On Automatic Data Structure Selection and Code Generation for Sparse Computations*

Aart J.C. Bik 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 and harryw@cs.leidenuniv.nl

Abstract

Traditionally restructuring compilers were only able to apply program transformations in order to exploit certain characteristics of the target architecture. Adaptation of data structures was limited to e.g. linearization or transposing of arrays. However, as more complex data structures are required to exploit *characteristics of the data* operated on, current compiler support appears to be inappropriate. In this paper we present the implementation issues of a restructuring compiler that automatically converts programs operating on dense matrices into sparse code, i.e. after a suited data structure has been selected for every dense matrix that in fact is sparse, the original code is adapted to operate on these data structures. This simplifies the task of the programmer and, in general, enables the compiler to apply more optimizations.

Index Terms: Restructuring Compilers, Sparse Computations, Sparse Matrices.

1 Introduction

Development and maintenance of sparse codes is a complex task. The user has to deal with complicated data structures to exploit sparsity of matrices with respect to *storage requirements* and *computational time*. Additionally, the application of conventional transformations to sparse programs becomes more complex because the functionality of the code is obscured. Therefore, in [7, 8] we have examined if it is possible to let the compiler handle sparsity, i.e. the program is written in a dense format, while the compiler performs program and data structure transformations to deal with the fact that some of the matrices operated on are sparse. We have proposed a data structure selection and transformation method that enables the compiler to select appropriate data structures. Since run-time behavior of sparse codes is heavily dependent on the input data, efficient code is only achieved if certain peculiarities of sparse matrices are accounted for in this selection.

In this paper we generalize this method and present implementation issues. In section 2, a brief description of the data structure selection and transformation method is given, in combination with the notation used throughout this paper. Since reduction of sparse overhead is only obtained if certain constraints on the data structure are satisfied, we present a mechanism to select a data structure according to these constraints in section 3. In section 4, the generation of code is discussed, illustrated with examples in section 5. Finally, we state issues for future research.

2 Background

Exploitation of sparsity requires identification of statements of which some instances are nullified, and selection of an appropriate *compact* data structure.

2.1 Sparsity Exploitation

A **sparse matrix** A is defined by its nonzero structure: $Nonz_A = \{(i,j) \in I_A \times J_A | a_{ij} \neq 0\}$, where $I_A = \{1,..,m\}$ and $J_A = \{1,..,n\}$ determine the **index set** $I_A \times J_A$ of the enveloping dense matrix. Reduction of *storage requirements* is achieved by only storing the nonzero elements in **primary storage**, although some storage is necessary to reconstruct the underlying matrix, referred to as **overhead storage**. Elements that are stored explicitly are called **entries**. Set E_A with $Nonz_A \subseteq E_A \subseteq I_A \times J_A$ is used to indicate the indices of all entries. If E_A changes during program execution, a *dynamic* data structure is required to handle the insertion of a new entry (**creation**) and the deletion of an entry that becomes a zero element (**cancellation**), while a *static* data structure might be used otherwise.

Because the original code operates on dense matrices, every occurrence of a sparse matrix A appears as a two-dimensional array in an **indexed statement** S of degree l, where the strides of all surrounding loops have been normalized to 1:

*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. This paper is an extended abstract of technical report 93-04 [9].

```
DO I_1 ∈ V_1
  ...
    DO I_l ∈ V_l
S :   ... A(f_1(I⃗), f_2(I⃗)) ...
    ENDDO
  ...
ENDDO
```

The bounds of **execution set** V_i might depend on indices $\mathtt{I}_1, \ldots, \mathtt{I}_{i-1}$. Only linear subscript functions with integer coefficients are considered. Consequently, both functions can be represented by a single mapping $F_A : \mathbf{Z}^l \rightarrow \mathbf{Z}^2$ of the form $F_A(\vec{\mathtt{I}}) = \vec{m}_A + M_A \cdot \vec{\mathtt{I}}$:

$$\begin{pmatrix} m_{10} \\ m_{20} \end{pmatrix} + \begin{pmatrix} m_{11} & \ldots & m_{1l} \\ m_{21} & \ldots & m_{2l} \end{pmatrix} \cdot \vec{\mathtt{I}}$$

Subscript bounds are not violated, i.e. $F_A(\vec{\mathtt{I}}) \in I_A \times J_A$ for all valid $\vec{\mathtt{I}}$. A loop at nesting depth i is called **controlling** if $(m_{1i} \neq 0) \vee (m_{2i} \neq 0)$, or **non-controlling** otherwise. The indices of all elements operated on by different **instances** of statement `S` in one execution of the innermost controlling loop at nesting depth c, i.e. $(m_{1i} = 0) \wedge (m_{2i} = 0)$ for $c < i \leq l$, is referred to as an **access pattern** of the corresponding occurrence of matrix A:

$$P_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}} = \{F_A(\vec{\mathtt{I}}) | \mathtt{I}_c \in V_c\}$$

Access patterns are called **row-wise** if $(m_{1c} = 0) \wedge (m_{2c} \neq 0)$, **column-wise** if $(m_{1c} \neq 0) \wedge (m_{2c} = 0)$, while all remaining access patterns are referred to as **diagonal-wise**.[1] The **direction** of an access pattern is $\vec{d}_A = (m_{1c}, m_{2c})$.

The notion of an **abstract data structure** $\mathtt{A}'$ is introduced to reason about a compact data structure as long as no actual implementation has been selected. For an occurrence $\mathtt{A}(F_A(\vec{\mathtt{I}}))$ a **guard** '$F_A(\vec{\mathtt{I}}) \in E_A$' is used in a multiway IF-statement to differentiate between operations on an entry or a non-entry. Function $\sigma_A : E_A \rightarrow AD_A$ maps indices of an entry to its address in $\mathtt{A}'$. Consequently, the following notation is used for an assignment of an arbitrary expression to an element, where function new_A returns a new address in $\mathtt{A}'$ and adapts σ_A, E_A and AD_A accordingly as side-effect to account for creation:

```
IF F_A(I⃗) ∈ E_A THEN
  A'[σ_A(F_A(I⃗))] = ...
ELSEIF F_A(I⃗) ∉ E_A THEN
  A'[new_A(F_A(I⃗))] = ...
ENDIF
```

This kind of notation offers the possibility to eliminate the branches in which sparsity can be exploited to save *computational time*, because statement instances where a zero is assigned to a non-entry or where an arbitrary variable is updated with a zero do not have to be executed. For example, one branch remains for statement '`ACC = ACC + A(`$F_A(\vec{\mathtt{I}})$`)`':

```
IF F_A(I⃗) ∈ E_A THEN
  ACC = ACC + A'[σ_A(F_A(I⃗))]
ENDIF
```

The identification of statements that can exploit sparsity is done by means of an attribute grammar [1, 17], based on a context free grammar for assignment statements.[2] The following semantic rules are used to associate the strongest **condition** ψ, constructed from guards, with each expression in a synthesized attribute *nz*, to indicate when the value of this expression is nonzero, under the assumption that the value of dense variables and entries is always nonzero:

Production	Semantic Rule
$E \rightarrow E_1$ + E_2	E.nz = dis(E_1.nz, E_2.nz);
$E \rightarrow E_1$ − E_2	E.nz = dis(E_1.nz, E_2.nz);
$E \rightarrow E_1$ * E_2	E.nz = con(E_1.nz,E_2.nz);
$E \rightarrow E_1$ / E_2	E.nz = E_1.nz;
$E \rightarrow E_1$ ** E_2	E.nz = E_1.nz;
$E \rightarrow$ − E_1	E.nz = E_1.nz;
$E \rightarrow$ (E_1)	E.nz = E_1.nz;
E → var	E.nz = var.grd;
E → const	E.nz = is_zero(const.val) ? 'false':'true';

The guard of each variable is supplied as synthesized attributed *grd*, and is 'true' for dense variables. The value of constants is supplied in attribute *val*. Functions *dis* and *con* construct a disjunction and conjunction respectively of the arguments. The result is simplified according to logical equivalences that are based on e.g. absorption laws or the fact that 'true' is the identity of the '∧'-operator. The subtlety of a zero right operand for '/'- and '**'-operators is ignored. For example, (simplified) condition ψ is shown below for some expressions, if A and B are sparse matrices:

Expression	ψ
`A(I,J)+A(I,J)*B(I,J)`	'$(\mathtt{I},\mathtt{J}) \in E_A$'
`A(I+1,J)*(X+B(I,J))`	'$(\mathtt{I}+1,\mathtt{J}) \in E_A$'
`A(I,J)+B(I,J)/5.0`	'$(\mathtt{I},\mathtt{J}) \in E_A \vee (\mathtt{I},\mathtt{J}) \in E_B$'
`X+A(I,J)`	'true'
`-0.0*(X+A(I,J))`	'false'

From these conditions, the strongest condition χ is associated with each statement in attribute *cnd*, that indicates the instances that must be executed. A pointer to the left-hand side variable in an assignment statement, supplied as synthesized attribute *nm*, is copied in an inherited attribute *lhs* of the right-hand side expression, and is passed down the parse tree:

Production	Semantic Rule
stmt → var = E;	E.lhs = var.nm;
$E \rightarrow E_1$ + E_2	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1$ − E_2	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1$ * E_2	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1$ / E_2	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow E_1$ ** E_2	E_1.lhs = E_2.lhs = E.lhs;
$E \rightarrow$ − E_1	E_1.lhs = E.lhs;
$E \rightarrow$ (E_1)	E_1.lhs = E.lhs;

[1] Occurrences without controlling loops give rise to **scalar-wise** singleton access patterns.

[2] The ambiguity of this grammar is resolved by assignment of the usual precedence and associativity to all operators.

