Compilation Techniques for Sparse Matrix 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
ajcbik@cs.leidenuniv.nl and harryw@cs.leidenuniv.nl

Abstract

The problem of compiler optimization of sparse codes is well known and no satisfactory solutions have been found yet. One of the major obstacles is formed by the fact that sparse programs deal explicitly with the particular data structures selected for storing sparse matrices. This explicit data structure handling obscures the functionality of a code to such a degree that the optimization of the code is prohibited, e.g. by the introduction of indirect addressing. The method presented in this paper postpones data structure selection until the compile phase, thereby allowing the compiler to combine code optimization with explicit data structure selection. Not only enables this method the compiler to generate efficient code for sparse computations, also the task of the programmer is greatly reduced in complexity.

Index Terms: Compilation Techniques, Optimization, Program Transformations, Restructuring Compilers, Sparse Computations, Sparse Matrices.

1 Introduction

A significant part of scientific codes consists of sparse matrix computations which show notoriously bad efficiency on today's supercomputers. Mostly only a small fraction of the computing power of these computers can be utilized. There are many reasons for this. First, sparse matrix computations induce irregular data access which can significantly reduce memory bandwidth and cache utilization, e.g. spatial locality cannot always be exploited as efficiently. Secondly, the amount of possible reuse of data (temporal locality), is limited due to the fact that many operations are nullified making the ratio computations performed over the size of data sets very small. Not only prevents this data locality optimizations, also the communication overhead when these codes are executed on distributed memory computers can be substantial.

The third problem is caused by the fact that sparse matrices need to be represented in a compact way so that the *storage requirements* and *computational time* are kept to reasonable levels. This causes the representation of a sparse code in either FORTRAN with the occurrence of indirect addressing or in another language with pointer structures to be very distorted. This is probably the most important problem of the ones listed above, because it does not only complicate software maintenance and the effort of producing sparse computation codes, but also most compiler optimizations get disabled.

The latter problem has been recognized for a long time now and there have been many efforts to overcome this problem. Most of the methods proposed in the past rely on pre–evaluation of the index sets of indirect addressed loops [8], possibly followed by reordering of the index set [9]. A problem of these methods is that the data structure as selected by the programmer cannot be changed. As different architectural features might favor different data structures for sparse matrices, the code will stay inefficient regardless of the optimizations performed if the sparse code is not specifically targeted for the architectural features of a particular machine. So, in effect, to overcome the problem of sparse matrix computations the compiler should not only be able to perform program transformations but also transformations on data structures themselves.

In order to tackle this sparse data structure problem, examination of the following generic definition is useful: *sparse computations are computations that compute on sparse data structures* and *sparse data structures (i.e. sparse matrices) are data structures that are logically contained in enveloping data structures (i.e. dense matrices).* The underlying problem for sparse computations now is *where* to deal with the fact that

*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 [2].

only parts of the enveloping data structures are computed on. The common approach is to deal with this at the programming level. However, it is also possible to deal with this issue at a lower level, i.e. at the compilation level.

This implies that the computation is defined on enveloping data structures, i.e. dense matrices, and that the compiler transforms the code to execute on sparse data structures. This has the advantage that the compiler does not need to extract program knowledge from an obscured code, but is presented with a much cleaner program on which regular data dependence checking and standard optimizations can be performed. Another advantage is that the compiler performs the final data structure selection, possibly in combination with standard program transformations if this selection cannot be resolved efficiently. Saving the initial program defined on the enveloping data structures enables the compiler to retarget the same code to other machines.

A user defined annotation is used to identify which of the declared dense data structures of the program are actually sparse. This annotation only needs to occur in the declarative part of the program and as such is not intrusive. The compiler can also prompt the programmer with the data structures declared in the program and inquire explicitly about the sparsity of each of them. After the sparse data structures are identified, the compiler might need specific information about the sparsity structure of these data structures, e.g. the density or bandwidth, to make efficient data structure selection. This can be obtained by interactive inquiries, user defined annotation, or by automatic evaluation of these characteristics using a library function which reads the sparse data structure from file, and performs the necessary statistics. Note that if sparse data structures are on file then specific input/output routines have to be inserted in the code.

In this paper, a short outline of the data structure selection and transformation method as described in [2] is given. This method is based on a *bottom-up* approach and consists of a three phase process. In the first phase the instructions in the code are identified which will be affected by the sparsity of data structures and transformations are applied to make fully use of these statements. In the second phase the data structure selection is performed and conflicts are resolved. In the third phase the actual transformations are performed to generate the resulting code.

The paper is organized as follows. In section 2 some background is given. In section 3, a small description of the *bottom-up* approach is presented which is further elaborated in section 4 with an example. Many details have been omitted. For a more comprehensive description see [2].

