A Framework for Sparse Matrix Code Synthesis from High-level Specifications

Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill
Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract

We present compiler technology for synthesizing sparse matrix code from (i) dense matrix code, and (ii) a description of the index structure of a sparse matrix. Our approach is to embed statement instances into a Cartesian product of statement iteration and data spaces, and to produce efficient sparse code by identifying common enumerations for multiple references to sparse matrices. The approach works for imperfectly-nested codes with dependences, and produces sparse code competitive with hand-written library code for the Basic Linear Algebra Subroutines (BLAS).

1 Introduction

Many applications that require high-performance computing perform computations on sparse matrices. For example, the finite-element method for solving partial differential equations approximately requires the solution of large linear systems of the form `Ax = b` where `A` is a large sparse matrix. Some web-search engines and data-mining codes compute eigenvectors of large sparse matrices that represent how often certain words occur in documents of interest.

Sparse matrices are usually stored in *compressed formats* in which zeros are not stored explicitly [18]. This reduces storage requirements, and in many codes, also eliminates the need to compute with zeros. Figure 1 shows a sparse matrix and a number of commonly used compressed formats that we will use as running examples in this paper.

The simplest format is *Co-ordinate storage* (COO) in which three arrays are used to store non-zero elements and their row and column positions. The non-zeros may be ordered arbitrarily. *Compressed Sparse Row storage* (CSR) is a commonly used format that permits indexed access to rows but not columns. Array `values` is used to store the non-zeros of the matrix row by row, while another array `colind` of the same size is used to store the column positions of these

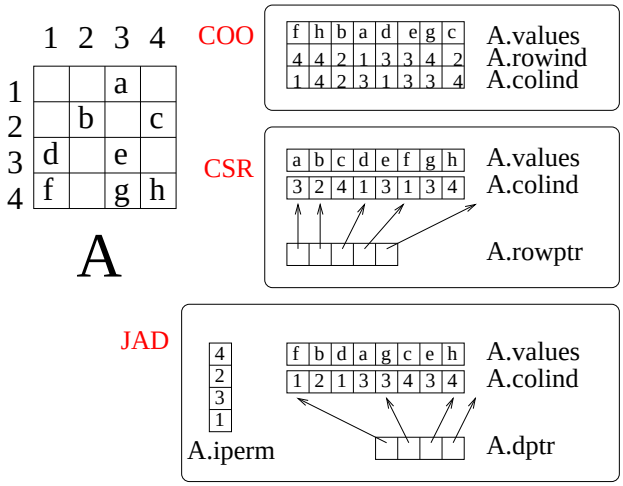

Figure 1: Sparse Storage Formats

entries. A third array `rowptr` has one entry for each row of the matrix, and it stores the position in `values` of the first non-zero element of each row of the matrix. *Compressed Sparse Column storage* (CSC, not shown) is the transpose of CSR in which the non-zeros are stored column-by-column, and it offers indexed access to columns.

A more complex format is the Jagged Diagonal (JAD) format.[1] An instance of a JAD matrix is constructed by (i) "compressing" the rows of the matrix so that zero elements are eliminated (introducing an auxiliary array, `colind`, to maintain the original column indices); (ii) sorting the compressed rows by the number of non-zeros within each row in *decreasing* order (introducing a permutation vector, `iperm`); and (iii) storing the columns of the compressed and sorted matrix, which are called the "diagonals", in two vectors, `colind` and `values`. Finally, Figure 2 illustrates the Diagonal (DIA) storage format which is appropriate for banded matrices. Only the diagonals containing non-zero elements are stored, and elements are addressed by diagonal and offset.

In this paper, we will focus on language and systems sup-

[0]This work was supported by NSF grants CCR-9720211, EIA-9726388, ACI-9870687, EIA-9972853.

[1]See the Appendix for a detailed description of the JAD format.

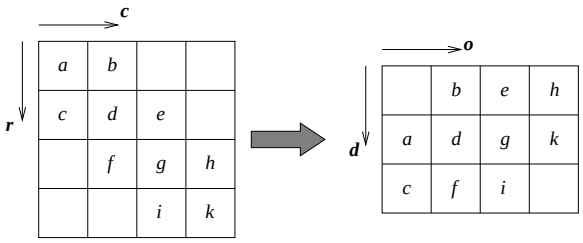

Figure 2: DIA Storage Format

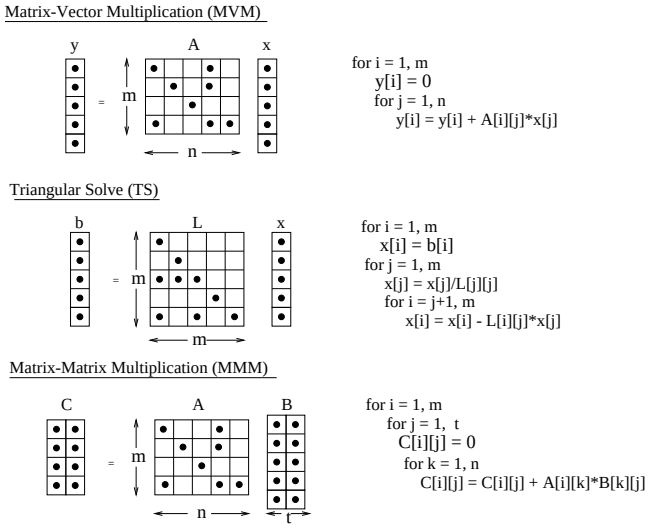

Figure 3: Basic Linear Algebra Subroutines

port for iterative sparse matrix algorithms. There are at least forty or fifty formats that are widely used, and it is common to use application-specific formats. Since each format requires its own carefully tuned code, the problem of designing libraries of iterative algorithms which can support all these compressed formats and which can be easily extended to new formats is a formidable one.

The approach taken by the numerical analysis community (for example, in the PETSc library from Argonne [2]) is to encapsulate the format-dependent code into a set of *Basic Linear Algebra Subroutines* (BLAS), the most important of which are shown in Figure 3. These routines are invoked by format-independent implementations of iterative methods. Therefore, the high-level iterative codes have to be written just once, but they must be linked with format-specific BLAS. For dense matrices, highly tuned implementations of BLAS are routinely provided by computer vendors [7]. For sparse matrices, the software problem is much more difficult because of the need to support such a large number of formats. Although a number of sparse BLAS libraries have been written [8, 10, 20], they have had limited success because (i) they support only a small number of formats, and (ii) they provide no leverage for people designing new formats.

One solution, first proposed by Bik and Wijshoff [4, 5], is to use restructuring compiler technology to *synthesize* sparse matrix programs from dense matrix programs. Their compiler restructured input codes to match a *Compressed Hyperplane Storage* (CHS) format (CSR and CSC are special cases of this format) whenever possible. More recently, Pugh and Shpeisman [19] proposed an intermediate program representation for sparse codes that allows them to predict asymptotic program efficiency and make decisions about choosing sparse matrix formats.

