simple and is therefore used very often in practice. However, the sparse matrices that arise in PDE solvers often have many rows with the same non-zero structure, and are best represented as a collection of small dense matrices in which each dense matrix arises from gathering data from rows with the same non-zero structure [7]. Formats like the jagged-diagonal format are used when the target architecture is a vector machine [11].

	No dependences	Dependences
No fill	MVM, MMM with dense or preallocated left-hand side	Solution of triangular systems
Fill	MVM, MMM with sparse left-hand side	Matrix factorizations

Table 1: Classification of sparse matrix codes

We have chosen to make the programmer responsible for specifying sparse formats. Therefore, our compiler must solve the problem of generating efficient sparse code, given a loop nest containing dense matrix computations and a specification of sparse matrix formats. In an earlier paper, we used techniques from relational algebra to solve this problem for the special case of DO-ANY loop nests [9]. As Table 1 shows, many common programs such as matrix-vector product and matrix-matrix product are included in this class, but other programs of great practical importance such as triangular solve and Cholesky factorization are not in this class.

In this paper, we extend our techniques to the problem of generating code for loop nests with dependences, restricting ourselves to the special case of sparse formats which represent Compressed Hyperplane Storage (of which Compressed Row/Column Storage are special cases). The rest of the paper is organized as follows. Section 2 reviews our relational approach to sparse matrix program compilation. Section 3 describes how these methods can be extended to handle perfectly nested loops with dependences. Section 4 describes how transformations for imperfectly nested loops are generated. Section 5 compares our approach with the access reshaping and guard encapsulation techniques of Bik and Wijshoff. Section 6 describes the future directions of our work.

2 RELATIONAL APPROACH TO SPARSE MATRIX CODE COMPILATION

2.1 AN EXAMPLE

Consider the loop which computes the inner product of two vectors $\mathbf{X}$ and $\mathbf{Y}$:

DO $i = 1, n$
$\quad dot = dot + X(i) * Y(i)$

We can view the arrays X and Y, and the iteration set of the loop as *relations* (or tables):

$$
\begin{aligned}
R_X &= \{\langle i_x, v_x \rangle \mid X(i) = v_x\} && (1)\\
R_Y &= \{\langle i_y, v_y \rangle \mid Y(i) = v_y\} && (2)\\
R_I &= \{\langle i \rangle \mid 1 \le i \le n\} && (3)
\end{aligned}
$$

The loop enumerates over the tuples $\langle i, i_x, v_x, i_y, v_y \rangle$ which satisfy the following query [1] :

$$Q = \sigma_{(i=i_x \wedge i=i_y)}(R_I(i) \times R_X(i_x, v_x) \times R_Y(i_y, v_y)) \quad (4)$$

Conceptually, the relations R_X and R_Y store both zero and non-zero values of the arrays X and Y. When the arrays are sparse, they might actually be stored using some compressed scheme. In this case, only non-zeroes are stored explicitly and zeros are implicit. Following the practice in sparse matrix literature, we assume that only the elements that are not explicitly stored are zero. We use the predicates $NZ(R_X(i))$ and $NZ(R_Y(i))$ to test if an element is stored and should be assumed to be non-zero. It is important to notice that the NZ predicate always evaluates to *true* for a dense vector, even though some elements might be numerically zero.

If the vectors are sparse, then only those iterations need to be executed for which $X(i) \neq 0 \wedge Y(i) \neq 0$. In terms of our relational query formulation we wish to select only those i's for which $NZ(R_X(i)) \wedge NZ(R_Y(i))$ is true. If we let $\mathcal{P} \stackrel{\text{def}}{=} NZ(R_X(i)) \wedge NZ(R_Y(i))$, then the query for the sparse loop nest becomes:

$$
\begin{aligned}
&Q_{\text{sparse}} = \sigma_{\mathcal{P}} Q = \\
&= \sigma_{\mathcal{P}} \sigma_{(i=i_x \wedge i=i_y)}(R_I(i) \times R_X(i_x, v_x) \times R_Y(i_y, v_y)) \quad (5)
\end{aligned}
$$

To evaluate this query efficiently, we push the $\sigma_{i=i_x \wedge i=i_y}$ selection into the cross products to obtain equi-joins [2] :

$$
\begin{aligned}
Q_{\text{sparse}} &= \sigma_{\mathcal{P}}(R_I(i) \bowtie R_X(i, v_x) \bowtie R_Y(i, v_y)) \\
&= \sigma_{\mathcal{P}}(R_X(i, v_x) \bowtie_i R_Y(i, v_y)) \quad (6)
\end{aligned}
$$

Once equi-joins in the query are exposed, we can apply one of three algorithms for computing equi-joins (see [12], for example):

- *Enumerate-Search:* Walk over the tuples of one relation and search for the common attribute in the other.
- *Merge-Join:* If both relations are sorted on the join attribute, then walk over both of them "in step".
- *Hash-Join:* Scatter the tuples of one of the relations into a hash table (which can be a dense vector), then walk over the other and probe the hash table with the join attribute.