2 Preliminaries

In this section we present some notational conventions and terminology.

2.1 Sparse Matrices

A dense $m \times n$ matrix A is defined as a set of $m \cdot n$ elements a_{ij}, where $(i,j) \in I_A \times J_A$ (the **index set** of A), $I_A = \{1, \ldots, m\}$ and $J_A = \{1, \ldots, n\}$. If many elements are zero, the matrix is said to be sparse. A sparse matrix can be defined by its nonzero structure: $Nonz(A) = \{(i,j) \in I_A \times J_A | a_{ij} \neq 0\}$. The size of this set is the number of nonzero elements.

For a sparse matrix A sparsity can be exploited to save *storage requirements* by only storing the nonzero elements. Storage required to store the numerical values is called **primary storage**, while storage that is necessary to reconstruct the underlying matrix is referred to as **overhead storage**. In some cases, it is practical to store some zeros too, since usage of a simpler storage scheme with less overhead storage might compensate for the increase in the amount of primary storage and results in less run-time overhead. Elements that are stored explicitly are called **entries**. The set E_A with $Nonz(A) \subseteq E_A \subseteq (I_A \times J_A)$ indicates the indices of the entries. Therefore, $\forall_{(i,j) \in (I_A \times J_A) \setminus E_A} : a_{ij} = 0$ necessarily holds. If the maximum nonzero structure of A can be determined at compile-time, E_A can be chosen accordingly and a *static* data structure for A can be used. 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**). In the case of LU-factorization the term fill-in is used to refer to creation. Since some problems are only feasible if compact storage is used, the storage requirements of the data structure must be $O(|E_A|)$.

2.2 Indexed Statements

A FORTRAN-like language, extended with mathematical constructs, is used throughout this paper to illustrate analysis and transformations on the source code. The following construct, for example, is used to indicate a loop, where the loop-control variable `I` iterates over all values in **execution set** V. The execution order is specified by the natural ordering, unless stated otherwise. In this manner, it is possible to deal with an irregular stride.

```
DO I ∈ V
  S1(I)
  S2(I)
ENDDO
```

Statements that appear in a loop body at nesting depth d are called **indexed statements** of degree d, and are denoted $S(I_1, \ldots, I_d)$. Such statements

have different **instances**, where each instance is obtained by substitution of a corresponding value for every surrounding loop-control variable. The concept of static dependences [7, 8, 13], formed by dependences on statement instances, is used where required.

3 Data Structure Selection and Transformation

The computation in the original program is defined on a two dimensional array `REAL A(M,N)` as enveloping data structure for every sparse $m \times n$ matrix A. The kind of statements in this dense program that can exploit sparsity to save *computational time* can be identified with the following observation:

> **Observation:** Instances of statements where a zero is assigned to a non-entry or where an arbitrary variable is updated with a zero can be eliminated.

We will exploit this observation for zeros that result from constants or sparse data structures. Consider, for example, the following fragment, in which a sparse matrix A is used. The nonzero structure of A is given, in which every 'x' indicates an entry:

```
DO I = 1, 3
  DO J = 1, 3
S1 : ACC = ACC + A(I,J)
S2 : A(I,J) = A(I,J) * 2.0
  ENDDO
ENDDO
```

A	1	2	3
1	x		x
2			x
3		x	

It is clear, that only the following statement instances need to be executed, since S_1 updates a variable with elements of A, while S_2 manipulates elements of A respectively:

$S_1(1,1)$: `ACC = ACC + A(1,1)`
$S_2(1,1)$: `A(1,1) = A(1,1) * 2.0`
$S_1(1,3)$: `ACC = ACC + A(1,3)`
$S_2(1,3)$: `A(1,3) = A(1,3) * 2.0`
$S_1(2,3)$: `ACC = ACC + A(2,3)`
$S_2(2,3)$: `A(2,3) = A(2,3) * 2.0`
$S_1(3,2)$: `ACC = ACC + A(3,2)`
$S_2(3,2)$: `A(3,2) = A(3,2) * 2.0`

A property in common for this kind of statements is that usually the instances of these statements can be executed in arbitrary order. This is certainly true if no cross-iteration dependences hold, which occurs frequently for manipulating statements (e.g. S_2). If cross-iteration dependences exist, the original execution order must be preserved. However, if all cross-iterations dependences are caused by an accumulation (e.g. S_1) the ordering may change if it is allowed that roundoff errors, due to inexact computer arithmetic, might accumulate in a different way.