From attributes *nz* and *lhs*, another synthesized attribute *ne* associated with expressions is constructed that records when the right-hand side expression is not equal to the left-hand side variable:

Production	Semantic Rule
E → E_1 + E_2	E.ne = con(dis(E_1.ne, E_2.nz), dis(E_1.nz,E_2.ne));
E → E_1 − E_2	E.ne = dis(E_1.ne,E_2.nz);
E → (E_1)	E.ne = E_1.ne;
E → var	E.ne = equal(var.nm.E.lhs) ? 'false':'true';
/* others */	E.ne = 'true';

For example, the value of attribute *ne* associated with the right-hand side expression in assignment statement '`ACC=ACC+A(I,J)*X`' is '$(\mathtt{I},\mathtt{J}) \in E_A$'. Evaluation dependences are depicted in figure 1:

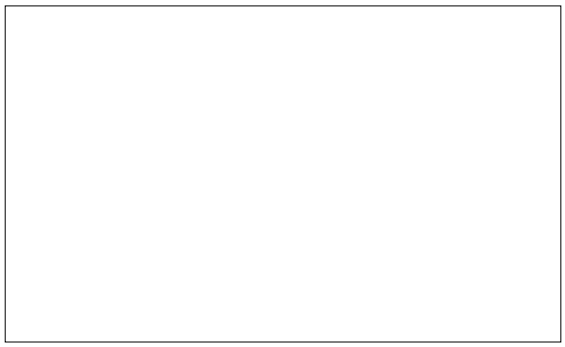


Figure 1: Parse Tree

Finally, the condition of each statement is determined. An instance of an assignment statement must be executed if the left- and right-hand sides are not equal, and at least one of these expressions is nonzero: for production 'stmt → var = E;', the semantic rule is 'stmt.cnd = con(E.ne,dis(var.grd,E.nz));'. The condition χ associated with some statements is shown below. If χ ='false', the statement can be eliminated since none of its instances have to be executed.

Statement	χ
`B(I,J)=B(I,J)+A(I,J)`	'$(\mathtt{I},\mathtt{J}) \in E_A$'
`A(2*I,J)=X*A(2*I,J)`	'$(2\mathtt{I},\mathtt{J}) \in E_A$'
`A(I,J)=X`	'true'
`A(I,J)=A(I,J)`	'false'
`A(I,J)=A(I,J)*B(I,J)` `+C(I,J)`	'$(\mathtt{I},\mathtt{J}) \in E_A$ $\vee (\mathtt{I},\mathtt{J}) \in E_C$'

Effectively, condition χ consists of the disjunction of all conditions in the branches that remain in the corresponding IF-statement, as is illustrated below for statement '`A(I,J)=B(I,J)`' with condition '$(\mathtt{I},\mathtt{J}) \in E_A \vee (\mathtt{I},\mathtt{J}) \in E_B$', where subroutine del_A is used to indicate cancellation:

```
IF (I,J) ∈ E_A ∧ (I,J) ∈ E_B THEN
  A'[σ_A(I,J)] = B'[σ_B(I,J)]
ELSEIF (I,J) ∈ E_A ∧ (I,J) ∉ E_B THEN
  CALL del_A(I,J)
ELSEIF (I,J) ∉ E_A ∧ (I,J) ∈ E_B THEN
  A'[new_A(I,J)] = B'[σ_B(I,J)]
ENDIF
```

2.2 Overhead Elimination

The presence of guards and σ_A-lookups reflects overhead that is due to the fact that a compact data structure must be scanned to determine *if* and *where* an entry is stored. Additionally, skipping operations by means of conditionals does not result in much gain in execution time, because usually such tests are equally expensive [13, 24]. Therefore, techniques to eliminate this kind of overhead are required.

2.2.1 Guard Encapsulation

The disjunction of the conditions associated with all statements in a loop body is called the **loop condition**. It indicates the iterations in which at least one statement instance has to be executed. A *guard* ψ **dominates** a *condition* χ, iff. $\chi \rightarrow \psi$ is a tautology. So, if a guard ψ dominates a loop condition χ, iterations in which ψ does not hold do not have to be executed. This can be achieved by **encapsulation** of a dominating guard '$F_A(\vec{\mathtt{I}}) \in E_A$' in the execution set of a directly surrounding *controlling* loop. Because the resulting execution set $\{\mathtt{I}_c \in V_c | F_A(\vec{\mathtt{I}}) \in E_A\}$ cannot be generated directly in general, the following conversion is performed:

```
DO I_c ∈ V_c
 IF F_A(Ī) ∈ E_A THEN
  ACC = ACC + A'[σ_A(F_A(Ī))]
 ENDIF
ENDDO      ↓
   DO AD ∈ PAD_A^{I_1,...,I_{c-1}}
    ACC = ACC + A'[AD]
   ENDDO
```

If we can easily obtain $PAD_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}} = \{\sigma_A(t) | t \in (P_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}} \cap E_A)\} \subseteq AD_A$, only iterations in which an entry (an element for which the dominating guard holds) is operated on, are executed. Fewer iterations result since the size of the execution set decreases, while test overhead is eliminated completely. Because $\mathtt{AD} = \sigma_A(F_A(\vec{\mathtt{I}}))$ holds and σ_A is invertible, the value of $\mathtt{I}_c$ can be restored by using the following equations, where $\pi_i \cdot \vec{x} = x_i$:

$$\begin{cases} \pi_1 \cdot \sigma_A^{-1}(AD) = m_{10} + m_{11} I_1 \ldots + m_{1c} I_c \\ \pi_2 \cdot \sigma_A^{-1}(AD) = m_{20} + m_{21} I_1 \ldots + m_{2c} I_c \end{cases}$$

Guard encapsulation is feasible if the following constraints on the data structure are satisfied: (1) fast generation of addresses in $PAD_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}}$ is supported for all valid $\mathtt{I}_1, \ldots, \mathtt{I}_{c-1}$, (2) if a dependence is carried by the $\mathtt{I}_c$-loop, address-generation corresponds to the original execution order of this loop,[3] and (3) to restore the value of $\mathtt{I}_c$, $\pi_1 \cdot \sigma_A^{-1}$ values are available if $m_{1c} \neq 0$ or $\pi_2 \cdot \sigma_A^{-1}$ values if $m_{2c} \neq 0$ (i.e. row or column index of each entry). For example, in the following fragment encapsulation of guard '$(\mathtt{I}, \mathtt{I}+\mathtt{J}) \in E_A$', that dominates the (identical) loop condition, in the execution

[3] Usually, dependences caused by accumulations are ignored.

set of the J-loop is possible and requires $\pi_2 \cdot \sigma_A^{-1}(\mathtt{AD})$ to reconstruct the value of J. Addresses in $PAD_A^{\mathtt{I}}$ belong to entries along $P_A^{\mathtt{I}} = \{(\mathtt{I}, \mathtt{I}+\mathtt{J}) | 1 \leq \mathtt{J} \leq 3\}$:

```
DO I = 1, M
  DO J = 1, 3
    IF (I,I+J) ∈ E_A THEN
      ACC = ACC + A(I,I+J) * J
    ENDIF
  ENDDO
ENDDO          ↓
   DO I = 1, M
     DO AD ∈ PAD_A^I
       J = π_2 · σ_A^{-1}(AD) - I
       ACC = ACC + A'[AD] * J
     ENDDO
   ENDDO
```

Constraint (1) can be relaxed at the expense of run-time overhead, if fast generation of addresses in $\overline{PAD}_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}}$ that corresponds to entries along an **enveloping** access pattern $\overline{P}_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}} \supseteq P_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}}$ is required, under the condition that it can be determined at run-time whether an entry corresponds to the actual access pattern. If only **longitudinal** enveloping access patterns are considered this can be done for each entry by testing inclusion '$\mathtt{I}_c \in V_c$' on the restored loop index (see section 4.2). A longitudinal enveloping access pattern $\overline{P}_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}}$ has the following form, where $g = \gcd(m_{1c}, m_{2c})$, $T_1 \leq g \cdot \min(V_c)$, and $T_2 \geq g \cdot \max(V_c)$:

$$\{(\ldots + \tfrac{m_{1c} \cdot t}{g}, \ldots + \tfrac{m_{2c} \cdot t}{g}) | T_1 \leq t \leq T_2\}$$

Examples are $P_A^{\mathtt{I}} = \{(100 - 3\mathtt{J}, \mathtt{I}) | 1 \leq \mathtt{J} \leq 33\}$ with $\overline{P}_A^{\mathtt{I}} = \{(100-t, \mathtt{I}) | 3 \leq t \leq 99\}$ and $P_A = \{(4\mathtt{J}, 2\mathtt{J}) | 1 \leq \mathtt{J} \leq 25\}$ with $\overline{P}_A = \{(2t, t) | 2 \leq t \leq 50\}$.

Complications occur for guard encapsulation if creation or cancellation along the (enveloping) access pattern affects the corresponding address set. We distinguish **forward** and **backward** creation, corresponding to insertions that affect the value of an encapsulated guard in following or preceding iterations respectively. For some data structures, creation along *arbitrary* access patterns might affect other address sets if garbage collection is occasionally required. Deletion of an entry that corresponds to the encapsulated guard is referred to as **inward** cancellation. If inward cancellation cannot occur, the corresponding dominating guard in a loop body for which $c < l$ holds can be **hoisted** out the innermost non-controlling loops.

Note that if entries are sorted on column or row index, encapsulation of more general conditions, like conjunctions and disjunctions of guards, becomes possible by generating code that performs an **in-phase scan** of the associated data structures [13].

2.2.2 Access Pattern Expansion

Another way to reduce overhead is based on the idea to avoid binary operations that involve two sparse vectors and to reduce test overhead of certain occurrences that cannot exploit sparsity, with work dependent on the number of entries only [11, 13, 24]. The compiler can decide to **expand** a sparse access pattern before it is operated on with a **scatter**-operation, so that operations on this dense representation do not suffer from sparse lookup overhead. If assignments occur, storage is obtained or released directly to account for creation or cancellation if necessary. This can be done with a variant on the switch technique [24], where a bit array records which elements are entries. The actual values are stored back afterwards with a **gather**-operation. These operations are feasible under constraints that are similar to the requirements for guard encapsulation: (1) the address set of the (enveloping) access pattern is available, and (2) appropriate index information is available per entry.