In our previous work [14], we argued that (i) sparse matrices should be viewed as *sequential-access* data structures [22], and (ii) efficient sparse codes should be organized if possible as *data-centric computations* that enumerate nonzero elements of sparse matrices and perform computations with these elements as they are enumerated. This view is in contrast to the conventional view of arrays as *random-access* data structures, a view that is useful only when the array is dense. An important refinement to the sequential-access view is that some sparse formats such as CSR and CSC have an indexing structure, and should therefore be viewed as *indexed-sequential-access* structures [22]. For example, the CSR format permits indexing to rows (but not to columns), and this indexing structure must be exploited in some codes such as matrix multiplication.

To avoid having to write different data-centric programs for each sparse format, we exploit the idea of *generic programming* [16]. The idea behind generic programming is to program the algorithm abstractly in a data-structure-neutral fashion just once, obtaining concrete, data-structure-specific programs by instantiating the abstract code with different data structure implementations. The most well-known example of this approach is the Standard Template Library (STL) in C++. Generic programming is also exploited in the MTL matrix library [21].

Conventional generic programming uses a *single* API which is (i) used by the algorithm designers and (ii) supported by all data structure implementors. To obtain efficient code, we found that our system needed *two* API's: a high-level one used by the algorithm designers, and a low-level one supported by data structure implementors. The high-level API is the API of dense matrix programs, while the low-level API permits sparse matrix format designers to specify details such as the indexing structure of matrices. *The instantiation of generic programs into concrete programs requires restructuring compiler technology to translate from the high-level API to the low-level API.* This translation requires restructuring of the dense code at a deep level to make it data-centric for the desired sparse format, so it is considerably more complex than the C++ template instantiation mechanism. For example, generic triangular solve (TS) can be coded in our system as shown in Figure 4.[2] The programmer writes code as though all matrices were dense, but

[2] To keep the examples simpler, we assume that the result is to be stored in the right-hand side vector b.

```
#pragma instantiate with Bernoulli
template <class T, class BASE>
void ts(T L, BASE b[])
{
    for (int j=0; j<L.columns(); j++) {
/*S1*/  b[j] = b[j]/L[j][j];
        for (int i=j+1; i<L.columns(); i++)
/*S2*/      b[i] = b[i] - L[i][j]*b[j];
    }
}

// Will be instantiated with the Bernoulli compiler.
template void ts(Jad<double> L, double b[]);
```

Figure 4: Generic Triangular Solve with Instantiation

specifies which classes must be used to implement sparse matrices. Pragmas indicate which template definitions are to be instantiated by the sparse compiler; the rest of the template definitions are handled by the underlying C++ compiler.

Previously, we showed how this restructuring could be done in the simpler case when the program is a perfectly-nested loop nest in which iterations can be executed in any order [11]. However, many codes of interest such as the triangular solve in Figure 4 are not perfectly-nested and data dependences do not allow executing statements in arbitrary order. This paper develops the sparse code synthesis technology for a general class of codes consisting of imperfectly-nested loops with dependences.

The rest of the paper is organized as follows. In Section 2, we describe how the user can specify sparse matrix formats in our generic programming system. In Section 3, we describe a general restructuring compiler technology for restructuring imperfectly-nested loops. In Section 4, we discuss how this technology can be used to synthesize sparse matrix code from dense matrix code and sparse format descriptions. Although our approach can be used to handle codes in which sparse matrices suffer fill, we focus on codes without fill in this paper since this is adequate for the BLAS codes. In Section 5, we present experimental results demonstrating that our approach produces code competitive with a hand-optimized sparse BLAS library. Finally, we summarize the paper in Section 6.

2 Generic Programming and Matrix Abstraction

For the purpose of this paper, the most important aspect of a sparse format is its index structure.

To appreciate the importance of exploiting the index structure in code restructuring, consider the triangular solve code of Figure 4. Vector `b` is dense and the lower triangular matrix `L` is sparse. The code is imperfectly-nested because statement `S1` is not nested in the `i` loop. Since matrix `L` is traversed by columns and CSC permits random access to columns, it is relatively straight-forward to generate data-centric sparse code for CSC, shown in Figure 5, that enumerates the non-zero elements of the matrix and performs

```
for col = enumerate cols of L in increasing order
  for row = enumerate L[*][col] in increasing order
    val = L[row][col];
      if (row == col)          //diagonal element
        b[row] = b[row]/val;
      else if (row > col)      //lower triangle
        b[row] = b[row] - val*b[col];
      else ;                   //upper triangle
```

Figure 5: Data-centric Pseudocode for Triangular Solve

$$
\begin{array}{rcl}
E & : & \mathit{Index} \rightarrow E \\
 & | & \mathit{map}\{F(\mathit{in}) \mapsto \mathit{out} : E\} \\
 & | & \mathit{perm}\{P(\mathit{in}) \mapsto \mathit{out} : E\} \\
 & | & E \cup E \\
 & | & E \oplus E \\
 & | & v \\
 & & \\
\mathit{Index} & : & \mathit{attribute} \\
 & | & \langle \mathit{attribute}, \ldots, \mathit{attribute} \rangle \\
 & | & (\mathit{attribute} \times \cdots \times \mathit{attribute})
\end{array}
$$

Figure 6: Sparse Matrix Abstraction

computations with each of these elements.

For CSR storage however, it is necessary to restructure the code first so that it walks over rows of `L`, since CSR storage provides random access only to rows of a matrix and not to its columns. Therefore, we need a way of describing the index structure of sparse formats, and we need technology to restructure code to match this index structure.

The grammar in Figure 6 is used to describe the index structure of a sparse matrix to our system [14]. The most important rule for specifying index structure is the $\mathit{Index} \rightarrow E$ (*nesting*) production rule. For example, a CSR matrix is described as $r \rightarrow c \rightarrow v$, indicating that rows must be accessed first, and within each row, elements within columns can be enumerated. The $\mathit{map}\{F(\mathit{in}) \mapsto \mathit{out} : E\}$ and $\mathit{perm}\{P(\mathit{in}) \mapsto \mathit{out} : E\}$ rules are used to describe linear and permutation transformations on the matrix indices. For example, a matrix in DIA storage format can be described as $\mathit{map}\{d + o \mapsto r, o \mapsto c : d \rightarrow o \rightarrow v\}$, while the *perm* operator is useful for describing formats like JAD. The $E' \cup E''$ (*aggregation*) rule is used to describe a matrix that is a collection of two formats, such as a format in which the diagonal elements are stored separately from the off-diagonal ones. Enumerating the elements of such matrix requires enumerating both E' and E''. Finally, the $E' \oplus E''$ (*perspective*) rule means that the matrix can be accessed in different ways, using either of the index structures E' or E''. As we will see, JAD is an example of such a format.

The $\langle \mathit{attribute}, \ldots, \mathit{attribute} \rangle$ notation describes an index

obtained from multiple co-ordinates enumerated together, as in the COO format ($\langle r, c\rangle \rightarrow v$). On the other hand, (*attribute* $\times \cdots \times$ *attribute*) denotes independent indices, as in a dense matrix ($(r \times c) \rightarrow v$).