3.1 First Phase

The identification of statements that can exploit sparsity enables the compiler to take a bottom-up approach to program restructuring. First, every statement in the program is checked whether it contains an occurrence of a dense matrix A (e.g. `A(I,J)`) which in fact is sparse. If so, guards are created for such statements to differentiate between code that operates on entries and zero elements, and a two-way IF-statement results:

```
IF (I,J) ∈ E_A THEN
 ..A(I,J).. ← operation on an entry
ELSEIF (I,J) ∉ E_A THEN
 ..A(I,J).. ← operation on a zero element
ENDIF
```

The resulting code is called a **guarded block**, in which the '$\in E_A$'- and '$\notin E_A$'-tests are referred to as **guards**. Every *different* occurrence of a sparse matrix (i.e. the same matrix with different subscript functions or another matrix) in a statement introduces two additional guards in the guarded block. Such a guarded block is usually presented as a multiway IF-statement with a branch for every possible conjunction of the guards. This kind of code is highly inefficient, but reflects the overhead that will be introduced by the usage of a compact data structure, and offers the possibility to examine in which branches sparsity can be exploited.

In order to reason about a compact data structure for a sparse matrix A, still independent of its actual implementation, an abstract data structure $\mathtt{A}'$ is used in the description of the guarded block. Every entry a_{ij} of matrix A is stored at $\mathtt{A}'[\sigma_A(i,j)]$ with a bijective storage function $\sigma_A : E_A \rightarrow AD_A$, that maps the indices of entries to addresses in AD_A. The computation of such an address will be referred to as a σ_A-lookup. The following statement, for instance, is transformed into the given guarded block, where the second branch is eliminated because it can exploit sparsity:

```
ACC = ACC + A(I,2*J)   →   IF (I,2*J) ∈ E_A THEN
                            ACC = ACC + A'[σ_A(I,2*J)]
                           ENDIF
```

If matrix A occurs at the left-hand side and an arbitrary expression at the right-hand side, then sparsity cannot be exploited because the two branches must handle the alternation of the value of an entry, or creation respectively:

```
                        IF (I,J) ∈ E_A THEN
                         A'[σ_A(I,J)] = EXPR
A(I,J) = EXPR   →       ELSEIF (I,J) ∉ E_A THEN   (*)
                         A'[new_A(I,J)] = EXPR
                        ENDIF
```

Due to the fact that the nonzero structure of A changes in the second branch, function new_A is introduced. It returns the address of a new entry a_{ij}, and adapts σ_A, E_A and AD_A accordingly as a side-effect. Assigning a zero to an element of a sparse matrix only requires an action for entries:

```
                          IF (I,J) ∈ E_A THEN
A(I,J) = 0.0      →        CALL del_A(I,J)
                          ENDIF
```

Cancellation is indicated with subroutine del_A. The knowledge in which branches the right-hand side expression is zero will be based on the sparsity of arrays and constants only, i.e. it is assumed that the contents of dense variables and entries is always nonzero. Otherwise additional tests on the value of expressions are required. For example, the following guarded block results if matrix B is also sparse, while $(*)$ is used otherwise:

```
                A(I,J) = B(I,J)
                       ↓
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
```

Because guards introduce a lot of unwanted overhead, actions are taken to eliminate this overhead. A basic technique consists of including a guard that **dominates** its guarded block, which means that it is satisfied by the condition in every branch that remains, in the execution set of a surrounding loop, referred to as **guard encapsulation**. For example, guard '$(\mathtt{I},\mathtt{J}) \in E_A$' dominates the following guarded block, and is a candidate for encapsulation in the execution set of the J-loop:

```
DO I = 1, M
 DO J = 1, N
  IF (I,J) ∈ E_A THEN
   A'[σ_A(I,J)] = A'[σ_A(I,J)] * 10.0
  ENDIF
 ENDDO
ENDDO
```

Guard encapsulation results in an irregular index set, that corresponds to those iterations in which entries (elements for which the guard holds) are used. Consequently, encapsulation is only feasible if the addresses of these entries can be easily generated. This is true if, for example, these addresses are contiguous, which also improves locality. In general, if an invertible subscript function f_2 is used, and the set PAD_A^i contains the addresses of all entries referenced in iteration i, the following conversion results in semantically equivalent code:[1]

```
DO I ∈ V
 DO J ∈ V^I
  IF (f1(I), f2(J)) ∈ E_A THEN
   S(I,J): ...A'[σ_A(f1(I), f2(J))]...
  ENDIF
 ENDDO
ENDDO
```

[1] If cancellation or creation along the access pattern occurs complex code is required to deal with a variable execution set PAD_A^i.

```
                          ↓
DO I ∈ V
 DO JA ∈ PAD_A^I
  S(I, f2^-1(π2 · σ_A^-1(JA))) : ... A'[JA] ...
 ENDDO
ENDDO
```

Test overhead is eliminated from the loop-body, and $|PAD_A^i| \leq |V^i|$. Computation of $\pi_2 \cdot \sigma_A^{-1}(ad)$, where $ad \in PAD_A^i$ and $\pi_i \cdot \vec{x} = x_i$, is necessary if the value of J, i.e. the original iteration, must be reconstructed (similarly $\pi_1 \cdot \sigma_A^{-1}(ad)$ is required if the roles of I and J are interchanged). If an ordering constraint on the statement instances of one execution of the J-loop must be satisfied, the order in which the addresses are generated must correspond to the original execution order. Otherwise, this order might be arbitrary.