2.2.3 Nonzero Structures

If with information about the nonzero structure, that is either user-supplied or obtained by on-file analysis, it can be determined that certain (enveloping) access patterns are rather dense, or become dense by frequently occurring creation, the compiler can decide to use the dense representation as *only* storage during the whole program. This eliminates the lookup and creation overhead that is inherent to sparse data structures, while the number of redundant operations performed is probably small.

Information about the nonzero structure can also be used to apply **iteration space reduction**. Consider for example the code for $\vec{y} \leftarrow A \cdot \vec{x}$ for a band matrix A, i.e. $(a_{ij} \neq 0) \rightarrow (-b_1 \leq j - i \leq b_2)$. After application of the unimodular transformation $U = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix}$ [6, 27], all elements a_{ij} with $j - i = \mathtt{J}$ are accessed in one execution of the I-loop:

```
  DO I = 1, M
    DO J = 1, N
      Y(I) = Y(I) + A(I,J) * X(J)
    ENDDO
  ENDDO          ↓
DO J = 1 - M, N - 1
  DO I = MAX(1,1-J), MIN(M,N-J)
    Y(I) = Y(I) + A(I,J+I) * X(J+I)
  ENDDO
ENDDO
```

Compile-time encapsulation in the execution set of the *outermost* controlling loop results in a reduced execution set $\{-b_1, \ldots, b_2\}$ for the J-loop, and dense storage of the remaining diagonals becomes desirable.

2.3 Storage

Access patterns that are dense are stored in full-sized arrays. Sparse access patterns are stored as sparse vectors in data structures (D1) or (L1), illustrated in figure 2 (inspired on existing data structures [13, 24, 26, 30]). In these implementations, the

numerical values of entries are stored in an array AVAL, while corresponding column or row indices are stored in a parallel integer array AIND. The implementations differ in the way each access pattern is stored: consecutively (D1), or as a linked list (L1), and the information required per access pattern: first and last index (D1), or a pointer to the first entry (L1). Enough **working space** must be available to account for creation in case of dynamic data structures. An ordering (determined by links for (L1)) between entries on the associated index within one access pattern can be maintained at the penalty of more expensive insertions and deletions.

An insertion or deletion in (L1) only affects the values of a few links. For (D1) the addresses of other entries in the same access pattern might alter as a result of data movement, while *extra* data movement results if an insertion is performed in an access pattern with no directly surrounding space.[4]

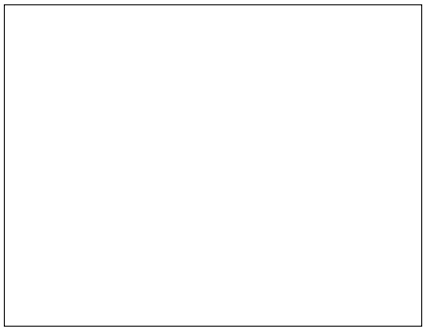


Figure 2: Sparse Data Structure

3 Storage Selection

Most overhead reduction would be obtained if all imposed constraints on the actual data structure of sparse matrices were satisfied. However, because usually elements are involved in differently shaped access patterns, this would demand for storage schemes with much overhead storage that are expensive to maintain in order to support fast generation of addresses in all address sets. For example, storing entries according to row- *and* column-wise access patterns requires a linked-lists implementation, where four integers are associated with each entry (row and column index, and two links). The overhead storage requirements of this scheme can be reduced, but at the expense of more expensive lookups [10]. Additionally, to simplify data structure construction and code generation, a simple correspondence between address sets and loop indices must hold.

[4] The access pattern is moved to the end of the array where most working space is kept. If insufficient space remains, garbage collection is required to compress all data, which might affect the addresses of *all* entries.

Overhead storage can be kept reasonably small if only the constraints of **consistent** access patterns, i.e. that are either *disjoint* or *identical*, are satisfied. Data structure selection for each sparse matrix in a program is, therefore, controlled by an **inconsistency table**, which initially has the following form:

A	d	c	r
r	0	0	0
c	0	0	
d	0		

Every table-entry contains the number of static inconsistencies between row-, column- or diagonal-wise access patterns of occurrences of a sparse matrix A. Since an implementation is straightforward to select in case of an empty table, reducing the number of nonzero entries in this table is the most important goal during data structure selection. Clearly, compromises have to be made in some cases.

3.1 Self-Inconsistencies

Although all elements along one access pattern are distinct, it is possible that access patterns $P_A^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}}$ of one occurrence overlap for different $\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}$. If such access patterns are not identical, a static **self-inconsistency** arises, as is illustrated for row-wise access patterns in the following example:

```
DO J = 1, N / 2
  DO K = 1, J
    ACC = ACC + A(2,2*K)
  ENDDO
ENDDO
```

A	d	c	r
r	0	0	1
c	0	0	
d	0		

Encapsulation of '$(2, 2\mathtt{K}) \in E_A$' in the execution set of the K-loop, for instance, is feasible if fast generation of the addresses of entries at even positions in *every prefix* of the second row is supported. This abundance of constraints is reflected in the inconsistency between $P_A^{\mathtt{J}} = \{(2,\mathtt{K}) | 1 \leq \mathtt{K} \leq \mathtt{J}\}$ for different values of J. Consideration of e.g. $\overline{P}_A^{\mathtt{J}} = \{(2,t) | 1 \leq t \leq \mathtt{N}\}$ achieves self-consistency, and requires fast generation of addresses of entries in only one access pattern, justifying the simplification made in the next section.

3.2 Collection

In order to get a better grip on all access patterns of *one* occurrence, these are characterized by an **access pattern collection** C_A^i. This is a possibly enveloping, but also consistent and simple description of all access patterns. First, a conservative approximation of the index set of all used elements a_{xy} is constructed in terms of a variant on the **simple section** [3, 4]:

$$\left\{ \begin{array}{c} l_1 \leq x \leq u_1 \\ l_2 \leq y \leq u_2 \\ l_3 \leq r_1 \cdot x + r_2 \cdot y \leq u_3 \\ l_4 \leq r_2 \cdot x - r_1 \cdot y \leq u_4 \end{array} \right. \qquad (3.2)$$

We use $\vec{r} = (\frac{s \cdot |m_{1c}|}{\gcd(m_{1c}, m_{2c})}, \frac{|m_{2c}|}{\gcd(m_{1c}, m_{2c})})$, where $s = (m_{1c} \cdot m_{2c} \geq 0)$? 1:-1, in case of diagonal-wise access patterns, or $\vec{r} = (1, 1)$ otherwise. Consequently, two **section bounds** are parallel to the access patterns, and fairly complex shaped sections can be dealt with. Since the correspondences $x = f_1(\vec{\mathtt{I}})$ and $y = f_2(\vec{\mathtt{I}})$ hold, the values of $\vec{l}$ and $\vec{u}$ in a section of form (3.2) are easily obtained by successive consideration of worst-case values for $\mathtt{I}_i$ in decreasing order, since execution set V^i might depend on $\mathtt{I}_j$, for all $j < i$. Consider, for example, $P_A^{\mathtt{I},\mathtt{J}} = \{(\mathtt{I}+\mathtt{K}, \mathtt{J}+2\mathtt{K}) | 1 \leq \mathtt{K} \leq 3\}$ for $1 \leq \mathtt{I} \leq 3$ and $1 \leq \mathtt{J} \leq 3$. The following section results, where the value of e.g. u_4 is determined as $2x - y = 2\mathtt{I} - \mathtt{J} \leq 2\mathtt{I} - 1 \leq 5$:

$$\begin{cases} 2 \leq x \leq 6 \\ 3 \leq y \leq 9 \\ 8 \leq x + 2y \leq 24 \\ -1 \leq 2x - y \leq 5 \end{cases}$$

The access pattern collection consists of an enumeration of *all* access patterns within this section: $C_A^i = \{(R^i + \frac{s \cdot |m_{1c}| \cdot t}{\gcd(m_{1c}, m_{2c})}, C^i + \frac{|m_{2c}| \cdot t}{\gcd(m_{1c}, m_{2c})}) | T_1^i \leq t \leq T_2^i\}$, for $I_1 \leq i \leq I_2$. In this way, distinctions between partially overlapping or multiple traversed access patterns are eliminated, while the direction is normalized such that the y-component is non-negative.

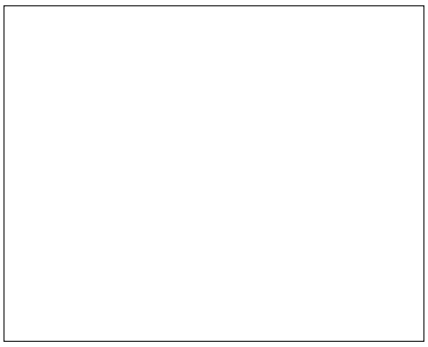
Offsets and bounds for row-, column- and diagonal-wise access patterns are given in the following tables, where $l^* = l_1$ and $u^* = u_1$ if $s = 1$, or $l^* = u_1$ and $u^* = l_1$ otherwise:

	R^i	C^i	I_1	I_2
row	i	0	l_1	u_1
col.	0	i	l_2	u_2
diag.	$v_2 \cdot i$	$-v_1 \cdot i$	l_4	u_4

T_1^i
$\max(l_2, l_3 - i, i - u_4)$
$\max(l_1, l_3 - i, i + l_4)$
$\lceil \max(\frac{l^* - v_2 \cdot i}{r_1}, \frac{l_2 + v_1 \cdot i}{r_2}, \frac{l_3 + (r_2 v_1 - r_1 v_2) \cdot i}{r_1^2 + r_2^2}) \rceil$

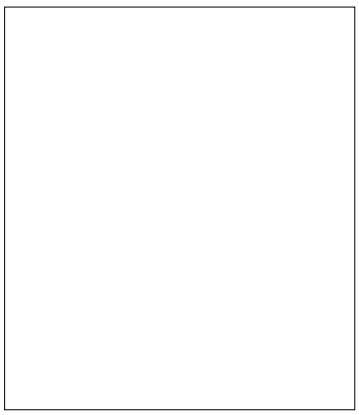
T_2^i
$\min(u_2, u_3 - i, i - l_4)$
$\min(u_1, u_3 - i, i + u_4)$
$\lfloor \min(\frac{u^* - v_2 \cdot i}{r_1}, \frac{u_2 + v_1 \cdot i}{r_2}, \frac{u_3 + (r_2 v_1 - r_1 v_2) \cdot i}{r_1^2 + r_2^2}) \rfloor$