The best choice for the join algorithm depends on the *access methods* available for the relations and their properties: for example, Merge-Join is competitive only if the join index in both relations can be enumerated efficiently in sorted order. These algorithms can be combined to compute joins of more than two relations. They can also be generalized to enumerate over the zeros as well as non-zeroes. This is necessary, for example, in the case of computing a sum of two vectors: the equi-join is the same as in (6), but the sparsity predicate $\mathcal{P}$ is different: $\mathcal{P} \stackrel{\text{def}}{=} NZ(R_X(i)) \vee NZ(R_Y(i))$.

2.2 GENERAL APPROACH

In the inner product example, it was easy to expose joins in the query expression. More generally, we perform the following steps to expose joins and generate code; the interested reader can find the details in [9].

1. Dense loop nests are converted to relational queries as in (4). Array access functions generate appropriate selection predicates.
2. A sparsity predicate is computed as described in [1, 5] and is converted into a selection in terms of the $NZ(\dots)$ predicates as in (5).
3. Equi-joins are discovered by "pushing" selections with equality predicates "through" cross-products.
4. In general, we might have several equi-joins, and these are ordered (nested) to minimize expensive searches in compressed data structures.
5. Implementation algorithms are selected for the equi-joins and are specialized to enumerate the correct combination of zeros and non-zeros as given by the predicate $\mathcal{P}$.

We discuss Steps 3 and 4 in more detail since they have to modified to accommodate loops with dependences. Suppose we have a perfectly nested loop with a single statement S and the sparsity predicate $\mathcal{P}$:

DO $\mathbf{i} \in \mathcal{B}$
$\quad$ IF $\mathcal{P}$ THEN
$\quad\quad S : A_1(\mathbf{F}_1 \mathbf{i} + \mathbf{f}_1) = \dots A_k(\mathbf{F}_k \mathbf{i} + \mathbf{f}_k) \dots$

where $\mathbf{i}$ is the vector of loop indices and $\mathcal{B}$ represents the loop bounds. We make the usual assumption that the loop bounds are polyhedral, and that the arrays A_k, $k = 1 \dots N$, are addressed using affine access functions.

To generate the relational query for computing the set of sparse loop iterations, it is useful to define the following vectors and matrices:

$$\mathbf{H} = \begin{pmatrix} \mathbf{I} \\ \mathbf{F}_1 \\ \vdots \\ \mathbf{F}_N \end{pmatrix} \quad \mathbf{a} = \begin{pmatrix} \mathbf{i} \\ \mathbf{a}_1 \\ \vdots \\ \mathbf{a}_N \end{pmatrix} \quad \mathbf{f} = \begin{pmatrix} \mathbf{0} \\ \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_N \end{pmatrix} \quad (7)$$

[1] σ is the relational algebra *selection* operator. The notation $R(a, b)$ names the fields in the relation R for use in the selection predicates.

[2] We can eliminate R_I from the query since it does not restrict the result any more than R_X and R_Y do.

$$\begin{pmatrix} c_1 & 0 & 0 & & 0 \\ L_2' & c_2 & 0 & \dots & 0 \\ \multicolumn{2}{c}{L_3'} & c_3 & & 0 \\ \vdots & & & \ddots & \\ \multicolumn{4}{c}{L_r'} & c_r \end{pmatrix}$$

Figure 1: Permuted Column Echelon Form of Data Access Matrix

Following [10], the matrix $\mathbf{H}$ is called a *data access matrix*. Notice that $\mathbf{H}$ always has full column rank. It is easy to see that the following *data access equation* holds:

$$\mathbf{a} = \mathbf{f} + \mathbf{H}\mathbf{i} \tag{8}$$

As described in [9], we view the arrays A_k as relations with the following attributes:

- $\mathbf{a}_k$, which stands for the vector of array indices
- v_k, which is the value of $A_k(\mathbf{a}_k)$

Given all of the above, the sparse loop nest can be thought of as an enumeration of the tuples that satisfy the following relational query (R_I is the iteration space relation):

$$\sigma_{\mathcal{P}}\sigma_{(\mathbf{a}=\mathbf{f}+\mathbf{H}\mathbf{i})}\left(R_I(\mathbf{i}) \times \ldots \times A_k(\mathbf{a}_k, v_k) \times \ldots\right) \tag{9}$$

To evaluate this query efficiently, we discover equi-joins between different fields (attributes) $\mathbf{a}_k$, as mentioned before. More precisely, we discover *affine joins* between pairs of attributes that are related by affine equalities for all values of $\mathbf{i}$. In [9] this is done by computing the *permuted echelon form* of the data access matrix $\mathbf{H}$:

$$\mathbf{H}' = \mathbf{P}\mathbf{H}\mathbf{U} \tag{10}$$

where $\mathbf{P}$ is the matrix of row permutations and $\mathbf{U}$ is the unimodular matrix of "zeroing" transformations. The structure of the matrix $\mathbf{H}'$ is shown in Figure 1 (r is the rank of $\mathbf{H}$). Each column vector $\mathbf{c}_k$ is *all* non-zero.