Note that if an **access pattern** P_A^i of one particular occurrence of a sparse matrix is defined as the indices of all the elements operated on in one iteration of the I-loop, then PAD_A^i contains the addresses of the elements with indices in $P_A^i \cap E_A$. Since some access patterns are very likely going to be subsets of other access patterns in different parts of the program, and every entry is preferred to be stored only once, usually address set $\overline{PAD_A^i}$ is considered which consist of addresses of all entries that correspond to an access pattern contained in the same row or column that envelops the actual access pattern of the loop. In this case an additional test is required to check whether addresses belong to entries of the original access pattern: $f_2^{-1}(\pi_2 \cdot \sigma_A^{-1}(\mathtt{JA})) \in V^i$. However, the overhead of this test is small.

Further steps are taken to optimize the resulting code: guard manipulations. Some of these manipulations resemble standard program transformation (loop distribution, loop fusion, expression splitting and statement reordering), while others are specifically for guards (guard collapsing, movement and absorption in user-supplied conditionals). The most important goal of these manipulations is to increase the potential of guard encapsulation, because the resulting loops fully exploit sparsity. Reduction of test and lookup overhead is another important goal. Some examples are given in section 4, while the manipulations are presented in more detail in [2].

3.2 Second Phase

In the second phase, all access patterns through the matrices are examined. With knowledge of the properties of sparse vector storage schemes [4, 12], certain constraints are imposed on the storage of the entries along access patterns that belong to one particular occurrence of a matrix, (1) to enable guard encapsulation, or (2) to make the innermost operations fast. The following data structure, based on existing data structures [3, 4, 6, 11, 12, 14], is one of the possible data structures for general sparse matrices that might be selected:

Entries belonging to one access pattern are stored consecutively as a sparse vector in two parallel arrays AVAL and AIND, in which the numerical values and $\pi_2 \cdot \sigma_A^{-1}$-values are stored. For every access pattern there is a corresponding element in arrays ALOW and AHIGH to record the first and last index used. Note that all PAD_A^i can be easily generated since addresses can be referred to by their index, and that all $\pi_2 \cdot \sigma_A^{-1}$-values are available, which makes guard encapsulation for the corresponding access patterns feasible. A possible ordering constraint on the generation of addresses in PAD_A^i imposes an ordering requirement on the entries within one access pattern. Insertions and deletions of entries along unordered access patterns is fast if there is some surrounding working space, but requires some extra data movement otherwise.[2] Another advantage of this data structure is that a dense representation of the access pattern can be constructed before it is operated on, by expanding the sparse storage. With a **scatter**-operation all entries along the access pattern are copied to their real position in a dense array AP, on which all operations can be performed without the lookup overhead inherent to sparse representations. Insertion is accounted for whenever a zero element in AP changes into a nonzero element. If assignments to elements of AP are done, a **gather**-operation is required afterwards.

For every sparse matrix A, the constraints imposed by all relevant occurrences are collected, i.e. it is examined if particular (enveloping) access patterns through a sparse matrix in a program can be stored consistently. If an element appears in two different access patterns, then a conflict results, since single storage of every element is preferred to prevent redundancy. Conventional restructuring techniques, e.g. loop interchanging or index set splitting, can assist in reshaping some access patterns to achieve consistency. Sometimes, conflicts cannot be solved. In this case a data structure has to be selected that does not match all access patterns. Access patterns that are traversed many times, take precedence in this decision.

3.3 Third Phase

Finally, in the last phase, the data structure transformations are applied, i.e. the *code* is converted into a format that operates on the selected data structures.

[2] In this paper we will assume that access patterns can always be moved to the end of the array where most working space is kept in order to limit the effect of address alteration, which e.g. complicates guard encapsulation. In general, however, garbage collection is occasionally required, possibly affecting the addresses of *all* entries.