Diagonal-wise enumeration along $r_2 \cdot x - r_1 \cdot y = i$ for successive i is obtained by application of unimodular transformation $U = \begin{pmatrix} r_2 & v_1 \\ -r_1 & v_2 \end{pmatrix}$ on the row-wise enumeration, where $\vec{v}$ is taken such that $\det(U) = r_2 \cdot v_2 + r_1 \cdot v_1 = 1$. Such integers can be found during computation of $\gcd(m_{1c}, m_{2c})$ as shown in [6]. Consequently, the access pattern collection of the previous example is $C_A^i = \{(t, 2t - i) | \lceil \max(2, \frac{3+i}{2}) \rceil \leq t \leq \lfloor \min(6, \frac{9+i}{2}) \rfloor\}$ for $-1 \leq i \leq 5$:



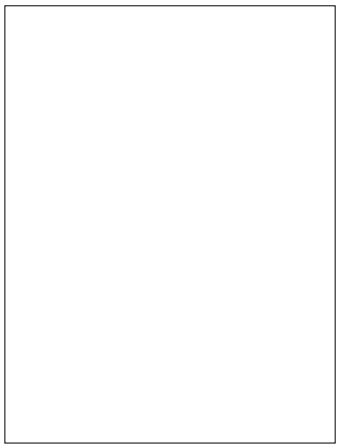
The collection of $P_A^{\mathtt{I},\mathtt{J}} = \{(\mathtt{I}+\mathtt{J}, \mathtt{K}) | 1 \leq \mathtt{K} \leq \mathtt{I}+\mathtt{J}\}$ for $1 \leq \mathtt{I} \leq 3$ and $1 \leq \mathtt{J} \leq 2$ is $C_A^i = \{(i, t) | 1 \leq t \leq i\}$ for $2 \leq i \leq 5$, as depicted in the following figure. Section bound $0 \leq x - y$ results, because the minimum value of `-K` is `-I-J`:

$$\begin{cases} 2 \leq x \leq 5 \\ 1 \leq y \leq 5 \\ 3 \leq x + y \leq 10 \\ 0 \leq x - y \leq 4 \end{cases}$$



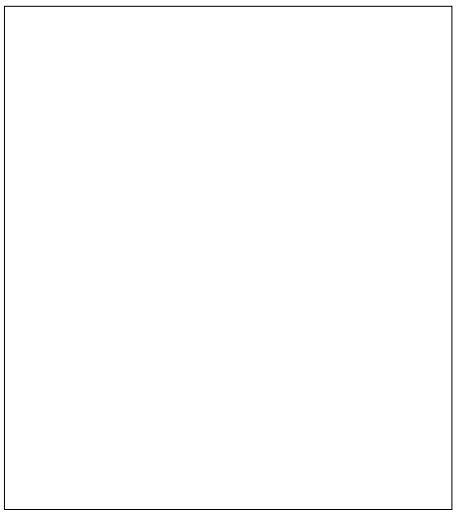
The collection of column-wise access patterns $P_A^{\mathtt{I}} = \{(2\mathtt{I} + \mathtt{J} - 2, \mathtt{I}) | 1 \leq \mathtt{J} \leq 2\}$ for $1 \leq \mathtt{I} \leq 4$, is $C_A^i = \{(t, i) | i \leq t \leq i + 4\}$ for $1 \leq i \leq 4$:

$$\begin{cases} 1 \leq x \leq 8 \\ 1 \leq y \leq 4 \\ 2 \leq x + y \leq 12 \\ 0 \leq x - y \leq 4 \end{cases}$$



Approximation of actual access patterns by the access pattern collection, which is uniquely determined by the values of $\vec{d}_A$ $\vec{l}$, and $\vec{u}$, yields a uniform and self-consistent description. However, some accuracy might be lost as shown in the previous example where longitudinal enveloping access patterns result. The collection of $P_A^{\mathtt{I}} = \{(4\mathtt{J} - 3, 2\mathtt{I} - 2\mathtt{J} + 5) | 1 \leq \mathtt{J} \leq 3\}$ for $1 \leq \mathtt{I} \leq 3$ with normalized direction, which is $C_A^i = \{(i - 2t, t) | \lceil \frac{i-9}{2} \rceil \leq t \leq \lfloor \frac{i-1}{2} \rfloor\}$ for $11 \leq i \leq 19$,

consists of too many points because not only longitudinal enveloping access patterns are considered, but also because **transversal** enveloping access patterns are included, e.g. for $i = 12$:



Clearly, there are $I_2 - I_1 + 1$ access patterns in C_A^i, with a total number of $\sum_{i=I_1}^{I_2}(T_2^i - T_1^i + 1)$ elements. For an arbitrary element a_{xy} in collection C_A^i, the value of i is determined as x, y, or $r_2 \cdot x - r_1 \cdot y$ for row-, column- and diagonal-wise access patterns respectively, so that it is stored in the k^{th} access pattern, where k is $i - l_1 + 1$, $i - l_2 + 1$, or $i - l_4 + 1$. Element a_{74}, for example, belongs to the 5^{th} access pattern in the previous example, since $i = 15$ and $l_4 = 11$. The value of t is determined as y, x, or $v_1 \cdot x + v_2 \cdot y$ for row- column- and diagonal-wise access patterns.

3.3 Complications

Although most occurrences of matrices in numerical programs fit the framework of section 2.1, the compiler must occasionally deal with complex subscripts (e.g. subscripted subscripts), or unknown loop bounds, as in the following fragments for an 100×100 matrix A:

```
DO I = 1, 10
  Z = ...
  ... A(Z,I)
ENDDO
```

```
DO I = 1, 50
  DO J = 1, 90
    ... A(PV(I),J)
  ENDDO
ENDDO
```

```
Z = ...
DO I = 10, Z
  DO J = 1, I
    ... A(5+I,J)
  ENDDO
ENDDO
```

If a complex subscript occurs, direction $\vec{d}_A$, and thus the access pattern collection, can only be determined if this subscript is *invariant* in one execution of the innermost controlling loop. Otherwise, only the *section* with e.g. $\vec{r} = (1, 1)$ can be determined, which is used in the computation of static inconsistences. Clearly, the direction is unknown for the first fragment, while $\vec{d}_A = (0, 1)$ holds in the second.

An approximation of the section can still be obtained, if constraints $1 \leq x \leq m$ and $1 \leq y \leq n$, that hold if subscript bounds are not violated, are used in those cases where compile-time analysis fails (cf. [3]). For complex subscripts, the new constraints are used directly for the *whole* subscript, so that, for instance, $u_3 = 190$ results for the second fragment since $x + y =$'PV(I)'+J $\leq$'PV(I)'+90. This yields collection $C_A^i = \{(i,t) | 1 \leq t \leq 90\}$ for $1 \leq i \leq 100$. Taking this simple approach for subscripts in which at least one loop index has (indirectly) unknown bounds yields, for example, $u_3 = 200$ and $l_4 = -99$ for the third fragment because both x and y contain such indices. However, in this case more accurate results are obtained if constraint $5 + \mathtt{I} \leq 100$ is used to derive an upper bound for I.[5] For example, $u_3 = 195$ is obtained by $x + y = 5 + \mathtt{I} + \mathtt{J} \leq 5 + 2\mathtt{I}$, while computation of $l_4 = 5$ does not require any bounds, because $x - y = 5 + \mathtt{I} - \mathtt{J} \geq 5 + \mathtt{I} - \mathtt{I}$. Collection $C_A^i = \{(i,t) | 1 \leq t \leq i - 5\}$ for $15 \leq i \leq 100$ results.

More accuracy is achieved by incorporation of advanced techniques. Induction variable recognition [1] can be used to replace some complex subscripts by linear functions of loop indices, while symbolic manipulations (cf. symbolic dependence testing [23]) improve computations on remaining complex subscripts.

3.4 Table Computation

Computation of the inconsistency table is closely related to the data dependence analysis problem, where static dependence between the statements of occurrences A_k and $A_{k'}$ is assumed if the sections of $C_{A_k}^i$ and $C_{A_{k'}}^i$ overlap. However, a static inconsistency between the access patterns of these occurrences is only recorded if also the sections *or* normalized directions differ. As a result of normalization, identical access patterns that are traversed in opposite directions are also considered consistent.