The point of computing this echelon form is the following. If we let $\mathbf{b} = \mathbf{P}(\mathbf{a} - \mathbf{f})$ and $\mathbf{j} = \mathbf{U}^{-1}\mathbf{i}$, the data access equation (8) can be rewritten as:

$$\mathbf{b} = \mathbf{H}'\mathbf{j} \tag{11}$$

We can now partition vector $\mathbf{b}$ into r blocks according to the partitioning of $\mathbf{H}'$ in Figure 1 and obtain the following equation for each block $m = 1, \ldots, r$:

$$\mathbf{b}_m = \mathbf{L}_m \cdot \mathbf{j}_{1..(m-1)} + \mathbf{c}_m \cdot \mathbf{j}_m \tag{12}$$

In the generated code, $\mathbf{j}_m$ is the mth loop variable. Since the values $\mathbf{j}_{1..(m-1)}$ are enumerated by the outer loops, the affine joins for this loop are defined by the following equations for $m = 1, \ldots, r$:

$$\mathbf{b}_m = \text{invariant} + \mathbf{c}_m * \mathbf{j}_m \tag{13}$$

If we let $\mathbf{a}' = \mathbf{P}\mathbf{a}$ and $\mathbf{f}' = \mathbf{P}\mathbf{f}$, we can rewrite (13) as:

$$\begin{aligned} \mathbf{a}'_m &= \mathbf{f}' + \text{invariant} + \mathbf{c}_m * \mathbf{j}_m = \\ &= \text{another\_invariant} + \mathbf{c}_m * \mathbf{j}_m \end{aligned} \tag{14}$$

Therefore, at each level, the permuted attributes are related by simple affine equations (through $\mathbf{j}_m$). That is, each loop variable $\mathbf{j}_m$ of the new loop nest enumerates over the results of an affine join. Notice that the matrix $\mathbf{U}^{-1}$ from (10) gives us the loop transformation from the original loop nest to loops that enumerate joined attributes.

How is $\mathbf{P}$ in (10) being chosen? This permutation gives us the nesting order in which different fields in the relations (arrays) are being enumerated. Since most sparse matrix formats have a hierarchical structure, $\mathbf{P}$ must be chosen carefully to avoid expensive searches.

We illustrate all of these ideas on an example. Consider the case of the product of a sparse matrix $\mathbf{A}$ and a sparse vector $\mathbf{X}$, which is being stored into a dense vector $\mathbf{Y}$. Assume that the matrix is compressed along the diagonals into a sparse array A (i.e., it is stored in the indices $s = i - j$ and $t = j$, where s indexes the diagonals and t runs within each diagonal). The array A itself is stored in CRS [3]. To exploit this hierarchical structure, we must enumerate s in the outer loop and t in the inner loop. The loop nest (with sparsity predicate) is:

```
DO i = 1, n
   DO j = 1, n
      IF (A(i - j, j) ≠ 0 ∧ X(j) ≠ 0)THEN
         Y(i) = Y(i) + A(i - j, j) * X(j)
```

The relations is this query are: $R_I(i,j)$, $R_A(s,t,v_a)$, $R_X(j_x, v_x)$ and $R_Y(i_y, v_y)$. If we define the following predicates :

$$\mathcal{P} \stackrel{\text{def}}{=} NZ(R_A(i-j,j)) \wedge NZ(R_X(j)) \tag{15}$$

$$\mathcal{A} \stackrel{\text{def}}{=} s = i - j \wedge t = j \wedge i = i_y \wedge j = j_x \tag{16}$$

then the query is:

$$\begin{aligned} Q = \sigma_{\mathcal{P}}\sigma_{\mathcal{A}}\Big(&R_I(i,j) \times R_A(s,t,v_a) \times \\ &\times R_X(j_x, v_x) \times R_Y(i_y, v_y)\Big) \end{aligned} \tag{17}$$

The data access equation (with s permuted into the outermost position) is:

$$\begin{pmatrix} s \\ i \\ j \\ t \\ i_y \\ j_x \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{18}$$

To get this system into echelon form we need to add the first column of the data access matrix to the second. This can

[3] Such combination of an index transformation and CRS defines an instance of CHS.

```
DO s ∈ R_A
    DO ⟨t, v_a, j_x, v_x⟩ ∈ Merge(R_X, R_A(s,*))
        v_y = search R_Y for (s + t)
        v_y = v_y + v_a * v_x
```

Figure 2: Sparse matrix vector product

be done using the column transformation matrix $\mathbf{U}$ and the loop transformation matrix $\mathbf{U}^{-1}$, shown below:

$$\mathbf{U} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \qquad \mathbf{U}^{-1} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \tag{19}$$

The data access equation in echelon form is:

$$\begin{pmatrix} s \\ i \\ j \\ t \\ i_y \\ j_x \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \qquad \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{20}$$

This reveals two nested joins. The outer one simply enumerates across the diagonals s of A. For a fixed value $s = s_0$, the inner loop joins the diagonals with the vectors X and Y:

$$i_y - s_0 = t = j_x \tag{21}$$

Notice that each inner join has a different starting value for i_y. Suppose that the vector X is sparse and the vector Y is dense. If the indices of X and the indices t within the diagonals can be enumerated in sorted order, we obtain the code shown in Figure 2. Each diagonal of A is joined with X using merge-join and the result is joined with Y using enumerate-search join.