Each term E is optionally annotated with the following *enumeration properties*.

- *Enumeration order*: a description of the order in which coordinate values can be enumerated efficiently. For the CSR format above, r is random-access, and within each row, c can be enumerated efficiently in increasing order.
- *Enumeration bounds*: a description of the coordinate values that actually occur in the enumeration. A lower triangular matrix, for example, could be annotated $1 \leq c \leq r \leq \texttt{N}$.

In addition to specifying this index structure, the sparse format designer must write the actual code to perform these enumerations. Each production in the view grammar given in Figure 6 has an associated interface, which we have implemented in C++ as a small number of abstract classes [14]. The programmer conveys views of a storage format to the sparse compiler by writing a set of classes that inherit from the appropriate interfaces. Enumeration order is incorporated into the class hierarchy by specifying different classes for enumerations that are unordered/increasing/decreasing etc. The bounds on the stored indices are conveyed to the compiler using a pragma.

In the running example of Figure 4, we will assume that the sparse lower triangular matrix `L` is stored in JAD format. Even though JAD is designed for fast enumeration along the long "diagonals", it is also possible to access the matrix rows through the indirection `iperm`. In our notation, this structure can be described by the expression $\mathit{perm}\{\texttt{iperm}[r'] \mapsto r : (r' \rightarrow c \rightarrow v) \oplus (\langle r', c\rangle \rightarrow v)\}$. Enumeration properties are used to tell the compiler that $r, r' \geq c$ and that when the $r' \rightarrow c \rightarrow v$ perspective is used, r' is random-access and c can be enumerated in increasing order. As shown in the Appendix, the JAD format can be implemented by the following classes, each implementing one fragment of the index structure expression.

- `Jad`: $\mathit{perm}\{\texttt{iperm}[r'] \mapsto r : \ldots\}$
- `JadPers`: $\ldots \oplus \ldots$
- `JadFlat`: $\langle r', c\rangle \rightarrow v$
- `JadHier`: $r' \rightarrow \ldots$
- `JadRow`: $c \rightarrow v$

Since `L` can be efficiently accessed either by "diagonal" or by row, and the code in Figure 4 accesses it by column, it is necessary to restructure this code to make it match JAD storage. The technology described in the rest of this paper accomplishes this.

3 Framework for Data-centric Restructuring

In this section, we sketch a *data-centric* framework for restructuring imperfectly-nested dense matrix codes with dependences. It extends the framework we developed in [1] for locality enhancement of dense matrix codes. For lack of space, we only sketch the ideas here; full details are available in [12].

Our framework makes the usual assumptions about programs: (i) programs are sequences of statements nested within loops, (ii) all memory accesses are through array references, and there is no array aliasing, and (iii) all loop bounds and array indices are affine functions of surrounding loop indices and symbolic constants.

We will use `S1`, `S2`, ..., `Sn` to name the statements in the program in syntactic order. An *instance* $\vec{i}_k$ of a statement `Sk` is the execution of statement `Sk` at iteration $\vec{i}_k$ of the surrounding loops. Flow-, anti-, and output-dependences from statement instances $\vec{i}_s$ to statement instances $\vec{i}_d$ can be expressed as matrix inequalities of the form $\mathcal{D} : D(\vec{i}_s, \vec{i}_d)^T + d \geq 0$ which we call *dependence classes* [1].

For our running example in Figure 4, it is easy to show that there are two relevant dependence classes.[3] The first dependence class $\mathcal{D}_1 = \{1 \leq j_1 \leq \texttt{N}, 1 \leq j_2 < i_2 \leq \texttt{N}, j_1 = j_2\}$ arises because statement `S1` writes to a location `b[j]` which is then read by statement `S2`; similarly, the second dependence class $\mathcal{D}_2 = \{1 \leq j_1 \leq \texttt{N}, 1 \leq j_2 < i_2 \leq \texttt{N}, j_1 = i_2\}$ arises because statement `S2` writes to location `b[i]` which is then read by reference `b[j]` in statement `S1`.

3.1 Modeling Program Transformations

We model program transformations as follows. We map dynamic instances of statements to points in a Cartesian space $\mathcal{P}$. We then enumerate the points in $\mathcal{P}$ in lexicographic order, and execute all statements mapped to a point when we enumerate that point. If there are more than one statement instances mapped to a point, we execute these statement instances in original program order. Intuitively, the Cartesian space $\mathcal{P}$ models a perfectly-nested loop, and the maps model transformations that embed individual statements into this perfectly nested loop. It should be understood that this perfectly-nested loop is merely a logical device—the code generation phase produces an imperfectly-nested loop from the space and the maps.

For example, if we chose $\mathcal{P} = j \times i$, we can embed the code in Figure 4 into $\mathcal{P}$ using the maps $F_1 = (j, j)$ for statement `S1`, and $F_2 = (j, i)$ for statement `S2`. This embedding preserves the original program order.

Clearly, not all spaces and maps correspond to legal transformations. However, if the execution order of the transformed program respects all dependences (i.e. for each

[3] There are other dependences, but they are redundant.

dependence, the source statement instance is enumerated and executed before the destination statement instance), then the resulting program is semantically equivalent to the original program. We must therefore address three problems.

1. *What is the Cartesian space $\mathcal{P}$ for the transformed program?*

   Each statement has an *iteration space* and a *data space*. The iteration space is a Cartesian space whose dimension is equal to the number of loops surrounding that statement. The data space is a Cartesian space whose dimensions are the dimensions of all references to arrays on which we might want to be data-centric. In our context, these are the references in the statement to sparse arrays. The *statement space* of a statement is the product of its iteration space and data space. We denote the statement space of statement Sk by $\mathcal{S}_k$. If we need to distinguish between the iteration space and data space dimensions of a statement instance $\vec{i}_k$, we will denote the iteration space coordinates by $\vec{it}_k$, and the data coordinates by $\vec{dt}_k$, *i.e.* $\vec{i}_k = (\vec{it}_k, \vec{dt}_k)$. A *product space* $\mathcal{P}$ for a program is the Cartesian product of its individual statement iteration spaces. For the purposes of this paper, the order in which individual dimensions appear in this product is left unspecified, and each order corresponds to a different product space.

   Data-centric code can be obtained by enumerating the data dimensions first.

   For the example of Figure 4, L is sparse, so the data space for S2 will have two dimensions corresponding to the row and column of L. The statement spaces for the two statements are $\mathcal{S}_1 = j_1 \times l_1^r \times l_1^c$ and $\mathcal{S}_2 = j_2 \times i_2 \times l_2^r \times l_2^c$, where the name of each dimension has been chosen to reflect its pedigree. A product space has 7 dimensions, and there are a total of $7!$ product spaces.

   Embedding the code in the product space $\mathcal{P} = l_1^r \times l_2^r \times l_1^c \times l_2^c \times j_1 \times j_2 \times i_2$ would result in data-centric code, as the data dimensions would be enumerated before the iteration space dimensions.