The intersection of two sections of form (3.2) with identical $\vec{r}$ is again a section of form (3.2), and is constructed by taking the most interior values for all section bounds (cf. [3]). Consequently, two sections described by $\vec{l}, \vec{u}$ and $\vec{l}', \vec{u}'$ respectively overlap, if this intersection is non-empty:

$$\bigwedge_{1 \leq i \leq 4} \max(\pi_i \cdot \vec{l}, \pi_i \cdot \vec{l}') \leq \min(\pi_i \cdot \vec{u}, \pi_i \cdot \vec{u}')$$

The sections are identical if $\vec{l} = \vec{l}'$, $\vec{u} = \vec{u}'$. The intersection of two sections with different $\vec{r}$ is not necessarily of form (3.2), and more computations are required to detect overlap. It should be noted that, since sections are a conservative approximation, overlap does not necessarily imply that the underlying diophantine equations ($F_{A_k}(\vec{\mathtt{I}}) = F_{A_{k'}}(\vec{\mathtt{I}}')$ for valid $\vec{\mathtt{I}}$ and $\vec{\mathtt{I}}'$) actually have a solution [4, 5, 16, 22, 28]. Consider, for example, code that computes the sum of the diagonal and off-diagonal parts of a square sparse matrix A:

[5] The compiler must also deal with a system of inequalities if maximum and minimum functions are used in lower and upper bounds, which increases the complexity of section computation.

```
DO I = 1, N
  DG = DG + A1(I,I)
  DO J = 1, I - 1
    LW = LW + A2(I,J)
    UP = UP + A3(J,I)
  ENDDO
ENDDO
```

A	d	c	r
r	0	0	0
c	0	0	
d	0		

Clearly, the sections of $C^i_{A_1}$, $C^i_{A_2}$ and $C^i_{A_3}$ are disjoint. For example, the sections of $C^i_{A_1}$ and $C^i_{A_2}$ do not overlap because $\max(0,1) > \min(0, \mathtt{N}-1)$:[6]

$$\left\{\begin{array}{c} 1 \le x \le N \\ 1 \le y \le N \\ 2 \le x+y \le 2N \\ 0 \le x-y \le 0 \end{array}\right. \leftrightarrow \begin{array}{c} 1 \le x \le N \\ 1 \le y \le N-1 \\ 2 \le x+y \le 2N-1 \\ 1 \le x-y \le N-1 \end{array}$$

If the following fragment also appears in the program, the inconsistency table changes because the section of $C^i_{A_4}$, that consists of the whole index set, overlaps with parts of all other sections:

```
DO I = 1, N
  DO J = 1, N
    A4(I,J) = A4(I,J) * 10.0
  ENDDO
ENDDO
```

A	d	c	r
r	1	1	1
c	0	0	
d	0		

Inconsistences between access patterns of two collections with *identical* normalized directions can be solved, if the collections are combined by minimal expansion of the sections into one section of form (3.2), i.e. the section described by bounds $\min(\pi_i \cdot \vec{l}, \pi_i \cdot \vec{l}')$ and $\max(\pi_i \cdot \vec{u}, \pi_i \cdot \vec{u}')$ for $1 \le i \le 4$ (cf. [3]) is considered. For example, combining $C^i_{A_2}$ and $C^i_{A_4}$ yields collection $C^i_{A_4}$ for *both* occurrences and eliminates the 'r-r' entry. However, since the section of A_2 is expanded, two new static inconsistencies arise:

A	d	c	r
r	2	2	0
c	0	0	
d	0		

Reshaping access pattern by application of conventional loop transformations, such as distribution, index set splitting, unrolling, and unimodular transformation [2, 6, 20, 23, 25, 27, 28, 29], can assist in eliminating inconsistencies between arbitrary access patterns. For example, the previous code is consistent after fragmentation of collection $C^i_{A_4}$ by a sequence of such transformations on the loop-nesting of $\mathtt{A}_4$:

```
DO I = 1, N
  DO J = 1, I - 1
    A4a(I,J) = A4a(I,J) * 10.0
  ENDDO
  A4b(I,I) = A4b(I,I) * 10.0
ENDDO
DO J = 1, N
  DO I = 1, J - 1
    A4c(I,J) = A4c(I,J) * 10.0
  ENDDO
ENDDO
```

$C^i_{A_{4_a}} = C^i_{A_2}$
$C^i_{A_{4_b}} = C^i_{A_1}$
$C^i_{A_{4_c}} = C^i_{A_3}$

[6] The peculiarity of section bounds $1 \le x$ and $2 \le x+y$ for $C^i_{A_2}$ arises because the execution of J is empty for I=1. Making these bounds more tight is desirable for comparisons.

A	d	c	r
r	0	0	0
c	0	0	
d	0		

If the compiler cannot eliminate all inconsistencies, constraints imposed by certain occurrences are favored over those of less relevant occurrences. This is done by combining the collections of inconsistent access patterns by minimal expansion of both sections into a new section of form (3.2) for *one* direction. This requires partial recomputation of the sections for occurrences with **ignored** direction, to account for another value of $\vec{r}$. Since encapsulation or expansion is disabled if storage is not done according to the normalized direction, the direction that belongs to occurrences with the potential of encapsulation or expansion in frequently executed statements is taken. This approach has as advantage that only *one* collection is associated with every occurrence, avoiding the need for run-time tests to determine in which of the data structures of overlapping collections an element might be stored. The major disadvantage, however, is that the presence of collections with large associated sections that cannot be fragmented, diminish the potential to account for peculiarities of the sparse matrix.

3.5 Storage Construction

Because some collections are combined or fragmented, a number of non-overlapping access pattern collections remains per sparse matrix A that belong to **representative occurrences** $\mathtt{A}_{k_1} \cdots \mathtt{A}_{k_r}$. The sections of these $C^i_{A_{k_j}}$ usually form a partition of the index set of the matrix, as illustrated in figure 3.

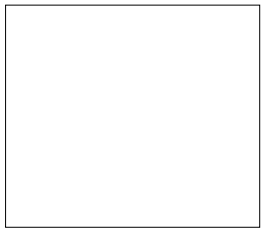


Figure 3: Partition

Per collection it is recorded whether dense or sparse storage of the access patterns is appropriate. All sparse collections are stored in one data structure (D1) or (L1), where per collection $\pi_1 \cdot \sigma_A^{-1}$ or $\pi_2 \cdot \sigma_A^{-1}$ values are stored for column-wise, or row- or diagonal-wise access pattern respectively. If *at least* one loop for which guard encapsulation will be applied requires an ordering as explained in section 2.2, this is also recorded for the corresponding collection. This requires insertion and deletion routines that keep the entries stored in monotonic *increasing* order on stored values, as a result of direction normalization. Because the access

patterns in one collection will be stored in one format, application of transformations is also useful to isolate access patterns with different properties.

One of the following declarations results for sparse storage of a matrix A. Sufficient working space is supplied for dynamic data structures. Parameter ρ is the density, u is the total number of access patterns (sum of $I_2 - I_1 + 1$ over all collections), v is the total number of *elements* along all access patterns (sum of $\sum_{i=I_1}^{I_2}(T_2^i - T_1^i + 1)$ over all collections), w is the maximum number of elements along one access pattern (maximum of $1 + \max_{I_1 \le i \le I_2}(T_2^i - T_1^i)$ over all collections), and x is some tunable constant:

```
      INTEGER   ANP, ASZ

(D1):PARAMETER (ANP = u,ASZ =
    +           ρ·v + 2·w + x)[7]
      REAL      AVAL(ASZ)
      INTEGER   AIND(ASZ), AHIGH(ANP),
    +           ALOW(ANP), ALAST


(L1):PARAMETER (ANP = u,ASZ =
    +           ρ·v + x)
      REAL      AVAL(ASZ)
      INTEGER   AIND(ASZ), ALINK(ASZ),
    +           AFIRST(ANP), AFREE
```

Scalar `ALAST` points to the last used element in (D1) to support extra data movement, while `AFREE` is a pointer to a linked list in (L1) that contains all free elements. Per collection $C^i_{A_{k_j}}$, an initially zero offset is recorded, and increased with I_2+I_1+1 after consideration of each collection, such that the k^{th} access pattern in a collection starts at location `ALOW(D)` or `AFIRST(D)` in the data structure, where `D`$=\mathit{offset}_{A_{k_j}}+k$. For every access pattern collection $C^i_{A_{k_j}}$ that requires dense storage, the following declaration is generated, where a unique label $\mathit{lab}_{A_{k_j}}$ is used per collection and $m = \max_{I_1 \le i \le I_2}(T_2^i - T_1^i)$. Because the maximum length is taken, this format is not very well suited for e.g. triangular shaped dense access patterns. If $I_2 - I_1 = 0$ holds, a one-dimensional array is used.

$$\texttt{REAL ADNS}\mathit{lab}_{A_{k_j}}(1+m, I_2 - I_1 + 1)$$

4 Code Generation

First, compiler-supplied primitives are presented, followed by a discussion of code generation for specific program constructs and sparse occurrences.

4.1 Primitive Operations

The definition of certain primitive operations for data structures (D1) and (L1) is supplied by the compiler, so that these primitives can be used in the generated code. The following table lists some basic operations with a short description, where '_' can be `D1` or `L1`:

`INT_()`	Initialization of data structure
`LKP_()`	Lookup of location of an entry ($\perp$ otherwise)
`INS_()`	Insertion of an entry
`DEL_()`	Deletion of entry at a location

Location $\perp$ cannot be used as storage in a data structure for A, but it has the property that `AVAL(`$\perp$`)=0.0`. Insertions and deletions are fast, provided that data movement is not required in (D1). However, no ordering is maintained on the entries. Therefore, the following primitives are also supported, where `ODELL1` is equal to `DELL1` since data movement does not occur for data structure (L1):

`OINS_()`	Ordered insertion of an entry
`ODEL_()`	Ordered deletion of an entry

Two operations are supplied to support expansion of an access pattern:

`SCAT_()`	Expansion into dense
`GATH_()`	Compression into sparse

A detailed description and definitions of these primitives are given in [9].

4.2 Encapsulated Guards

If guard encapsulation in the execution set of a surrounding $\texttt{I}_c$-loop is feasible and cancellation or creation along the (enveloping) access pattern cannot occur, code for the following conversion is generated:

```
DO Ic ∈ Vc
  IF (FA(I) ∈ EA) THEN
    ... A(FA(I)) ...
  ENDIF
ENDDO   ↓
 DO AD ∈ PAD_A^{I1,...,Ic-1}
   ... A'[AD] ...
 ENDDO
```

Addresses in $\overline{PAD}_A^{\texttt{I}_1,\ldots,\texttt{I}_{c-1}}$ that correspond to an access pattern in a sparse collection $C^i_{A_{k_j}}$ are found through $\texttt{D} = \mathit{offset}_{A_{k_j}} + k$ in the data structure, where the value of k is determined *independently* of the value of $\texttt{I}_c$ before the resulting loop as shown below, where $R = m_{10} + m_{11}\texttt{I}_1 + \ldots + m_{1c-1}\texttt{I}_{c-1}$ and $C = m_{20} + m_{21}\texttt{I}_1 + \ldots, + m_{2c-1}\texttt{I}_{c-1}$ (see section 3.2):

	k	i
r	$i - l_1 + 1$	R
c	$i - l_2 + 1$	C
d	$i - l_4 + 1$	$r_2 \cdot R - r_1 \cdot C$

One of the following frameworks is generated:

```
(D1): DO AD = ALOW(D), AHIGH(D)
        ...
      ENDDO
(L1): AD = AFIRST(D)
      DO WHILE (AD ≠ ⊥)
        ...
        AD = ALINK(AD)
      ENDDO
```

[7] The size of *extra* working space in (D1) to prevent the need for frequent garbage collection is inspired on [12, 15].