To summarize, here are the two key matrices and their roles in the code generation process:

- The permuted echelon form $\mathbf{H}' = \mathbf{PHU}$ reveals affine joins; these joins can be implemented in several ways.
- The matrix $\mathbf{U}^{-1}$ represents the transformation from the original loop nest to one that enumerates the joined attributes.

3 PERFECTLY NESTED LOOPS WITH DEPENDENCES

In Section 2, we considered only DO-ANY loops, so we did not worry about the legality of the transformation $\mathbf{U}^{-1}$. We now describe how a legal transformation can be generated, if at all possible, when the loop nest has dependences.

3.1 A MOTIVATING EXAMPLE

The loop nest in Figure 3 computes the solution to a sparse unit lower triangular system $\mathbf{Lx} = \mathbf{b}$ (the solution is accumulated in $\mathbf{b}$). The dense loop nest has two dependences, which the generated sparse code has to satisfy:

- A flow dependence from the write into $b(i)$ to the read from $b(j)$. It can be expressed as a distance/direction vector $\mathbf{d} = \binom{+}{+}$

```
DO i = 1, n
    DO j = 1, i - 1
        IF (L(i,j) ≠ 0) THEN
            b(i) = b(i) - L(i,j)b(j)
```

Figure 3: A generic sparse unit lower triangular solver

```
DO j = 1, n
    w_b = search R_b for j
    DO ⟨i, v_L⟩ ∈ R_L(*, j), i > j
        v_b = search R_b for i
        v_b = v_b - v_L · w_b
```

Figure 4: Sparse column-oriented unit lower triangular solver

- An output dependence between successive updates to $b(i)$. We will ignore this dependence, assuming that the updates are commutative and associative, as is standard.

The data access equation for this loop nest is:

$$\mathbf{a} = \begin{pmatrix} i \\ j \\ i_L \\ j_L \\ i_B \\ j_B \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{22}$$

Suppose that $\mathbf{L}$ is compressed along the columns. Then the desired permuted echelon form of the data access equation is:

$$\begin{pmatrix} j_L \\ j_B \\ j \\ i_L \\ i_B \\ i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \qquad \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} i \\ j \end{pmatrix} \tag{23}$$

The column transformation matrix and its inverse are:

$$\mathbf{U} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \mathbf{U}^{-1} \tag{24}$$

If we assume that the vector $\mathbf{b}$ is dense, then both joins can be performed using the Enumerate-Search strategy, because searches in the dense relations R_I and R_B are simple look-ups which are inexpensive. This gives us the code in Figure 4.

We must now verify that the transformed loop does not violate dependences. The transformation $\mathbf{U}^{-1}$ is legal only if it preserves dependences in the original code. This can be expressed as follows:

$$\mathbf{U}^{-1}\mathbf{d} \succ 0 \tag{25}$$

where $\succ$ corresponds to lexicographic order. In our case:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} + \\ + \end{pmatrix} = \begin{pmatrix} + \\ + \end{pmatrix} \succ 0 \tag{26}$$

Therefore, the transformation is legal. Notice that the following matrix also gives us an echelon form:

$$\mathbf{U} = \mathbf{U}^{-1} = \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \tag{27}$$

but the resulting loop nest is illegal because:

$$\begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} + \\ + \end{pmatrix} = \begin{pmatrix} - \\ - \end{pmatrix} \not\succ 0 \tag{28}$$

Therefore, we were lucky to obtain the legal transformation matrix $\mathbf{U}^{-1}$ from (24). The problem therefore is to invent an algorithm that finds an echelon form with a legal transformation matrix, if one exists.

3.2 FINDING A LEGAL TRANSFORMATION

Formally, the problem is as follows.