3.3.1 Encapsulated Guards

If a guard has been encapsulated in the execution set of a directly surrounding loop, the following construct is generated, where $\tilde{\imath}$ indicates the index in ALOW and AHIGH that is associated to access pattern P_A^i. The bounds of the J-loop are possibly dependent on surrounding loop-control variable I, and f_2(J) has the form $a_2 + b_2 \cdot$ J. Note that the computation of the inverse of f_2 might be required to perform the loop bounds checking $(*)$:

```
                ...
                 DO J = L^I, U^I
                  ...A(f1(I), f2(J))...
                 ENDDO
                ...
                          ↓
...
 DO JA = ALOW(Ĩ), AHIGH(Ĩ)
  J = AIND(JA) - a2
  IF (J MOD b2 = 0) THEN   } ← required if b2 ≠ 1
    J = J DIV b2           }
   IF ((L^I ≤ J) AND (J ≤ U^I)) THEN  (*)
    ...AVAL(JA)...      ↑
   ENDIF     if a compare value coincides with a bound
 ENDIF       of all stored (enveloping) access patterns,
 ENDO        the corresponding compare is eliminated
...
```

3.3.2 Scatter and Gather Operations

If many operations are performed along a certain access pattern, it can be **expanded** to a dense vector before it is operated on. This temporary dense representation can be accessed without lookup overhead. The following framework is constructed:

```
 DO I = 1, M
  CALL SCATTER(AVAL,AIND,ALOW(Ĩ),AHIGH(Ĩ),AP)
  DO J = 1, N
   IF (AP(f2(J)) = 0.0) THEN
    CALL INSERT(AVAL,AIND,ALOW(Ĩ),AHIGH(Ĩ),
+       f2(J),0.0,ALAST,ADIM)
   ENDIF
   AP(f2(J)) = ...AP(f2(J))...
  ENDDO
  CALL GATHER(AVAL,AIND,ALOW(Ĩ),AHIGH(Ĩ),AP)
 ENDDO
```

Subroutine INSERT inserts a new entry in the access pattern *without* a check if this entry is already present to keep the operation fast, and requires some additional information to support data movement.[3] Consequently, if a zero element is stored as an entry and becomes nonzero, a redundant insertion results. Subroutine GATHER stores the value of every entry, found in AP, back in the data structure, while cancellation based on the actual values is accounted for efficiently. Used elements of array AP are directly reset to zero, to support the next SCATTER operation *and* to ensure that redundant inserted entries are further ignored.

[3] Small modifications to some constructs in this section are required if an ordering on entries must be imposed.

3.3.3 Remaining Occurrences

After code has been generated for occurrences with encapsulated guards or along expanded access patterns, code is generated for the remaining occurrences.

Function LOOKUP returns the address of an element if it is an entry, or $\perp$ otherwise. The implementation is such that AVAL($\perp$)=0.0 holds. The following function call is substituted for an occurrence $\mathtt{A}(f_1(\mathtt{I}), f_2(\mathtt{J}))$ at the right-hand side without dominating guard:

```
AVAL( LOOKUP(AIND,ALOW(Ĩ),AHIGH(Ĩ),f2(J)) )
```

Assignment of a zero to an occurrence results in cancellation if the corresponding element is an entry, while no action is performed otherwise:

```
IND = LOOKUP(AIND,ALOW(Ĩ),AHIGH(Ĩ),f2(J))
IF (IND ≠ ⊥) THEN
 CALL DELETE(AVAL,AIND,ALOW(Ĩ),AHIGH(Ĩ),IND)
ENDIF
```

The following code results for an occurrence with a dominating guard that has not been encapsulated:

```
IND = LOOKUP(AIND,ALOW(Ĩ),AHIGH(Ĩ),f2(J))
IF (IND ≠ ⊥) THEN
  ... AVAL(IND) ...
ENDIF
```

If a loop invariant guard has been hoisted out a loop, the call to LOOKUP with following test is generated at the corresponding position, although care must be taken that address IND does not change as a result of data movement. Similarly, the lookup overhead can be shared if identical guards have been collapsed.

For an occurrence at the left-hand side without dominating guard, the following construct is generated, where AVAL(IND) can be used if the occurrence also appears at the right-hand side:

```
IND = LOOKUP(AIND,ALOW(Ĩ),AHIGH(Ĩ),f2(J))
EXPR = ... right hand side expression ...
IF (IND ≠ ⊥) THEN
 AVAL(IND) = EXPR
ELSE
 CALL INSERT(AVAL,AIND,ALOW(Ĩ),AHIGH(Ĩ),
+            f2(J),EXPR,ALAST,ADIM)
ENDIF
```

If there are *multiple* occurrences of sparse matrices in one guarded block, combinations of the previous codes are generated. Unique IND*i* are used when necessary. If code generation starts with occurrences that have dominating guards, most combinations are straightforward to construct. However, some combinations with an occurrence without dominating guard coerce further distinction, as is illustrated below:

```
            A(I,J) = A(I,J) * B(I,J)
                          ↓
IND1 = LOOKUP(AIND,ALOW(Ĩ),AHIGH(Ĩ),J)
IF (IND1 ≠ ⊥) THEN
 IND2 = LOOKUP(BIND,BLOW(Ĩ),BHIGH(Ĩ),J)
 IF (IND2 ≠ ⊥) THEN
  AVAL(IND1) = AVAL(IND1) * BVAL(IND2)
 ELSE
  CALL DELETE (AVAL,AIND,ALOW(Ĩ),AHIGH(Ĩ),IND1)
 ENDIF
ENDIF
```