Within the loop body, the value of $\mathtt{I}_c$ is restored and inclusion in V_c is tested. The resulting code for column-wise access patterns in shown below, where execution set $V_c = \{\mathtt{L}^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}}, \ldots, \mathtt{U}^{\mathtt{I}_1,\ldots,\mathtt{I}_{c-1}}\}$:

```
 I_c = AIND(AD) - m_10 - m_11 * I_1
+ ... - m_1c-1 * I_c-1
 IF (MOD(I_c, m_1c) = 0) THEN
   I_c = I_c / m_1c
   IF (L^I_1,...,I_c-1 ≤ I_c)
+    AND (I_c ≤ U^I_1,...,I_c-1) THEN
     ...
   ENDIF
 ENDIF
```

In this construct, the original loop body is generated in which most occurrences $\mathtt{A}(F_A(\vec{\mathtt{I}}))$ that correspond to the encapsulated guard are replaced by `AVAL(AD)` (details are given in section 4.5). If $m_{1c} = 1$, the `MOD` and integer division operations are not required, while a compare is eliminated if the compare value corresponds to the value of the index bound of the stored access patterns. If $\mathtt{I}_c$ is not used in the tests or loop body, this computation is omitted.

If the execution order must be preserved, an ordering on the entries will be maintained, and the same constructs can be used, with a negative stride for (D1) or preceding list reversal for (L1) if the direction in storage is reversed by normalization.[8] If forward or backward creation, or inward cancellation might occur, the same construct is used for (L1), in which computation `AD=ALINK(AD)` is performed *before* inward cancellation. For (D1) the following framework accounts for changes in the address set:

```
AD = ALOW(D)
DO WHILE (AD ≤ AHIGH(D))
  ...
  AD = AD + 1
ENDDO
```

Since in a WHILE-loop the condition is evaluated in every iteration, it might be useful to hoist an expensive computation of `D` out this loop. Additionally, adjustments to `AD` are made for one of the following events, while `AD` is decremented after inward cancellation:

forward creation	backward creation
AD=AD-ALOW(D)	AD=AD-ALOW(D)
...	...
AD=AD+ALOW(D)	AD=AD+ALOW(D)+1

After backward creation, the offset to the base address has to be incremented, as is illustrated in figure 4 for insertion of a_{32} in row-wise storage during operation on a_{33}, where extra data movement occurs.

Insertions in *other* access patterns of this collection that might cause garbage collection are correctly handled if the construct for forward creation is used, because this preserves the offset to the possibly changed base address. If several guards are encapsulated in the execution sets of loops in one nesting, different variables `AD`*lab*, declared as INTEGER, are used in the resulting nested constructs.

[8] For both data structures, we can also use a WHILE-loop that scans entries until the bounds are exceeded in order to limit the number of entries examined. Storing access patterns according to most frequent occurring directions reduces overhead for (L1), but might require routines that maintain a monotonic *decreasing* ordering on $\pi_i \cdot \sigma_A^{-1}$ values.

Figure 4: Extra Data Movement

4.3 Dense Storage

An occurrence $\mathtt{A}(F_A(\vec{\mathtt{I}}))$ in the program that is contained in the section of a dense and consistent collection $C^i_{A_{k_j}}$ is replaced by the corresponding dense data structure '`ADNS`$lab_{A_{k_j}}$`(E,D)`', where `D` is determined as shown in the previous section for a zero offset. For index `E`, the normalized value of t is taken, i.e. $v_1 \cdot f_1(\vec{\mathtt{I}}) + v_2 \cdot f_2(\vec{\mathtt{I}}) - T^i_1 + 1$, where i is determined as for `D`. For example, if dense storage is selected for collection $C^i_A = \{(t,i) | i \le t \le i+4\}$ for $1 \le i \le 4$ (cf. section 3.2), the following conversion is applied on the next loop, since $i = \mathtt{I}$:

```
DO I = 1, 4                    DO I = 1, 4
  DO J = 1, 2                    DO J = 1, 2
    B(I,J) = A(2*I+J-2,I)          B(I,J) = A(2*I+J-2,I)
  ENDDO                          ENDDO
ENDDO                          ENDDO
```

The trapezoidal part of the matrix is stored in rectangular storage, i.e. access patterns are skewed in storage. This also occurs for the dense storage of collection $C^i_A = \{(t, t-i) | \max(1, 1+i) \le t \le \min(\mathtt{M}, \mathtt{N}+i)\}$ for $-b_2 \le i \le b_1$ in the following fragment on a band matrix A (cf. section 2.2.3), since $i = -\mathtt{J}$:

```
DO J = -b_1, b_2                DO J = -b_1, b_2
  DO I = MAX(1,1-J), MIN(M,N-J)   DO I = MAX(1,1-J), MIN(M,N-J)
    Y(I) = Y(I) + A(I,J+I) * X(J+I)   Y(I) = Y(I) + A(I,J+I) * X(J+I)
  ENDDO                           ENDDO
ENDDO                           ENDDO
```

Loop invariant computations in `D` and `E` can be hoisted out the loop. In the resulting storage scheme, all access patterns are up-justified, as is illustrated below for a 4×4 band matrix A with $b_1 = b_2 = 1$:

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 \\ 0 & a_{32} & a_{33} & a_{34} \\ 0 & 0 & a_{43} & a_{44} \end{pmatrix} \quad \begin{array}{|c|c|c|} \hline a_{12} & a_{11} & a_{21} \\ a_{23} & a_{22} & a_{32} \\ a_{34} & a_{33} & a_{43} \\ - & a_{44} & - \\ \hline \end{array}$$

However, in this case it is also valid to use just *one* of the bounds that is derived from the section bounds without increasing storage requirements, i.e.

only $1+i \leq t$ or $1 \leq t$ is considered. This automatically yields more sophisticated storage schemes with regular properties [13, 24], since `E=I+J` or `E=I`:

-	a_{11}	a_{21}
a_{12}	a_{22}	a_{32}
a_{23}	a_{33}	a_{43}
a_{34}	a_{44}	-

$1+i \leq t$

a_{12}	a_{11}	-
a_{23}	a_{22}	a_{21}
a_{34}	a_{33}	a_{32}
-	a_{44}	a_{43}

$1 \leq t$

If transversal enveloping access patterns occur in the collection, or directions have been normalized, this is correctly accounted for in `D` and `E`, as is illustrated with the following conversion for dense storage of $C_A^i = \{(i-2t,t) | \lceil \frac{i-9}{2} \rceil \leq t \leq \lfloor \frac{i-1}{2} \rfloor\}$ for $11 \leq i \leq 19$, since $i =$ `4I+7` (cf. section 3.2):

```
DO I = 1, 3                          DO I = 1, 3
  DO J = 1, 3                          DO J = 1, 3
    B(I,J) = A(4*J-3,2*I-2*J+5)          B(I,J) = A(4*J-3,2*I-2*J+5)
  ENDDO                                ENDDO
ENDDO                                ENDDO
```

If the direction belonging to an occurrence that is contained in the section of a dense collection $C^i_{A_{k_j}}$ has been ignored, the value of `D` and `E` are both variant in the innermost loop because i is determined as $f_1(\vec{\mathtt{I}})$, $f_2(\vec{\mathtt{I}})$, or $r_1 \cdot f_1(\vec{\mathtt{I}}) - r_2 \cdot f_2(\vec{\mathtt{I}})$ for row-, column- and diagonal-wise *stored* access patterns respectively. For instance, if the previous up-justified diagonal-wise storage is used, the following conversion is applied on a fragment with row-wise access of this band, since $i =$ `I-J`:

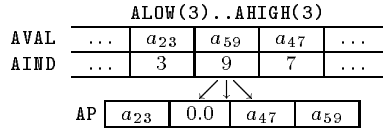
```
DO I = 1, 4                          DO I = 1, 4
  DO J = MAX(I-1,1), MIN(I+1,4)        DO J = MAX(I-1,1), MIN(I+1,4)
    Y(I) = Y(I) + A(I,J) * X(J)          Y(I) = Y(I) + A(I,J) * X(J)
  ENDDO                                ENDDO
ENDDO                                ENDDO
```

4.4 Expansion

Expansion of an access pattern in a sparse collection $C^i_{A_{k_j}}$, found through `D`, into a dense vector `AP` of appropriate length in which all elements are initialized to zero, is performed with a scatter-operation. The value of each entry in the sparse data structure is assigned to the corresponding element in `AP` at index `E`. During the scatter, each `E`$^{\text{th}}$ bit in an initially reset bit-vector `BIT` is also set to support a so-called switch [24], which provides information about the nonzero structure. Index `E` is computed as $\frac{val+o}{f}$ for integers f and o:

	val	f	o
r	y	1	$1 - T_1^i$
c	x	1	$1 - T_1^i$
d	y	r_2	$v_1 \cdot i - r_2(1 - T_1^i)$

These integers are passed as parameters to the scatter and gather subroutines, to construct index `E` from the stored row or column indices. Expansion of access pattern $P_A^{\mathtt{I},\mathtt{J}} = \{(\mathtt{I}+\mathtt{K}, \mathtt{J}+2\mathtt{K}) | 1 \leq \mathtt{K} \leq 3\}$ for `I=1` and `J=1`, for example, where storage is done according to the first collection in section 3.2, is performed as shown below, since `D=3`, $o = -1$ and $f = 2$:



Within the scope of this expansion, all operations are performed on the dense representation `AP`. A right-hand side occurrence `A`$(F_A(\vec{\mathtt{I}}))$ along the expanded access pattern is replaced by `AP(E)`, while the following constructs are used for a left-hand side occurrence, depending on the value of the associated guard and right-hand side expression:

	nonzero r.h.s.
$F_A(\vec{\mathtt{I}}) \in E_A$	`AP(E) = ...`

$F_A(\vec{\mathtt{I}}) \notin E_A$	`CALL INS_(...)`
	`BIT(E) = 1`
	`AP(E) = ...`

	zero r.h.s

$F_A(\vec{\mathtt{I}}) \in E_A$	`AD = LKP_(...)`
	`CALL DEL_(...)`
	`BIT(E) = 0`
	`AP(E) = 0.0`

$F_A(\vec{\mathtt{I}}) \notin E_A$	–

Guard '$F_A(\vec{\mathtt{I}}) \in E_A$' is evaluated without lookup overhead by bit-vector test '`BIT(E)=1`' (details of code generation are given in the next section). Creation or cancellation are directly accounted if required. Because each deletion (or *ordered* insertion) requires a preceding lookup, expansions become more useful if this does not occur frequently. After all operations have been performed on this access pattern, the actual values are stored back with a gather operation. Used elements of `AP` and `BIT` are reset during the gather operation to support next expansions.[9]

For example, expansion of each access pattern of A results in the following conversion if sparse matrices A and B are stored in row-wise (D1) data structures, so that `D=I`, $f = 1$ and $o = 0$. Since guard '$(\mathtt{I},\mathtt{J}) \in E_B$' has been encapsulated, the right-hand side expression is nonzero in every iteration, and the appropriate constructs are used:

```
DO I = 1, M                          DO I = 1, M
  DO J = 1, N                          CALL SCATD1(AVAL,AIND,ALOW(I),
    A(I,J) = A(I,J) + B(I,J)         +   AHIGH(I),AP,BIT,1,0)
  ENDDO                                DO AD = BLOW(I), BHIGH(I)
ENDDO                                   IF (BIT(J) = 0) THEN
                                          CALL INSD1(AVAL,AIND,ALOW,
                                     +      AHIGH,I,ANP,ASZ,ALST,J,.0)
                                        BIT(J) = 1
                                      ENDIF
                                      AP(J) = AP(J) + BVAL(AD)
                                    ENDDO
                                    CALL GATHD1(AVAL,AIND,ALOW(I),
                                   +   AHIGH(I),AP,BIT,1,0)
                                  ENDDO
```

[9] Resetting `BIT` is not necessary if the multiple switch technique [19, 24] is used with the value of `D`, but increases the storage requirements of vector `BIT`.

The more operations are performed on an expanded access pattern, the less scatter and gather overhead dominates. Insertions are performed to obtain storage only and, therefore, do not require actual values. If assignments to `AP` occur, all operations along the expanded access pattern must be done on `AP`, since it contains the most recent value. To avoid run-time tests for inclusion in an expanded access pattern, expansion of an access pattern in a sparse collection is only performed if there are no occurrences in the loop-body with ignored direction and the same collection.

Expansions for which `E` equals the stored column or row indices because $f = 1$ and $o = 0$, can be done with sparse BLAS subroutines [11] if available, although the switch has to be handled separately in that case. For example, a scatter and gather that resets used elements of `AP` in sparse BLAS for (D1) are shown below:

```
 CALL SSCTR(AHIGH(D)-ALOW(D)+1,
+   AVAL(ALOW(D)),AIND(ALOW(D)),AP)
  ...
 CALL SGTHRZ(AHIGH(D)-ALOW(D)+1,AP,
+   AVAL(ALOW(D)),AIND(ALOW(D)))
```

4.5 Sparse Occurrences

In this section we present a 'condition-driven' code generation method for the occurrences of sparse matrices in a statement. Most explanation is done for (D1) because conversion into (L1) is straightforward. The approach is based on the fact that guard '$F_A(\vec{\mathtt{I}}) \in E_A$' is 'true' for an occurrence with encapsulated guard or in dense storage, is evaluated with test '`BIT(E)=1`' for an occurrence along an expanded access pattern, or is evaluated as follows otherwise:

```
AD = LKPD1(AIND,ALOW(D),AHIGH(D),E)
IF (AD ≠ ⊥) ...
```

Index `E` and `D` are determined as explained earlier. The σ_A-lookup is obtained as 'side-effect' of guard evaluation if the test succeeds. An occurrence at the right-hand side is replaced by `AVAL(AD)`. For a left-hand side occurrence, the following constructs are required, depending on the value of its guard and right-hand side expression:[10]

	nonzero r.h.s.
$F_A(\vec{\mathtt{I}}) \in E_A$	`AVAL(AD) = ...`
$F_A(\vec{\mathtt{I}}) \notin E_A$	`CALL INS_(...)`
	zero r.h.s.
$F_A(\vec{\mathtt{I}}) \in E_A$	`CALL DEL_(...)`
$F_A(\vec{\mathtt{I}}) \notin E_A$	–

First, two variables χ and ψ are set to the conditions associated with the statement under consideration and right-hand side expression respectively (cf. section 2.1). All occurrences that are contained in the section of a dense collection are replaced as explained in section 4.3. Right-hand side occurrences with encapsulated guards are replaced by `AVAL(AD)`, while code generation for left-hand side occurrences is deferred. Corresponding guards in χ and ψ changed into 'true'.

Subsequently, evaluation code is generated for each occurrence $\mathtt{A}(F_A(\vec{\mathtt{I}}))$ where guard '$F_A(\vec{\mathtt{I}}) \in E_A$' dominates χ. If guard hoisting is valid (see section 2.2.1) this evaluation is generated at higher level, so that the test becomes more useful. A right-hand side occurrence is replaced by `AVAL(AD)` or `AP(E)`, while code generation is deferred again otherwise. Conditions χ and ψ are adapted accordingly.

Evaluation code for all disjunctions in χ is generated next, while ψ is adapted accordingly. Right-hand side occurrences are replaced by `AP(E)` or `AVAL(AD)`, which is valid since $\mathtt{AVAL}(\bot) = \mathtt{0.0}$ holds. For example, the following code results for '`X=X+A(I,J)+B(J,K)`', if occurrence `B` is along an expanded access pattern, because $\chi =$'$(\mathtt{I},\mathtt{J}) \in E_A \vee (\mathtt{J},\mathtt{K}) \in E_B$' holds:

```
AD = LKPD1(AIND,ALOW(D1),AHIGH(D1),E1)
IF (AD ≠ ⊥ ∨ BIT(E2)=1) THEN
  X = X + AVAL(AD) + AP(E2)
ENDIF
```

If there is a left-hand side sparse occurrence, further evaluation code on remaining guards in ψ is generated to differentiate between the cases $\psi =$'true' and $\psi =$'false' respectively. All remaining right-hand side occurrences are replaced by `AP(E)` or by:

```
AVAL(LKPD1(AIND,ALOW(D),AHIGH(D),E))
```
(4.5)

Finally, code for a possible left-hand side occurrence $\mathtt{A}(F_A(\vec{\mathtt{I}}))$ is generated. If the value of guard '$F_A(\vec{\mathtt{I}}) \in E_A$' is not known in the current scope, differentiating evaluation code is generated. The value of the right-hand side expression is determined from the value of ψ in the current scope. Corresponding constructs are taken from the previous tables.

For example, the following code results for statement '`A(I,J)=X+A(I,J)*B(J,I)`', because conditions χ and ψ are both 'true':

```
 AD = LKPD1(AIND,ALOW(D),AHIGH(D),E)
 EXPR = X + AVAL(AD) * BVAL(
+  LKPD1(BIND,BLOW(D1),BHIGH(D1),E1))
 IF (AD ≠ ⊥) THEN
   AVAL(AD) = EXPR
 ELSE
   CALL INSD1(AIND,ALOW,AHIGH,D,
+             ANP,ASZ,ALST,E,EXPR)
 ENDIF
```

The address of construct (4.5) is saved, since it can be used by the following evaluation code. Identical parts in both branches are computed in variable `EXPR` before the IF-statement to prevent code duplication. Similarly, identical parts in the constructs for an expanded access pattern can be placed after the IF-statement, as was done in section 4.4.

Statement '`A(I,J)=0.0`' is converted as shown below, since $\chi =$'$(\mathtt{I},\mathtt{J}) \in E_A$' and $\psi =$'false':

[10] To support fast deletion in `DELL1`, function `LKPL1` also supplies a pointer to an eventual *previous* entry. For an encapsulated guard, this pointer is maintained during the WHILE-loop.

```
AD = LKPD1(AIND,ALOW(D),AHIGH(D),E)
IF (AD ≠ ⊥) THEN
  CALL DELD1(AIND,ALOW(D),
             AHIGH(D),ALST,AD)
ENDIF
```

For statement '`A(I,J)=A(I,J)*B(I,K)`', code generation proceeds as follows, since χ='(I, J) $\in E_A$' and ψ='(I, J) $\in E_A \wedge$ (I, K) $\in E_B$' hold. First, evaluation code for the dominating guard is generated. Since ψ='(I, K) $\in E_B$' holds afterwards, and a sparse occurrence appears at the left-hand side, further distinction on B is made. Finally, the constructs for a nonzero and zero right-hand side expression that correspond to '(I, J) $\in E_A$' are generated:

```
 AD1 = LKPD1(AIND,ALOW(D1),
+            AHIGH(D1),E1)
 IF (AD1 ≠ ⊥) THEN
   AD2 = LKPD1(BIND,BLOW(D2),
+              BHIGH(D2),E2)
   IF (AD2 ≠ ⊥) THEN
     AVAL(AD1) = AVAL(AD1) * BVAL(AD2)
   ELSE
     CALL DELD1(AIND,ALOW(D1),
+               AHIGH(D1),ALST,AD1)
   ENDIF
 ENDIF
```

A complication arises if the guard of the left-hand side occurrence is involved in a disjunction in condition χ, since the exact value of this guard must be known to generate appropriate constructs. Therefore, evaluation code for that guard is generated first. In statement '`A(I,J)=A(I,J)*2.0+B(K,J)`', for example, χ='(I, J) $\in E_A \vee$ (K, J) $\in E_B$' holds. Consequently, the following IF-statement results, differentiating between the cases χ ='true' and χ ='(K, J) $\in E_B$'. Further evaluation of χ results in the ELSE-branch, followed by an insertion because ψ ='true' holds:

```
 AD1 = LKPD1(AIND,ALOW(D1),
+            AHIGH(D1),E1)
 IF (AD1 ≠ ⊥) THEN
   AVAL(AD1) = AVAL(AD1) * 2.0 +
+   BVAL(LKPD1(BIND,BLOW(D2),
+              BHIGH(D2),E2))
 ELSE
   AD2 = LKPD1(BIND,BLOW(D2),
+              BHIGH(D2),E2)
   IF (AD2 ≠ ⊥) THEN
     CALL INSD1(AIND,ALOW,AHIGH,D1,
+      ANP,ASZ,ALST,E1,BVAL(AD2))
   ENDIF
 ENDIF
```

4.6 Storage Initialization

The most general and flexible initialization method is from-file construction, since it can be used for arbitrary sparse matrices. In order to keep the input storage scheme simple, coordinate scheme storage is used, where each file consists of an integer nz, followed by nz triples (i, j, a_{ij}) [13, 14, 15, 18, 30]. In [9], it is discussed how the compiler can generate appropriate initializing routines.