2. *How do we determine maps F_k to obtain a legal program?*

   We embed statement spaces into a product space using affine embedding functions $F_k : \mathcal{S}_k \rightarrow \mathcal{P}$. For each dependence with source $\vec{i}_s$ and destination $\vec{i}_d$, the source of the dependence is mapped to $F_s(\vec{i}_s)$, and the destination is mapped to $F_d(\vec{i}_d)$. If $F_d(\vec{i}_d)$ is enumerated after $F_s(\vec{i}_s)$, the restructured program preserves the execution order between the dependent statement instances $\vec{i}_s$ and $\vec{i}_d$. To guarantee that lexicographic enumeration of the points in the product space preserves the original program execution order, we require that $F_d(\vec{i}_d) - F_s(\vec{i}_s) \succeq \vec{0}$ for all dependence pairs $(\vec{i}_s, \vec{i}_d)$ in all dependence classes $\mathcal{D} : D(\vec{i}_s, \vec{i}_d)^T + d \geq 0$. As dependence classes are described by systems of linear inequalities, we can use Farkas' Lemma [9] to compute the set of all legal embedding functions, as we have demonstrated in [1].

   For our example, one possible pair of embedding functions is $F_1(j_1, l_1^r, l_1^c) = (l_1^r, l_1^r, l_1^c, l_1^c, j_1, j_1, j_1)^T$ and $F_2(j_2, i_2, l_2^r, l_2^c) = (l_2^r, l_2^r, l_2^c, l_2^c, j_2, j_2, i_2)^T$, which embed statements S1 and S2 into product space $\mathcal{P} = l_1^r \times l_2^r \times l_1^c \times l_2^c \times j_1 \times j_2 \times i_2$. For all dependence pairs $(\vec{i}_s, \vec{i}_d)$ in dependence class $\mathcal{D}_1$, we have $F_2(\vec{i}_d) - F_1(\vec{i}_s) = (+, +, 0, 0, 0, 0, +)^T$. This vector is lexicographically positive, therefore lexicographic enumeration of the points in the product space will enumerate the sources of dependences before the destinations of dependences, and program semantics will be preserved. Similarly, for all dependence pairs $(\vec{i}_s, \vec{i}_d)$ in dependence class $\mathcal{D}_2$, we have $F_1(\vec{i}_d) - F_2(\vec{i}_s) = (0, 0, +, +, +, +, 0)^T \succeq \vec{0}$, therefore all dependences are preserved.

3. *How do we evaluate the efficiency of each transformed program?*

   In the context of sparse matrix code generation, we answer this question in Section 4.2.

4 Accounting for Sparse Matrices

Data-centric code for sparse matrices must enumerate the co-ordinates appropriate to the sparse matrix format (e.g., the diagonal d and offset o for the DIA storage format in Figure 2) rather than the dimensions of the enveloping dense matrix. Therefore, we define the *sparse data space* of a statement, and use that instead of the (dense) data space described in Section 3 to define statement and product spaces.

The sparse data space of a statement is defined by starting with its dense data space and recursing over the index structure of sparse matrices referenced in that statement. Whenever a production rule *map*$\{F(in) \mapsto out : E\}$ is encountered, we remove the dimensions *out* from the data space and add dimensions *in* to it. For example, for the DIA compressed format, that means replacing the r and c dimensions with the d and o dimensions. The *perm*$\{P(in) \mapsto out : E\}$ rule does not change the dimensions of the data space.[4] If no sparse matrix in the program contains a production $E' \oplus E''$ or $E' \cup E''$, this defines the statement sparse data space uniquely.

The aggregation and perspective structures modify the product spaces of a program. Intuitively, if statement Sk references a matrix described by $E' \cup E''$ rule, we split Sk

[4] Permutations however change the order of enumeration of a dimension, that order may be important for legality and is handled by the code generation phase.

$$G = \begin{array}{c} \begin{array}{cccc} j_1 & j_2 & i_2 & \end{array} \\ \left[\begin{array}{c|cc} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{array}\right] \end{array} \begin{array}{l} l_1^r \\ l_2^r \\ l_1^c \\ l_2^c \\ j_1 \\ j_2 \\ i_2 \end{array}$$

Figure 7: Redundant Dimensions

```
for row = enumerate rows of L in increasing order
  for col = enumerate L[row][*] in increasing order
    val = L[row][col];
    if (row == col)
      b[col] = b[col]/val;
    if (col < row)
      b[row] = b[row] - val*b[col];
```

Figure 8: Row Data-centric Pseudocode for TS

into two copies: `Sk'` accessing the matrix through structures E', and `Sk"` accessing it through E''. The aggregation rule requires the statement to be executed for both structures E' and E'', so the resulting product spaces have dimensions $\mathcal{P} = \mathcal{S}_1 \times \cdots \times \mathcal{S'}_k \times \mathcal{S''}_k \times \cdots \times \mathcal{S}_n$. On the other hand, the perspective rule $E' \oplus E''$ presents a choice of access structure, which gives rise to two groups of product spaces, the first group with dimensions $\mathcal{P}' = \mathcal{S}_1 \times \cdots \times \mathcal{S'}_k \times \cdots \times \mathcal{S}_n$, and the second group with dimensions $\mathcal{P}'' = \mathcal{S}_1 \times \cdots \times \mathcal{S''}_k \times \cdots \times \mathcal{S}_n$.

In our running example, the perspective $E' \oplus E''$ production rule in the structure of the sparse matrix `L` tells the compiler that `L` can be accessed either by row, using $E' = (r' \rightarrow c \rightarrow v)$, or along "diagonals", using $E'' = (\langle r', c\rangle \rightarrow v)$. Since both statements `S1` and `S2` reference `L`, and there are two choices for each reference, the code in Figure 4 has four groups (of 7! each) of product spaces. All product spaces have the same set of dimensions $\{j_1, l_1^r, l_1^c, j_2, i_2, l_2^r, l_2^c\}$ although the order of dimensions and enumeration properties are different for different product spaces.

4.1 Generating Data-centric Code

We can think of a product space and embeddings as representing a perfectly-nested loop nest with guarded statements where we enumerate the values of all dimensions, and execute statement `Sk` when the values being enumerated match the embedding $F_k(\vec{i}_k)$. However, this code will have very poor performance. To improve performance, it is necessary to (i) identify and eliminate redundant dimensions, (ii) allow different directions of enumeration, and (iii) use common enumerations for related dimensions.

Redundant dimensions To give ourselves more flexibility to restructure the code, we introduced many dimensions in the product space. Now, after we have determined the embeddings, we can identify the dimensions we do not need and eliminate them. For our example, consider the (ordered) product space $\mathcal{P} = l_1^r \times l_2^r \times l_1^c \times l_2^c \times j_1 \times j_2 \times i_2$, and the embedding functions $F_1(j_1, l_1^r, l_1^c) = (l_1^r, l_1^r, l_1^c, l_1^c, j_1, j_1, j_1)^T$ and $F_2(j_2, i_2, l_2^r, l_2^c) = (l_2^r, l_2^r, l_2^c, l_2^c, j_2, j_2, i_2)^T$. As $l_1^r = l_1^c = j_1$, $l_2^r = i_2$ and $l_2^c = j_2$, the values of dimensions l_2^r, l_2^c, j_1, i_2 and j_2 of the product space are determined by the values of the preceding dimensions l_1^r and l_1^c. More generally, we identify redundant dimensions as follows.