4 Example

The whole data structure selection and transformation method is illustrated with the following version of LU-factorization without pivoting, where matrix A is in fact sparse. Its *different* occurrences are subscripted accordingly in the code:

```
DO I = 1, N - 1
 DO J = I + 1, N
S1 : A1(J,I) = A1(J,I) / A2(I,I)
  DO K = I + 1, N
S2 : A3(J,K) = A3(J,K) - A4(J,I) * A5(I,K)
  ENDDO
 ENDDO
ENDDO
```

(1)

In the first phase, guarded blocks for S_1 and S_2 are generated with guards for every *different* occurrence of the sparse matrix, and it is determined which branches can be eliminated. The following code results, because S_1 manipulates $\mathtt{A}_1$ and S_2 is an updating statement on $\mathtt{A}_3$. Because the value of $\mathtt{A}_2$ is invariant in one execution of the J-loop, the corresponding guarded block is hoisted out this loop, and the computed value is saved in a temporary variable PIV. The occurrences of abstract data structure A′ are subscripted to reflect the correspondence with the original occurrences of matrix A:

```
DO I = 1, N - 1
 IF (I,I) ∈ EA THEN
  PIV = A'2[σA(I,I)]
 ELSEIF (I,I) ∉ EA THEN
  PIV = 0.0
 ENDIF
 DO J = I + 1, N
  IF (J,I) ∈ EA THEN
   A'1[σA(J,I)] = A'1[σA(J,I)] / PIV
  ENDIF
  DO K = I + 1, N
   IF (J,K) ∈ EA ∧ (J,I) ∈ EA ∧ (I,K) ∈ EA THEN
    A'3[σA(J,K)] = A'3[σA(J,K)] - A'4[σA(J,I)] * A'5[σA(I,K)]
   ELSEIF (J,K) ∉ EA ∧ (J,I) ∈ EA ∧ (I,K) ∈ EA THEN
    A'3[newA(J,K)] = 0.0 - A'4[σA(J,I)] * A'5[σA(I,K)]
   ENDIF
  ENDDO
 ENDDO
ENDDO
```

(2)

Guard '$(\mathtt{J},\mathtt{I}) \in E_A$' can be hoisted out the K-loop, because the value of this guard is invariant over the iterations of *one* execution of this loop. This hoisted guard can be collapsed with the identical remaining guard of S_1 because S_1 cannot change its value. The following code results, where occurrences belonging to the collapsed guards are considered equal:

```
DO I = 1, N - 1
 IF (I,I) ∈ E_A THEN
  PIV = A'_2[σ_A(I,I)]
 ELSEIF (I,I) ∉ E_A THEN
  PIV = 0.0
 ENDIF
 DO J = I + 1, N
  IF (J,I) ∈ E_A THEN
   A'_1[σ_A(J,I)] = A'_1[σ_A(J,I)] / PIV
   DO K = I + 1, N
    IF (J,K) ∈ E_A ∧ (I,K) ∈ E_A THEN
     A'_3[σ_A(J,K)] = A'_3[σ_A(J,K)] - A'_1[σ_A(J,I)] * A'_5[σ_A(I,K)]
    ELSEIF (J,K) ∉ E_A ∧ (I,K) ∈ E_A THEN
     A'_3[new_A(J,K)] = 0.0 - A'_1[σ_A(J,I)] * A'_5[σ_A(I,K)]
    ENDIF
   ENDDO
  ENDIF
 ENDDO
ENDDO
```

(3)

Guards '$(\mathtt{J},\mathtt{I}) \in E_A$' and '$(\mathtt{I},\mathtt{K}) \in E_A$' dominate their guarded blocks, and have the potential of encapsulation in the execution sets of the J- and K-loop. Occurrence $\mathtt{A}_3$ is a suited candidate for expansion of its access pattern.