5 Examples

Consider code generation for the first fragment of section 3.4, where the diagonal of a sparse matrix A is dense. For $C^i_{A_1}$, data structure `ADNS(N)` is used (note that $I_2 - I_1 = 0$ holds for this collection), while the row- and column-wise access patterns of $C^i_{A_2}$ and $C^i_{A_3}$ are stored in `AVAL`. After distribution of the `J`-loop, guard encapsulation of '(I, J) $\in E_A$' and '(J, I) $\in E_A$' becomes feasible, and the following code results for (D1):

```
DO I = 1, N
  DG = DG + ADNS(I)
  DO AD = ALOW(I), AHIGH(I)
    LW = LW + AVAL(AD)
  ENDDO
  DO AD = ALOW(N+I), AHIGH(N+I)
    UP = UP + AVAL(AD)
  ENDDO
ENDDO
```

Consider a 4×4 matrix, with zero elements a_{31}, a_{42}, a_{24}, and $a_{34} = 0$. Possibly resulting contents of data structure (D1) and `ADNS` for this matrix are shown in figure 5, where array `AHIGH` has been omitted for clarity. Row or column indices are stored for column- and row-wise access patterns respectively. No ordering is required on the entries.

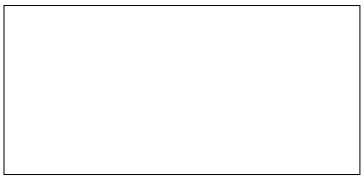


Figure 5: Data Structure (D1)

However, if many other occurrences in the program have row-wise access patterns, row-wise storage is selected for the whole matrix. If all attempts to reshape access patterns fail, the following code results. Usage of preceding tests for the first and last statement probably does not gain much execution time:

```
DO I = 1, N
 AD = LKPD1(AIND,ALOW(I),AHIGH(I),I)
 IF (AD ≠ ⊥) DG = DG + AVAL(AD)
 DO AD = ALOW(I), AHIGH(I)
  J = AIND(AD)
  IF (J ≤ I-1) LW = LW + AVAL(AD)
 ENDDO
 DO J = 1, I - 1
  AD = LKPD1(AIND,ALOW(J),AHIGH(J),I)
  IF (AD ≠ ⊥) UP = UP + AVAL(AD)
 ENDDO
ENDDO
```

Consider outer product code for matrix multiplication $C \leftarrow C + A \cdot B$, where matrices A, B and C are in fact sparse [13, 19, 21, 24]:

```
DO K = 1, N
 DO I = 1, M
  DO J = 1, L
    C(I,J) = C(I,J) + A(I,K) * B(K,J)
  ENDDO
 ENDDO
ENDDO
```

Condition '$(\mathtt{I},\mathtt{K}) \in E_A \wedge (\mathtt{K},\mathtt{J}) \in E_B$' is associated with the assignment statement. Clearly, guard '$(\mathtt{K},\mathtt{J}) \in E_B$' can be encapsulated in the execution set of the J-loop. However, because inward cancellation does not occur for A, the guard '$(\mathtt{I},\mathtt{K}) \in E_A$' can be hoisted and encapsulated in the execution set of the I-loop. Additionally, to avoid expensive lookups for matrix C, each row of this matrix is expanded before operated on. Selecting column-wise storage for A and row-wise storage for B and C results in the following code for (D1):

```
 DO K = 1, N
   DO AD1 = ALOW(K), AHIGH(K)
     I = AIND(AD1)
     CALL SCATD1(CVAL,CIND,CLOW(I),
+                CHIGH(I),AP,BIT,1,0)
     DO AD2 = BLOW(K), BHIGH(K)
       J = BIND(AD2)
       IF (BIT(J) = 0) THEN
         CALL INSD1(CVAL,CIND,CLOW,
+          CHIGH,I,CNP,CSZ,CLST,J,0.0)
         BIT(J) = 1
       ENDIF
       AP(J) = AP(J) + AVAL(AD1)
+              * BVAL(AD2)
     ENDDO
     CALL GATHD1(CVAL,CIND,CLOW(I),
+                CHIGH(I),AP,BIT,1,0)
   ENDDO
 ENDDO
```

Similarly, two guard encapsulations are feasible for the middle product JKI and inner product IKJ if both matrices are stored column- or row-wise respectively, and for middle product JIK if A and B are stored row- and column-wise. However, in inner product IJK operations on zeros are likely to occur, since a conjunction cannot be encapsulated.

In [8] several version for LU-factorization for (D1) were presented under the assumption that garbage collection did not occur. Here, we present a version for row-wise storage that correctly accounts for garbage collection, possibly caused by insertions in the $\mathtt{J}^{\text{th}}$ row. Because ψ='true' holds after code generation for condition χ='$(\mathtt{J},\mathtt{I}) \in E_A \wedge (\mathtt{I},\mathtt{K}) \in E_A$' of S_2, only a nonzero right-hand side construct is generated.

```
DO J = 2, N
  DO I = 1, J - 1
S1 : A(J,I) = A(J,I) / A(I,I)
    DO K = I + 1, N
S2 :   A(J,K) = A(J,K) - A(J,I) * A(I,K)
    ENDDO
  ENDDO
ENDDO
```

↓

```
DO J = 2, N
 CALL SCATD1(AVAL,AIND,ALOW(J),
             AHIGH(J),AP,BIT)
 DO I = 1, J - 1
  IF (BIT(I) ≠ 0) THEN
   AP(I) = AP(I) / AVAL(
+    LKPD1(AIND,ALOW(I),AHIGH(I),I))
   KA = ALOW(I)
   DO WHILE (KA ≤ AHIGH(I))
     K = AIND(KA)
     IF (I+1 ≤ K) THEN
       IF (BIT(K) = 0) THEN
         KA = KA - ALOW(I)
         CALL INSD1(AVAL,AIND,ALOW,
+          AHIGH,J,ANP,ASZ,ALST,K,0.0)
         KA = KA + ALOW(I)
         BIT(K) = 1
       ENDIF
       AP(K) = AP(K)-AP(I)*AVAL(KA)
     ENDIF
     KA = KA + 1
   ENDDO
  ENDIF
 ENDDO
 CALL GATHD1(AVAL,AIND,ALOW(J),
             AHIGH(J),AP,BIT)
ENDDO
```

6 Future Research

More research is necessary into strategies for solving inconsistencies, automatic analysis of nonzero structures, and problems that are transparent in the original dense code, such as the reduction of creation (fill-in). Since in practical implementations user-defined subroutines appear, problems that arise if sparse matrices are passed as parameter in the original code must also be dealt with.

Acknowledgements The authors would like to thank Arnold Niessen for his comments.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers Principles, Techniques and Tools.* Addison-Wesley publishing company, 1986.

[2] 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.

[3] Vasanth Balasundaram. *Interactive Parallelization of Numerical Scientific Programs.* PhD thesis, Department of Computer Science, Rice University, 1989.

[4] Vasanth Balasundaram. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *Journal of Parallel and Distributed Computing*, Volume 9:154–170, 1990.

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

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

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

[8] 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.

[9] Aart J.C. Bik and Harry A.G. Wijshoff. A sparse compiler. Technical Report no. 93-04, Dept. of Computer Science, Leiden University, 1993.

[10] A.R. Curtis and J.K. Reid. The solution of large sparse unsymmetric systems of linear equations. *Journal Inst. Maths. Applics.*, Volume 8:344–353, 1971.

[11] David S. Dodson, Roger G. Grimes, and John G. Lewis. Sparse extensions to the fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, Volume 17:253–263, 1991.

[12] I.S. Duff. Data structures, algorithms and software for sparse matrices. In David J. Evans, editor, *Sparsity and Its Applications*, pages 1–29. Cambridge University Press, 1985.

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

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

[15] I.S. Duff and J.K. Reid. Some design features of a sparse matrix code. *ACM Transactions on Mathematical Software*, pages 18–35, 1979.

[16] C. Eisenbeis, O. Temam, and H. Wijshoff. On efficiently characterizing solutions of linear diophantine equations and its application to data dependence analysis. In *Proceedings of the Seventh International Symposium on Computer and Information Sciences*, 1992.

[17] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[18] Alan George and Joseph W. Liu. The design of a user interface for a sparse matrix package. *ACM Transactions on Mathematical Software*, Volume 5:139–162, 1979.

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

[20] David J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, New York, 1978. Volume 1.

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

[22] Samuel P. Midkiff. *The Dependence Analysis and Synchronization of Parallel Programs.* PhD thesis, C.S.R.D., 1993.

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

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

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

[26] Harry A.G. Wijshoff. Implementing sparse blas primitives on concurrent/vector processors: a case study. Technical Report no. 843, Center for Supercomputing Research and Development, University of Illinios, 1989.

[27] 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.

[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.