Embedding functions are affine, and for each statement instance $\vec{i}_k = (\vec{it}_k, \vec{dt}_k)$, the data coordinates $\vec{dt}_k$ are affine functions of the loop indices $\vec{it}_k$. We can therefore represent the embedding functions as $F_k(\vec{i}_k) = G_k\vec{it}_k + \vec{g}_k$, where the matrix G_k defines the linear part of F_k, and the vector $\vec{g}_k$ is the affine part. We can use the matrix $G = [G_1 G_2 \ldots G_n]$ to identify redundant dimensions in the product space. We use G^k to refer to the k^{th} row of the matrix G. For our example, this matrix is shown in Figure 7.

If a row of the G matrix is a linear combination of preceding rows, the corresponding dimension of the product space is said to be *redundant*. In our example, only dimensions l_1^r and l_1^c are not redundant. It is not necessary to enumerate redundant dimensions since code is executed only for a single value in that dimension, and that value is determined by values of preceding dimensions, so we generate code to search for this value.

Enumeration Directions In Section 3 we said that the points in the product space are enumerated in lexicographic order. That requirement is reasonable for dense matrix codes, where we have random access to the matrix elements. Sparse matrices, however, may not support efficient enumeration along particular data dimension in increasing order. For example, enumerating a COO matrix in order of increasing row number would require a linear search for each element. As lexicographic enumeration of sparse data dimensions can be prohibitively expensive, we require it only when it is necessary for preserving dependences.

In our example, in order to not violate dependence class $\mathcal{D}_1$, the enumeration of dimension l_1^r must be in increasing order. However, the order in which we enumerate the remaining dimensions of the product space is irrelevant to the dependences in class $\mathcal{D}_1$—these dependences are already *satisfied* because of the lexicographic enumeration of dimension l_1^r. Similarly, dimension l_1^c must be enumerated in increasing order in order to satisfy dependence class $\mathcal{D}_2$. All other dimensions of the product space can be enumerated in arbitrary order.

In general, only some of the dimensions of the product space need to be enumerated in a particular direction in order to ensure legality. If the k^{th} dimension of the difference $F_d(\vec{i}_d) - F_s(\vec{i}_s)$ for some dependence class $\mathcal{D}$ is the first dimension with non-zero (*i.e.* positive) value, then dimension

```
template <>
void ts(Jad<double> &L, double b[])
{
    int m = L.columns();
    JadPers<double> LPers = L.subterm();
    JadHier<double> LHier = LPers.subterm2();

    for (int r=0; r<m; r++){
        JadHier<double>::iterator_type it_rr =
          search(LHier.begin(), LHier.end(), L.unmap(r));
        JadRow<double> Lrow = LHier.subterm(it_rr);
        for (JadRow<double>::iterator_type
                it_c = Lrow.begin();
             it_c != Lrow.end(); it_c++) {
            int c = *it_c;
            double v = Lrow.subterm(it_c);
            if (r > c) {
                b[r] -= b[c] * v;
            } else {
                b[r] = b[r] / v;
            }
        }
    }
}
```

Figure 9: Compiler-instantiated Code for TS

k of the product space must be enumerated in increasing order to satisfy dependence class $\mathcal{D}$.

Common enumerations An important optimization is recognizing groups of dimensions that could be enumerated together. In previous work [11], we developed technology for common enumeration of dimensions which are related through a single parametric variable (we called these *joinable* dimensions). We use common enumerations for groups of dimensions consisting of a non-redundant dimension, and redundant dimensions that immediately follow it and are linearly dependent on it. There are a number of ways of performing common enumerations which are closely related to join strategies in database systems such as merge-join and hash-join [11].

In the example, dimensions l_1^r and l_2^r are enumerated together, as are dimensions l_1^c and l_2^c. These common enumerations are trivial because they enumerate the same dimension of the same matrix. All iteration space dimensions are redundant and do not even need searches, as their values could be accessed directly.

The resulting data-centric pseudocode is shown in Figure 8. The important transformation that has happened is that matrix L is accessed by row to match the JAD format, while the original code in Figure 4 accessed it by column. This pseudocode is instantiated into the C++ code in Figure 9 in a straight-forward way.

4.2 Search Space and Cost Estimation

In theory, we can enumerate all legal enumeration-based codes as illustrated in Figure 10, then estimate the cost of each code, and select the best one. For the running example, that would involve (i) selecting one of the four groups of product spaces arising from the different ways to access

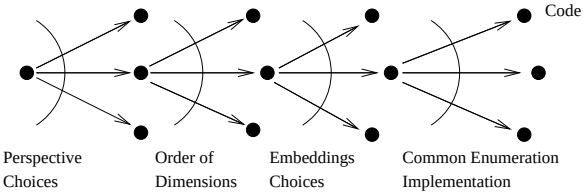

Figure 10: Search Space

L; (ii) selecting 1 of the possible 7! orders of the dimensions of the product space; (iii) chosing one set of embedding functions among the legal ones; and (iv) deciding how to combine the enumerations of L for the two accesses to it.

Figure 11 describes how we evaluate the cost of enumeration-based pseudocodes.[5] Each syntax rule is annotated with its associated cost. *EnumCost* depends on whether we are enumerating the dimension in a direction supported by the format, or whether dependences force us to enumerate in a different direction. *SearchCost* depends on the type of enumeration method available for that dimension (*e.g.*, whether it is an interval, or whether the values are sorted). *CommonEnumCost* depends on what common enumeration implementations are available for the corresponding data dimensions.

4.3 Heuristics to Limit the Search Space

Searching the full space of enumeration-based codes is impractical, but the following heuristics make the search space manageable.

Data-centric Execution Order: We only consider data-centric orders of dimensions of the product space (*i.e.*, orders in which all data dimensions come before any iteration space dimensions). In the running example for instance, we consider the product space $\mathcal{P} = l_1^r \times l_2^r \times l_1^c \times l_2^c \times j_1 \times j_2 \times i_2$ because all data dimensions are ordered before any iteration space dimensions, but we do not consider the product space $\mathcal{P} = l_1^r \times l_2^r \times l_1^c \times j_1 \times l_2^c \times j_2 \times i_2$ because an iteration space dimension (j_1 in this case) is ordered before a data space dimension (l_2^c in this case).

The indexing structure of sparse matrices puts further restrictions on the dimensions orderings we need to consider. For example, if L is accessed through the abstract structure $r' \rightarrow c \rightarrow v$ in statement S1, our compiler does not consider product spaces in which l_1^c is enumerated before l_1^r.