- Given the data access matrix $\mathbf{H}$ and matrix of dependence distance/direction vectors $\mathbf{D}$,
- find a row permutation matrix $\mathbf{P}$ and a column transformation matrix $\mathbf{U}$ such that:
  - $\mathbf{H}' = \mathbf{PHU}$ is in echelon form
  - $\mathbf{U}^{-1}\mathbf{D} \succ 0$ – i.e. the resulting loop transformation is legal
  - $\mathbf{P}$ satisfies some profitability conditions. We discuss this point later in Section 3.3

We start by finding *some* transformation matrix $\mathbf{U}$ such that $\mathbf{H}' = \mathbf{PHU}$ is in echelon form. If this matrix is not legal, we will update it by using the following fact:

Theorem 1 *Let* $\mathbf{H}$ *be a matrix with* M *rows and* N *columns. Assume* $\mathbf{H}$ *has full column rank. Let* $\mathbf{H}' = \mathbf{PHU}'$ *be a permuted echelon form of* $\mathbf{H}$*. Let* $\mathbf{Q}$ *be an* N*-by-*N *invertible matrix. Let* $\mathbf{H}'' = \mathbf{H}'\mathbf{Q}$*. Then* $\mathbf{H}''$ *is a permuted echelon form of* $\mathbf{H}$ *with the same column permutation matrix* $\mathbf{P}$ *if and only if* $\mathbf{Q}$ *is a lower triangular matrix.*

For lack of space, we omit the proof. This theorem suggests the following algorithm:

1. Find an echelon form $\mathbf{H}' = \mathbf{PHU}$ of $\mathbf{H}$ as described in [9].

2. Find a non-singular lower-triangular matrix $\mathbf{M}$ such that

$$\mathbf{MU}^{-1}\mathbf{D} \succeq 0 \tag{29}$$

   The method for finding $\mathbf{M}$ is a simple modification of the completion procedure of [10].

3. The new transformation is $\mathbf{V} = \mathbf{UM}^{-1}$. The fact that $\mathbf{H}'' = \mathbf{PHV}$ is in echelon form follows from Theorem 1 and the fact that the inverse of a lower triangular matrix is also lower triangular. Theorem 1 tells us that this way of finding a new (legal) transformation $\mathbf{V}$ is *complete* – if there is a legal transformation, it is a product of $\mathbf{U}^{-1}$ and a lower triangular matrix.

4. The remaining question is the following: What do we do if we can not find a legal transformation of $\mathbf{H}$ into its echelon form? This problem is addressed below.

If we cannot transform all of $\mathbf{H}$ into echelon form legally, we use a partial transformation that tries to get as many rows of $\mathbf{H}$ into echelon form (legally) as possible. Intuitively, the rows of the transformed $\mathbf{H}$ that are in echelon form correspond to array dimensions which we can enumerate using joins, while the rest of the rows correspond to array dimensions which have to be searched. More formally, let

$$\mathbf{PHU} = \mathbf{H}' = \begin{pmatrix} \mathbf{H}_1' & 0 \\ \mathbf{H}_2' & \mathbf{H}_3' \end{pmatrix} \tag{30}$$

be a "partial" echelon form. Here $\mathbf{H}_1'$ has full column rank and is in echelon form and $\mathbf{H}_2$ and $\mathbf{H}_3$ are some matrices. Let r be the number of rows in $\mathbf{H}_1'$. We will say that $\mathbf{H}'$ is an *r-partial echelon form* of $\mathbf{H}$.

If the transformation $\mathbf{U}$ is not legal, we try to augment it to a legal one by multiplying it on the right by a suitable non-singular matrix. It follows from Theorem 1 that if we multiply $\mathbf{H}'$ on the right by a non-singular matrix $\mathbf{B}$ which has structure:

$$\mathbf{B} = \begin{pmatrix} \mathbf{L} & 0 \\ \mathbf{M}_1 & \mathbf{M}_2 \end{pmatrix} \tag{31}$$

with $\mathbf{L}$ being lower triangular and the blocking being the same as in (30), then the result $\mathbf{H}'' = \mathbf{H}'\mathbf{B}$ is also a *partial* echelon form of $\mathbf{H}$ with the same structure as $\mathbf{H}'$.

It is not hard to see that for $\mathbf{B}$ from (31), the inverse $\mathbf{B}^{-1}$ has the same structure. This gives us the following algorithm for finding a legal r-partial echelon form of $\mathbf{H}$ for a given r:

1. Find *some* r-partial echelon form (30) by doing row permutations and column operations as in [9].

2. Find a non-singular matrix $\mathbf{B}$ with the structure as in (31) such that:

$$\mathbf{BU}^{-1}\mathbf{D} \succeq 0$$

   The method for finding $\mathbf{B}$ to satisfy this equation is similar to the completion procedure of [10]. If such $\mathbf{B}$ exists, then $\mathbf{V} = \mathbf{UB}^{-1}$ gives us a legal transformation.

3.3 THE SEARCH PROBLEM

In [9], the problem is to find the best permutation $\mathbf{P}$, which gives the ordering between different joins. Our case is complicated by the fact that different $\mathbf{P}$'s can lead to legal r-partial echelon forms of different sizes (r's). Therefore the best permutation $\mathbf{P}$ from the point of view of [9] might not lead to the fullest r-partial echelon form.