In the second phase, access patterns through the matrix are considered: $P^i_{A_1} = \{(j,i) | i < j \leq \mathtt{N}\}$ for $1 \leq i \leq \mathtt{N}$, $P_{A_2} = \{(i,i) | 1 \leq i \leq \mathtt{N}\}$, $P^j_{A_3} = \{(j,k) | i < k \leq \mathtt{N}\}$ for $i < j \leq \mathtt{N}$ and $P^i_{A_5} = \{(i,k) | i < k \leq \mathtt{N}\}$ for $1 \leq i \leq \mathtt{N}$. Sparse row-wise storage is consistent with enveloping access patterns of $P^j_{A_3}$ and $P^i_{A_5}$. Simple guard encapsulation in the execution set of the K-loop is possible, because no assignments to elements of row I are done in one execution of the K-loop. Generation of scatter- and gather-operations for $\mathtt{A}_3$ become feasible. No ordering is required on the storage of elements in every row, since there are no cross-iteration dependences carried by the K-loop. The disadvantage of inconsistency with access pattern $P^i_{A_1}$ is that encapsulation of guard '$(\mathtt{J},\mathtt{I}) \in E_A$' is disabled.

In the third phase, code is generated. Code generation starts with encapsulation of guard '$(\mathtt{I},\mathtt{K}) \in E_A$' in the execution set of K. A single compare suffices to test for inclusion in the old execution set. After that, scatter- and gather-operations for $\mathtt{A}_3$, with outermost controlling J-loop, are generated. Because no ordering is required, subroutine INSERT is generated. All references and assignment to elements on this access patterns are replaced by corresponding operations on the expanded access pattern AP, because this dense vector contains the most recent values. This affects $\mathtt{A}_1$, for which a test is generated at the position of its guard. Finally, the lookup code for remaining occurrence A_2 is generated before the J-loop, and the following code results:

```
DO I = 1, N - 1
 PIV = AVAL( LOOKUP(AIND,ALOW(I),AHIGH(I),I) )
 DO J = I + 1, N
  CALL SCATTER(AVAL,AIND,ALOW(J),AHIGH(J),AP)
  IF (AP(I) ≠ 0.0) THEN
   AP(I) = AP(I) / PIV
   DO KA = ALOW(I), AHIGH(I)
    K = AIND(KA)
    IF (I + 1 ≤ K) THEN
     IF (AP(K) = 0.0) THEN
      CALL INSERT(AVAL,AIND,ALOW(J),AHIGH(J),
+                 K,0.0,ALAST,ADIM)
     ENDIF
     AP(K) = AP(K) - AP(I) * AVAL(KA)
    ENDIF
   ENDDO
  ENDIF
  CALL GATHER(AVAL,AIND,ALOW(J),AHIGH(J),AP)
 ENDDO
ENDDO
```

(4)

As an illustration of the flexibility of automatic code generation, code that results if *column*-wise storage is selected as implementation of $\mathtt{A}'$ in version (3) is also presented. Because this storage is consistent with an enveloping access pattern of $P^i_{A_1}$, and no assignments to column I are done during one execution of the J-loop, simple guard encapsulation in the execution set of the J-loop is possible, which reduces the number of iterations of a loop at higher level than in version (4). No ordering constraint is imposed on the elements stored in one column. However, this choice disables guard encapsulation and scatter- and gather-operations for the innermost loop. The result is listed below, in which array AIND is now used to store *row* numbers:

```
DO I = 1, N - 1
 PIV = AVAL( LOOKUP(AIND,ALOW(I),AHIGH(I),I) )
 DO JA = ALOW(I),AHIGH(I)
  J = AIND(JA)
  IF (I + 1 ≤ J) THEN
   AVAL(JA) = AVAL(JA) / PIV
   DO K = I + 1, N
    IND1 = LOOKUP(AIND,ALOW(K),AHIGH(K),I)
    IF (IND1 ≠ ⊥) THEN
     IND2 = LOOKUP(AIND,ALOW(K),AHIGH(K),J)
     EXPR = AVAL(IND2) - AVAL(JA) * AVAL(IND1)
     IF (IND2 ≠ ⊥) THEN
      AVAL(IND2) = EXPR
     ELSE
      CALL INSERT(AVAL,AIND,ALOW(K),AHIGH(K),
+                 J,EXPR,ALAST,ADIM)
     ENDIF
    ENDIF
   ENDDO
  ENDIF
 ENDDO
ENDDO
```

(5)

The compiler can also decide to interchange the I- and J-loops in version (2). This transformation is valid, as can be concluded from dependence analysis on the *original* code (1). Because the execution set of the innermost loop depends on the loop-control variable of the outermost loop, so-called triangular loop interchanging is done [13]. Guard '$(\mathtt{J},\mathtt{I}) \in E_A$' can be