Common Enumerations: Efficient sparse code enumerates the data as few times as possible, so our goal is to use a single enumeration of a sparse matrix, and execute all statements which reference that matrix. That restricts our choice of embedding functions to just three per dimension: a common enumeration with a matching dimension of another statement, or, if that is not legal, embedding the statement

[5] The *guard* conditionals arise because of loop bounds.

$$
\begin{array}{lcll}
S & : & \texttt{for } i \in \texttt{enum}(\textit{iterator}) \texttt{ do } S & : \ \textit{EnumCost}(\textit{iterator}) * \textit{Cost}(S) \\
 & | & \texttt{for } i \in \texttt{enum}(\textit{itr}_1, \textit{itr}_2) \texttt{ do } S & : \ \textit{CommonEnumCost}(\textit{itr}_1, \textit{itr}_2) * \textit{Cost}(S) \\
 & | & \texttt{if } (i \in \texttt{search}(\textit{iterator})) \texttt{ then } S & : \ \textit{SearchCost}(\textit{iterator}) + \textit{Cost}(S) \\
 & | & \texttt{if } (\textit{guard}) \texttt{ then } x = y & : \ 1 \\
 & | & S_1; S_2 & : \ \textit{Cost}(S_1) + \textit{Cost}(S_2)
\end{array}
$$

Figure 11: Cost Estimation

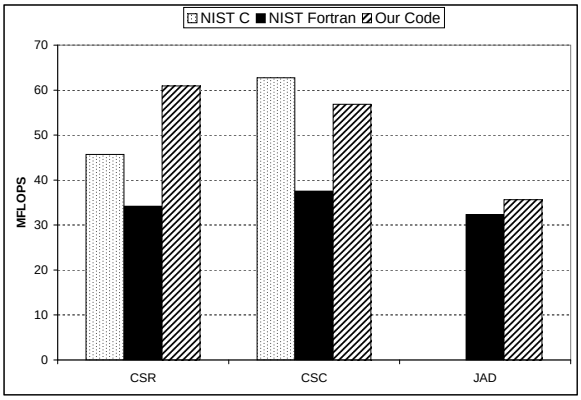

Figure 12: TS on SGI R12K

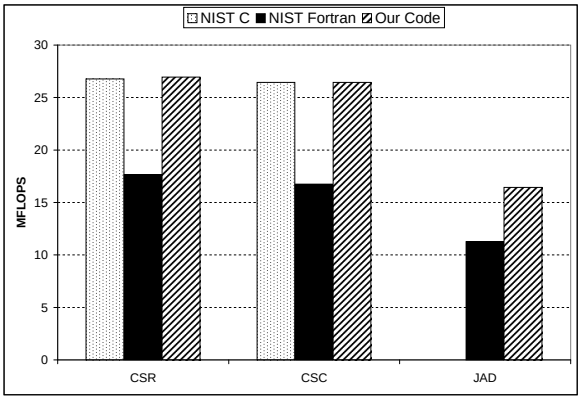

Figure 13: TS on Intel PII

before or *after* the enumeration of the matching dimension.

5 Experimental Results

We are implementing the algorithm presented in this paper in the Bernoulli Sparse Compiler. The generic programming system that we have implemented does not use `virtual` methods, as do the examples in this paper. Instead, we use the approach of Barton and Nackman [23] to ensure that all method invocations can be resolved to method definitions at compile time. See [13] for a more detailed discussion of the performance issues involved in implementing our system.

Here we present performance measurements for the running example (TS) for the CSR, CSC, and JAD formats, on an SGI Octane[6] and an Intel Pentium II[7] machines. We compared the NIST Sparse BLAS [10] library with code produced according to our algorithm with no further optimizations. The library provides Fortran and C implementations and supports 13 compressed formats. The more complicated formats such as JAD are not supported in the better optimized C implementation.

Our code is structurally equivalent to the one in the NIST C library. There are only minor syntactic differences, which result in small differences in performance between our code and the NIST C code on the SGI Octane. The NIST Fortran codes are less specialized (e.g, there is single code for a single or multiple right-hand sides), so they perform worse than both our code and the NIST C code.

Figures 12 and 13 present the performance on the matrix `can_1072` from the Harwell-Boeing collection [15]. The relative differences between the NIST codes and our code are representative for other inputs and benchmarks. These results indicate that the generic programming approach can successfully compete with hand-written library code.

6 Conclusions

We have presented a general framework that can be used for synthesizing sparse matrix codes from imperfectly-nested dense matrix codes and specifications of compressed formats. The compressed formats specification language is general enough to capture all sparse formats that we are aware of, and it also permits users to define new formats. Our code synthesis algorithm is able to exploit the index structure of sparse matrix formats and generate code competitive with hand-written library codes for the BLAS routines.

Automatic selection of sparse formats [3] for particular applications is an interesting extension to the work described here. One possibility is to make the compiler responsible for making this selection using cost estimation rules like the ones described in Section 4. Another possibility is to use an empirical optimization approach similar to that used in the ATLAS system [24] — the system generates code for a variety of promising formats, and determines experimentally

[6] 300MHz R12K processor, 2MB L2 cache, MIPSpro v.7.2 compiler, flags: -O3 -n32 -mips4.

[7] 300MHz, 512KB L2 cache, 256MB RAM, egcs-2.91.66 compiler, flags: -O3 -funroll-loops.

which one gives the best performance for the data sets of interest.

Synthesizing code competitive with handwritten code for matrix factorizations such as Cholesky or LU with pivoting remains an open problem. Handwritten codes for these routines [17, 6] use many special-purpose optimizations, and it is not clear how or even whether such optimizations should be incorporated into a general-purpose restructuring compiler system such as the one described in this paper.

Acknowledgements

We would like to thank the SC2000 referees and the Technical Papers Committee member Guang Gao for their valuable feedback which helped in improving the paper.

References

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000.

[2] Satish Balay, William Groopp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11 – Revision 2.0.15, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[3] Aart Bik and Harry Wijshoff. Automatic nonzero structure analysis. *SIAM Journal of Computing*, 28(5):1576–1587, 1999.

[4] Aart Bik and Harry A.G. Wijshoff. Compilation techniques for sparse matrix computations. In *Proceedings of the 1993 International Conference on Supercomputing*, pages 416–424, Tokyo, Japan, July 20–22, 1993.

[5] Aart Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31:14–24, 1995.

[6] J. Demmel, S. Eisenstat, J. Gilbert, X. Li, and J. Liu. A supernodal approach to sparse partial pivoting. Technical Report CSD-95-883, University of California, Berkeley, 1995.

[7] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[8] Iain S. Duff, Michele Marrone, Giuseppe Radicati, and Carlo Vittoli. A set of Level 3 Basic Linear Algebra Subprograms for sparse matrices. Technical Report RAL-TR-95-049, Computing and Information Systems Department, Atlas Centre, Rutherford Appleton Laboratory, Oxon OX11 0QX, England, September 11, 1995.

[9] Paul Feautrier. Some efficient solutions to the affine scheduling problem — part 1: one dimensional time. *International Journal of Parallel Programming*, October 1992.