In general, we might have to explore the space of all permutations $\mathbf{P}$ and all r-partial echelon forms (for each permutation). The size of the space is $M * M!$, where M is the number of attributes, and for each point in the space we need to find a (partial) echelon form and perform the completion procedure. Therefore, a brute force enumeration might be prohibitively expensive. We have found the following strategy to work well:

- Find a permutation $\mathbf{P}$ and the corresponding echelon form as described in [9]. In this approach, the attributes are ordered in one step by traversing a directed graph that represents the hierarchy of the fields in the storage formats used to represent the sparse matrices.
- Starting with the full echelon form, try various partial echelon forms for the permutation found in the previous step until a legal one is found.

```
DO i = 1, n
    DO j = 1, i − 1
        IF  (L(i, j) ≠ 0) THEN
            S1 : b(i) = b(i) − L(i, j) ∗ x(j)
        ENDIF
    ENDDO
    S2 : x(i) = b(i)/L(i, i)
```

Figure 5: Generic sparse lower triangular solution

4 IMPERFECTLY NESTED LOOPS

4.1 AN EXAMPLE

We use an example to illustrate how the ideas of the previous section can be extended to handle imperfectly nested loops. Consider the code fragment in Figure 5 that computes a solution to a lower triangular system. We assume that $\mathbf{x}$ and $\mathbf{b}$ are dense.

According to the abstract syntax tree numbering scheme of [8], the iteration vectors for the statements are:

$$\mathbf{i}_{S1} = \begin{pmatrix} i \\ 0 \\ 1 \\ j \end{pmatrix} \qquad \mathbf{i}_{S2} = \begin{pmatrix} i \\ 1 \\ 0 \\ i \end{pmatrix} \tag{32}$$

These iteration vectors encode both iteration numbers and statement order. There are two dependences:

- from all updates to $b(i)$ in $S1$ to the use of $b(i)$ in $S2$, and
- from the write to $x(i)$ in $S2$ to the read of $x(j)$ in $S1$.

The dependence matrix is:

$$\mathbf{D} = \begin{pmatrix} \mathbf{d}_1 & \mathbf{d}_2 \end{pmatrix} = \begin{pmatrix} 0 & + \\ 1 & -1 \\ -1 & 1 \\ + & 0 \end{pmatrix} \tag{33}$$

Suppose $\mathbf{L}$ is stored by column. Most of the computation is done in statement $S1$. For this statement, just as in the example of Section 3.1, we would like to interchange the i and j loops. A full transformation that does this is given by the matrix:

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{34}$$

It is easy to verify that this transformation is legal: $\mathbf{T} \cdot \mathbf{D} \succ 0$.

Here the first and the last rows of the matrix are the loop interchange transformation $\mathbf{U}^{-1} = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ *extended* into a full transformation matrix for the whole imperfectly nested loop. The sub-matrix $\mathbf{T}(2:3, 2:3)$ represents statement interchange. Without it we would not have a legal transformation.

Notice that $\mathbf{U}^{-1} = \left(\begin{smallmatrix} 0 & -1 \\ -1 & 0 \end{smallmatrix}\right)$ is also a good candidate for the partial transformation, since it brings the data access matrix for the statement $S1$ into echelon form. But the resulting full transformation

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} \qquad \text{OR} \qquad \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} \tag{35}$$

is not legal. We can adopt the same strategy as in Section 3 to build a legal $\mathbf{T}$: we form $\mathbf{T}$ row-by-row, either scaling the rows that come from $\mathbf{U}^{-1}$ (by a non-zero) or adding previous rows that came from $\mathbf{U}^{-1}$ to the current one. In effect, we are multiplying $\mathbf{U}^{-1}$ by a lower triangular matrix, thus keeping the echelon form of the data access matrix.

In our example, we start with the first and fourth rows of $\mathbf{T}$ filled:

$$\begin{pmatrix} 0 & 0 & 0 & -1 \\ ? & ? & ? & ? \\ ? & ? & ? & ? \\ -1 & 0 & 0 & 0 \end{pmatrix} \tag{36}$$

We need to change the first row, since it produces a negative direction when multiplied by $\mathbf{d}_1$. The only option at this point is to scale this row. Therefore, we negate it. To form a legal transformation, the next two rows are made to represent statement reordering. We can leave the last row unchanged to obtain the following matrix:

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} \tag{37}$$

It is easy to see that this transformation is legal. The entry $\mathbf{T}_{41} = -1$ has the effect of reversing the inner loop. The transformed code is shown in Figure 6.

```
DO j = 1, n
    S2 : x(j) = b(j)/L(j, j)
    DO i = n, j + 1, −1
        IF  (L(i, j) ≠ 0) THEN
            S1 : b(i) = b(i) − L(i, j) ∗ x(j)
```