hoisted out the K-loop and is collapsed with the same guard of S_1. This results in the following code, where dominating guard '(J,I) $\in E_A$' has been lifted to avoid the notational need for an additional branch:

```
DO J = 2, N
 DO I = 1, J - 1
  IF (J,I) ∈ E_A THEN
   IF (I,I) ∈ E_A THEN
    A'_1[σ_A(J,I)] = A'_1[σ_A(J,I)]  / A'_2[σ_A(I,I)]
   ELSEIF (I,I) ∉ E_A THEN
    A'_1[σ_A(J,I)] = A'_1[σ_A(J,I)]  / 0.0    (error)
   ENDIF
   DO K = I + 1, N
    IF (J,K) ∈ E_A ∧ (I,K) ∈ E_A THEN
     A'_3[σ_A(J,K)] = A'_3[σ_A(J,K)] - A'_1[σ_A(J,I)] * A'_5[σ_A(I,K)]
    ELSEIF (J,K) ∉ E_A ∧ (I,K) ∈ E_A THEN
     A'_3[new_A(J,K)] = 0.0 - A'_1[σ_A(J,I)] * A'_5[σ_A(I,K)]
    ENDIF
   ENDDO
  ENDIF
 ENDDO
ENDDO
```

(6)

Access patterns $P^j_{A_1} = \{(j,i)|1 \leq i < j\}$ for $2 \leq j \leq \mathtt{N}$, $P_{A_2} = \{(i,i)|1 \leq i \leq \mathtt{N}\}$, $P^j_{A_3} = \{(j,k)|i < k \leq \mathtt{N}\}$ for $2 \leq j \leq \mathtt{N}$, and $P^i_{A_5} = \{(i,k)|i < k \leq \mathtt{N}\}$ for $1 \leq i < j$ result. Clearly, sparse row-wise storage scheme is suited, since it matches enveloping access patterns of $P^j_{A_1}$, P_{A_3} and $P^i_{A_5}$. This storage scheme favors scatter- and gather-operations for $\mathtt{A}^j_3$, while loop interchanging has increased the number of operations performed on one expanded access pattern. Because assignments to elements of row J are done in one execution of the I-loop, this affects $\mathtt{A}_1$ and disables simple encapsulation of guard '(J,I) $\in E_A$'. However, encapsulation of guard '(I,K) $\in E_A$' is possible:

```
  DO J = 2, N
   CALL SCATTER(AVAL,AIND,ALOW(J),AHIGH(J),AP)
   DO I = 1, J - 1
    IF (AP(I) ≠ 0.0) THEN
     AP(I) = AP(I) /
+           AVAL( LOOKUP(AIND,ALOW(I),AHIGH(I),I) )
     DO KA = ALOW(I), AHIGH(I)
      K = AIND(KA)
      IF (I + 1 ≤ K) THEN
       IF (AP(K) = 0.0) THEN
        CALL INSERT(AVAL,AIND,ALOW(J),AHIGH(J),
+                  K,0.0,ALAST,ADIM)
       ENDIF
       AP(K) = AP(K) - AP(I) * AVAL(KA)
      ENDIF
     ENDDO
    ENDIF
   ENDDO
   CALL GATHER(AVAL,AIND,ALOW(J),AHIGH(J),AP)
  ENDDO
```

(7)

Running these different sparse versions (4), (5) and (7) on one CPU of a CRAY-YMP gave easily performance variations of up to a factor of 60. This clearly indicates that the transformations as described in this paper should be very carefully handled.

5 Conclusions

In this paper, a short outline was given of a method [2] that enables restructuring compilers to get a grip on sparse matrix computations. The method relies on the fact that the code information being obscured in sparse matrix code by indirect addressing and complicated branch structures, for example, can be presented in a much cleaner form to the compiler by a corresponding dense program. The method not only produces efficient sparse matrix code, but also allows enough flexibility to retarget the code to different architectures. In forthcoming papers we will describe implementation issues that arise if this method is embedded in an existing prototype compiler and how *data dependence analysis* on the original dense code can be used to optimize the resulting code on sparse data structures. Knowledge of which data dependences are unaffected by the generation of the presented code will turn out to be essential.

References

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

[2] Aart J.C. Bik and Harry A.G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. Technical Report no. 92-25, Dept. of Computer Science, Leiden University, 1992.

[3] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers.* Society for Industrial and Applied Mathematics, 1991.

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

[5] K.A. Gallivan, B.A. Marsolf, and H.A.G. Wijshoff. Mcsparse: A parallel sparse unsymmetric linear system solver. Technical Report no. 1142, Center for Supercomputing Research and Development, University of Illinios, 1991.

[6] Alan George and Joseph W. Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice-Hall Inc., 1981.

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

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

[9] Joel H. Saltz, Ravi Mirchandaney, and Kathleen Crowley. The doconsider loop. In *ACM Conference Proceedings, 3^{th} International Conference of Supercomputing*, pages 29–40, 1989.

[10] Joel H. Saltz, Ravi Mirchandaney, and Kathleen Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, pages 603–612, 1991.

[11] Reginal P. Tewarson. *Sparse Matrices.* Academic Press, New York, 1973.

[12] 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.

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

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