[10] BLAS Technical Forum. Sparse BLAS library: Lite and toolkit level specifications, January 1997. `http://math.nist.gov/spblas/`.

[11] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Proceedings of EUROPAR*, 1997.

[12] Nikolay Mateev. *A Generic Programming System for Sparse Matrix Computations*. PhD thesis, Cornell University, 2000.

[13] Nikolay Mateev, Keshav Pingali, and Paul Stodghill. The Bernoully Generic Matrix Library. Technical Report TR2000-1808, Cornell University, Computer Science, August 2000.

[14] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000.

[15] Matrix Market home page, February 29, 2000. `http://math.nist.gov/MatrixMarket/`.

[16] David R. Musser and Alexander A. Stepanov. Generic programming. In *First International Joint Conference of ISSAC-88 and AAECC-6*, Rome, Italy, July 4-8, 1988. Appears in LNCS 358.

[17] E. Ng and B. Peyton. Block sparse Cholesky algorithms on advanced uniprocessor systems. *SIAM Journal of Scientific Computing*, 5, 1993.

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

[19] William Pugh and Tatiana Shpeisman. Generation of efficient code for sparse matrix computations. In *The Eleventh International Workshop on Languages and Compilers for Parallel Computing*, LNCS, Springer-Verlag, Chapel Hill, NC, August 1998.

[20] Yousef Saad. SPARSKIT: a basic tool kit for sparse matrix computations, version 2, June 1994. `http://www.cs.umn.edu/Research/arpa/SPARSKIT/`.

[21] Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE '98*, 1998.

[22] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 and 2. Computer Science Press, Rockville, MD, 1988.

[23] Todd Veldhuizen. Techniques for scientific C++, version 0.3, August 1999. `http://extreme.indiana.edu/~tveldhui/papers/techniques/`.

[24] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software (ATLAS) home page, July 28, 2000. `http://www.netlib.org/atlas/`.

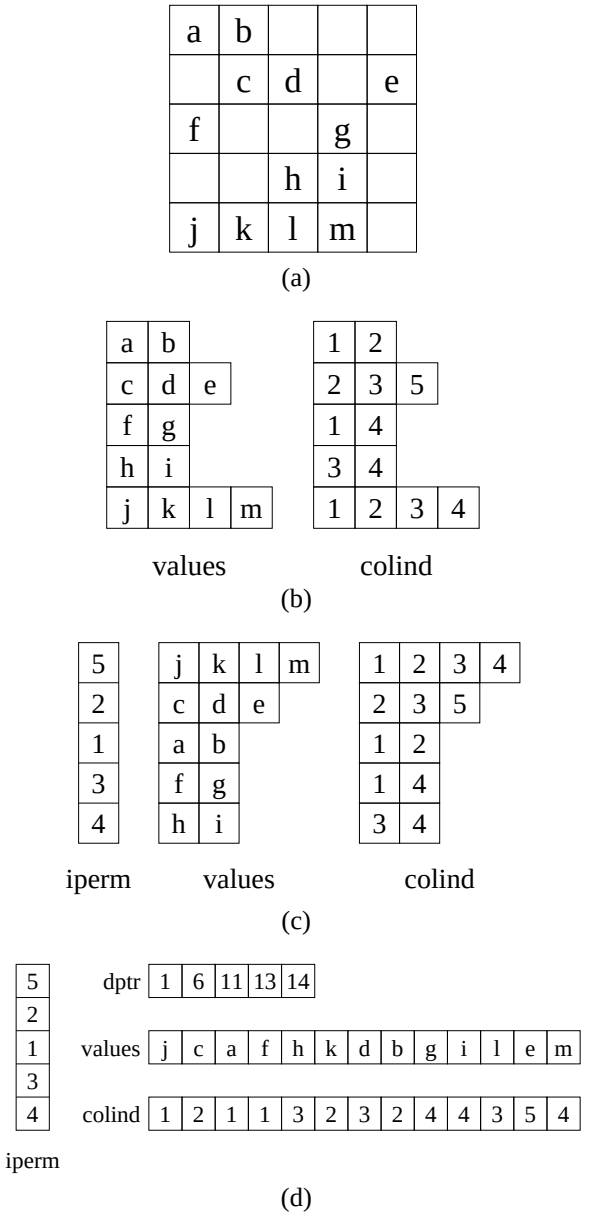

Figure 14: Building JAD Storage

A Jagged Diagonal Compressed Format

In this appendix, we present details of the Jagged Diagonal (JAD) format that was used as an example in this paper.

An instance of a JAD matrix may be constructed as follows. First, the rows of the matrix, as in Figure 14(a), are "compressed" so that zero elements are eliminated. This requires introducing an auxiliary array, `colind`, to maintain the original column indices. This is shown in Figure 14(b). Next, the rows of the compressed matrix are sorted by the number of non-zeros within each row in *decreasing* order. This requires introducing a permutation vector, `iperm`, as shown in Figure 14(c). Finally, the columns of the compressed and sorted matrix, which are called the "diagonals", are stored contiguously in two vectors, `colind` and `values`. The vector `dptr` is used to record the first index of the entries of each diagonal within `colind` and `values`. The final storage is shown in Figure 14(d).

The non-zero entries of a matrix in JAD format can be enumerated quickly and efficiently by enumerating the values of `colind` and `values`. In addition, if the program can be restructured to work with the permuted row indices instead of the row indices, then efficient row-oriented access can be provide as well. This is necessary for such computations as triangular solve, which place certain constraints on the order in which elements may be enumerated.

A.1 High-level API for JAD

The high-level API presents a dense-matrix view of the sparse matrix and is used by the algorithm designer.

The structure `JadStorage` is used to hold all of the components of the JAD storage within a single object. For each matrix in the JAD format there will be a single instance of this class which maintains the storage for that matrix. All other classes in the JAD implementation keep a reference to this instance.

```
//////////////////////////////////////////////////////
//                      JadStorage                  //
//////////////////////////////////////////////////////

template<class BASE>
struct JadStorage {
public:
    vector<int> *iperm;
    vector<int> *dptr;
    vector<int> *colind;
    vector<BASE> *values;
    const int n;
    const int nd;
    const int nz;

    JadStorage(vector<int> *_iperm, vector<int> *_dptr,
               vector<int> *_colind,
               vector<BASE> *_values)
        : iperm(_iperm), dptr(_dptr), colind(_colind),
          values(_values), n(iperm->size()),
          nd(dptr->size()-1), nz(colind->size()) {
    }
};
```