Figure 6: Transformed generic sparse lower triangular solution

How do we generate sparse code from the transformation $\mathbf{T}$? The main difference with the perfectly nested case is that the outer loop (j) now participates in joins for 2 different statements. The join for $S1$ is

$$R_L(i, j) \bowtie R_B(i) \bowtie R_X(j) \tag{38}$$

For this statement, the j loop performs the join between the columns of $\mathbf{L}$ and the elements of $\mathbf{x}$. Since $\mathbf{L}$ stores all columns and provides a cheap search for each column, and $\mathbf{x}$ is dense, the outer loop for the $S1$ iterates from 1 to n. The join for $S2$ is

$$R_L(j, j) \bowtie R_X(j) \tag{39}$$

A run-time error is signaled if R_L does not store a particular diagonal element. Here, again, the only choice is to execute j from 1 to n, and search in R_L and R_X. Overall, the outer loop runs over the union of these two (equal) intervals. The i loop only participates in statement $S1$. The code is shown in Figure 7.

```
DO j = 1, n
    v_x = search R_X for j
    v_L = search R_L(*, j) for j
    v_b = search R_B for j
    v_x = v_b / v_L
    DO ⟨i, w_L⟩ ∈ R_L(*, j), i > j, in "reverse"
        w_b = search R_B for i
        w_b = w_b − w_L * v_x
```

Figure 7: Column-oriented sparse lower triangular solution

```
DO k = 1, n
    DO j = k, n
        DO l = 1, k − 1
            S1 : A(j, k) = A(j, k) − A(k, l) * A(j, l)
        ENDDO
    ENDDO
    S2 : A(k, k) = sqrt(A(k, k))
    DO i = k + 1, n
        S3 : A(i, k) = A(i, k)/A(k, k)
    ENDDO
ENDDO
```

Figure 8: Left-looking Cholesky factorization

4.2 GENERAL FRAMEWORK

The imperfectly nested loop transformation framework of [8] allows for transformations which are combinations of statement reorderings and linear transformations along *disjoint* downward paths of the abstract syntax tree (AST) for the loop. For example, consider the loop nest in Figure 8, that computes Cholesky factorization of a matrix. The AST for this loop is shown on the left in Figure 10. [8] allows transformations which would combine (e.g. permute or skew) the k, j and l loops, but not j and i loops. A transformation into right-looking code is an example of a valid transformation. It permutes the k, j and l loops and reorders the children of the root. The resulting code is shown in Figure 9, and the AST for this loop nest is shown on the right in Figure 10. The dashed line marks the path in the AST along which the loop variables were combined.

[8] provides a completion procedure to build a full legal transformation out of the first few rows. This procedure is similar to the one used for perfectly nested loops in [10]. The main difference is the necessity to maintain a special structure for the transformation matrix, that reflects the reordering of the AST. This point is discussed in more detail in [8].

For lack of space, we do not describe our extension of the completion procedure here, but illustrate its behavior using the Cholesky factorization example. We start with the loop nest in Figure 8. The dependences for this example are:

$$\mathbf{D} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & -1 \\ -1 & 1 & 0 & 0 \\ 0 & -1 & -1 & 1 \\ + & 0 & + & + \\ 0 & 0 & - & + \\ 0 & + & + & 0 \end{pmatrix} \tag{40}$$

The query for the update statement $S1$ is:

$$R_A(j,k) \bowtie R_A(j,l) \bowtie R_A(k,l) \tag{41}$$

```
DO l = 1, n
    S2 : A(l, l) = sqrt(A(l, l))
    DO i = l + 1, n
        S3 : A(i, l) = A(i, l)/A(l, l)
    ENDDO
    DO k = l + 1, n
        DO j = k, n
            S1 : A(j, k) = A(j, k) − A(k, l) * A(j, l)
        ENDDO
    ENDDO
ENDDO
```

Figure 9: Right-looking Cholesky factorization

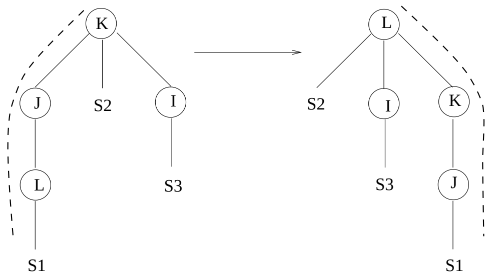

Figure 10: Transformation of the AST

If the matrix is stored using compressed column scheme, then the order of the joins should be l - k - j. The corresponding loop transformation for this statement is:

$$\mathbf{U}^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{42}$$

We start with the full transformation being:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? \\ ? & ? & ? & ? & ? & ? & ? \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{43}$$

The last two rows of $\mathbf{U}^{-1}$ are placed at the end for now, since we do not know the AST reordering yet. Rows 2 through 4 describe the reordering of the children of the root of the AST. To maintain a legal transformation, we use the permutation $\left(\begin{smallmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{smallmatrix}\right)$. This tells us that the last two rows of $\mathbf{U}^{-1}$ should become the 5th and 6th rows of the transformation. Completing the last row so that the whole matrix is non-singular, we get the following matrix:

$$\mathbf{T} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{44}$$

```
DO l = 1, n
    u = search R_A(*, l) for l
    S2 : u = sqrt(u)
    DO ⟨i, v⟩ ∈ R_A(*, l), i > l
        S3 : v = v/u
    ENDDO
    DO ⟨k, v⟩ ∈ R_A(*, l), k > l
        DO ⟨j, w, u⟩ ∈ R_A(*, k) ⋈ R_A(*, l), j ≥ k
            S1 : w = w − v ∗ u
        ENDDO
    ENDDO
ENDDO
```

Figure 11: Sparse Cholesky factorization

This transformation results in the right-looking Cholesky factorization shown in Figure 9. The sparse computation is shown in Figure 11.

5 PREVIOUS WORK

The results of this paper are largely complementary to the work done by Bik and Wijshoff on determining sparsity predicates, analyzing coarse-grain structure in sparse matrices and choosing storage orientation [1–6]. However, we improve on their access reshaping and guard encapsulation techniques as follows.

The access reshaping method of [2] generates the final loop transformation by composing a sequence of *legal* transformations for each loop of the original loop nest. This is more restrictive than our algorithm which uses the echelon form of the data access matrix to generate a legal transformation directly, since such a transformation might not decompose into a product of legal transformations. Theorem 1 tells us that if there is a legal transformation into an echelon form, then our algorithm will find one. No such guarantee exists for access reshaping.

Another important difference is in the way that enumerations over the sparse data structures are incorporated into the transformed loop nest. If a loop enumerates over multiple sparse data structures (as in the inner product example of Section 2.1), we have many choices for implementing this *simultaneous* enumeration (that is, join). Guard encapsulation performs enumeration over exactly one data structure per loop in a loop nest, and generates searches (possibly speeded up by *access pattern expansion*) for the rest of the data structures. In our framework, this is equivalent to performing a Hash-Join. However, as we have shown in [9], the Merge-Join algorithm might be a better alternative in some contexts, but the guard encapsulation technique does not explore this option.

Finally, Bik and Wijshoff do not have a unified framework for dealing with imperfectly nested loops. In particular, the loop permutation that our algorithm computes for Cholesky factorization cannot be derived in their framework, and can only be represented as a sequence of loop permutations and statement reorderings. It is not clear how such sequence can be derived automatically.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we have shown how the sparse compilation techniques of [9] can be extended to handle imperfectly nested loops with dependences by using the loop transformation framework of [8]. Our method is based on the following observations:

- Two different permuted echelon forms (having the same row permutation) of a matrix are related by a lower triangular matrix. This allows us to start with an illegal transformation and modify it into a legal one.
- The completion procedure of [8] can be extended to compute a legal transformation that brings the data access matrix into a desired echelon form.

Currently we only allow compressed hyper-plane storage formats in the compilation of loops with dependences. This is necessary because satisfying dependences requires relating the order of enumeration of the sparse arrays to the order of the loops. While this allows us to generate sparse code automatically for a variety of formats, these simple storage formats are inadequate if we want to exploit special structure in the matrices as is done, for example, in the BlockSolve package [7]. We are currently exploring ways of extending our techniques to such data structures.

References

[1] Bik, A. *Compiler Support for Sparse Matrix Computations.* PhD thesis, Leiden University, The Netherlands, May 1996.

[2] Bik, A. J., Knijnenburg, P. M., and Wijshoff, H. A. Reshaping access patterns for generating sparse codes. In *Seventh Annual Workshop on Languages and Compilers for Parallel Computing* (Aug. 1994).

[3] Bik, A. J., and Wijshoff, H. A. Non-zero structure analysis. In *International Conference on Supercomputing* (1994), pp. 226 – 235.

[4] Bik, A. J., and Wijshoff, H. A. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing 31* (1995), 14–24.

[5] Bik, A. J., and Wijshoff, H. A. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems 7*, 2 (1996), 109 – 126.

[6] Bik, A. J., and Wijshoff, H. A. The use of iteration sparse partitioning to construct representative simple sections. *Journal of Parallel and Distributed Computing 34* (1996), 95 – 110.

[7] Jones, M. T., and Plassmann, P. E. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Tech. Rep. ANL–95/48, Argonne National Laboratory, Dec. 1995.

[8] Kodukula, I., and Pingali, K. Transformations for imperfectly nested loops. In *Supercomputing* (Nov. 1996), ACM SIGARCH and IEEE Computer Society, ACM Press. (http://www.supercomp.org).

[9] Kotlyar, V., Pingali, K., and Stodghill, P. A relational approach to sparse matrix compilation. Submitted to *EuroPar* (1997). Also available as Cornell Computer Science Tech. Report 97–1627 (http://cs-tr.cs.cornell.edu).

[10] Li, W., and Pingali, K. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems 11*, 4 (Nov. 1993), 353–375.

[11] Saad, Y. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing 10*, 6 (Nov. 1989), 1200–1232.

[12] Ullman, J. D. *Principles of Database and Knowledge-Base Systems, v. I and II.* Computer Science Press, 1988.