The JadRandom class inherits from the matrix abstract class and implements the random access interface for the matrix by implementing the get and set abstract methods. The method ref within this class is responsible for finding a particular (r, c) entry within the matrix. It does this by first finding the corresponding row within the permuted index space, and then performing a linear search within the row for the given column index. A binary search could be used, if it were assumed that entries within a row were always sorted by column index.

```
//////////////////////////////////////////////////////
//                      JadRandom                   //
//////////////////////////////////////////////////////

template <class BASE>
class JadRandom : public matrix<BASE> {
protected:
    JadStorage<BASE> *A;

public:
    JadRandom(int m, int n, JadStorage<BASE> *A)
        : matrix<BASE>(m,n), A(A) { }

    virtual ~JadRandom() { }

    BASE *ref (int r, int c) {
        int rr = -1;
        for (rr=0; rr<A->n; rr++)
            if ((*A->iperm)[rr] == r) break;
        assert(rr != A->n);

        for (int d=0; d<A->nd; d++) {
            int jj_lo = (*A->dptr)[d];
            int jj_hi = (*A->dptr)[d+1];
            int jj = jj_lo + rr;
            if (jj >= jj_hi) break;
            if ((*A->colind)[jj] == c)
                return &(*A->values)[jj];
        }
        return 0;
    }
    virtual BASE get(int r, int c) {
        BASE *p = ref(r,c);
        if (p) { return *p; }
        else { return 0; }
    }
    virtual void set(int r, int c, BASE v) {
        BASE *p = ref(r,c);
        assert(p);
        *p = v;
    }
};
```

A.2 Low-level API for JAD

The low-level API presents the index structure of the format to the restructuring compiler.

Using the grammar presented in Section 2, the following view can be used to describe the index structure of the JAD format.

$$map\{\texttt{iperm}[rr] \mapsto r : ((< rr, c > \rightarrow v) \oplus (rr \rightarrow c \rightarrow v))\}$$

The following classes implement the different pieces of the view.

- Jad: $map\{\texttt{iperm}[rr] \mapsto r : \ldots\}$
- JadPers: $\ldots \oplus \ldots$
- JadFlat: $< rr, c > \rightarrow v$
- JadHier: $rr \rightarrow \ldots$
- JadRow: $c \rightarrow v$

We present the classes "inside-out".

The classes JadFlat and JadFlatIterator implement the view of the JAD format that is appropriate for fast enumeration. As its view suggests, this implementation is very similar to the implementation of co-ordinate storage presented earlier in the paper. The difference is that, with the JAD format, the row index is not stored with each entry, and must be computed on the fly. This is done in method JadFlatIterator::operator *.

```
//////////////////////////////////////////////////////
//                      JadFlat                     //
//////////////////////////////////////////////////////

template<class BASE> class JadFlatIterator;

template<class BASE>
class JadFlat
    : public term_nesting< JadFlatIterator<BASE>,
                           term_scalar<BASE> >
{
protected:
    JadStorage<BASE> *A;
public:
    JadFlat(JadStorage<BASE> *A) : A(A) { }
    virtual iterator_type begin()
        { return JadFlatIterator<BASE>(A,0); }
    virtual iterator_type end()
        { return JadFlatIterator<BASE>(A,A->nz); }
    virtual subterm_type subterm(iterator_type it) {
        return (*A->values)[it.jj]; }
};

//////////////////////////////////////////////////////
//                  JadFlatIterator                 //
//////////////////////////////////////////////////////

template<class BASE>
class JadFlatIterator :
    public increasing_iterator<pair<int,int> > {
    friend class JadFlat<BASE>;
protected:
    JadStorage<BASE> *A; int jj; int d;
    void frob_d() { if (jj == (*A->dptr)[d+1]) d++; }
public:
    JadFlatIterator(JadStorage<BASE> *A, int jj)
        : A(A), jj(jj), d(0) { }
    virtual void operator ++(int) { jj++; frob_d(); }
    virtual key_type operator *() {
        return
            make_pair(jj-(*A->dptr)[d],(*A->colind)[jj]);
    }
    virtual bool equal(
        const proto_iterator<pair<int,int> > &y) const
        { return jj ==
            dynamic_cast<const JadFlatIterator &>(y).jj;}
};
```

The JadHier, JadRow and JadRowIterator classes provide row-oriented access to the JAD format. The JadHier class provides access to the rows within the permuted row index space. The JadRow and JadRowIterator classes provide access to the non-zero elements within each row accessed via JadHier.

```
////////////////////////////////////////////////////////
//                        JadHier                     //
////////////////////////////////////////////////////////

template<class BASE> class JadRow;
template<class BASE> class JadRowIterator;

template<class BASE>
class JadHier
    : public term_nesting< interval_iterator<int>,
                           JadRow<BASE> >
{
protected:
    JadStorage<BASE> *A;
public:
    JadHier(JadStorage<BASE> *A) : A(A) { }
    virtual iterator_type begin()
        { return interval_iterator<int>(0); }
    virtual iterator_type end()
        { return interval_iterator<int>(A->n); }
    virtual subterm_type subterm(iterator_type it) {
        return JadRow<BASE>(A,*it); }
};

////////////////////////////////////////////////////////
//                        JadRow                      //
////////////////////////////////////////////////////////

template<class BASE>
class JadRow
    : public term_nesting< JadRowIterator<BASE>,
                           term_scalar<BASE> >
{

protected:
    JadStorage<BASE> *A; int r; int dmax;

public:
    JadRow(JadStorage<BASE> *A, int r) : A(A), r(r) {
        for (dmax = 0;
             dmax < A->nd-1 &&
                 r < (*A->dptr)[dmax+1]-(*A->dptr)[dmax];
             dmax++)
            ;
    }
    virtual iterator_type begin() {
        return JadRowIterator<BASE>(A,r,0); }
    virtual iterator_type end() {
        return JadRowIterator<BASE>(A,r,dmax); }
    virtual subterm_type subterm(iterator_type it) {
        return (*A->values)[(*A->dptr)[it.d]+r]; }
};

////////////////////////////////////////////////////////
//                    JadRowIterator                  //
////////////////////////////////////////////////////////

template<class BASE>
class JadRowIterator :
    public increasing_iterator<int> {
    friend class JadRow<BASE>;
protected:
    JadStorage<BASE> *A; int r; int d;
public:
    JadRowIterator(JadStorage<BASE> *A, int r, int d)
        : A(A), r(r), d(d) { }
    virtual void operator ++(int) { d++; }
    virtual key_type operator*() {
        return (*A->colind)[(*A->dptr)[d]+r]; }
    virtual bool equal(const
                       proto_iterator<int> &y) const
        { return
              r == dynamic_cast<const
                       JadRowIterator &>(y).r
              && d == dynamic_cast<const
                          JadRowIterator &>(y).d; }
};
```

The class `JadPers` simply wraps the `JadFlat` and `JadHier` classes together with $\oplus$, the perspective operator.

```
////////////////////////////////////////////////////////
//                        JadPers                     //
////////////////////////////////////////////////////////

template<class BASE>
class JadPers
    : public term_perspective2< JadFlat<BASE>,
                                JadHier<BASE> >
{
protected:
    JadStorage<BASE> *A;
public:
    JadPers(JadStorage<BASE> *A) : A(A) { }
    virtual subterm1_type subterm1() {
        return JadFlat<BASE>(A); }
    virtual subterm2_type subterm2() {
        return JadHier<BASE>(A); }
};
```

The top-most level of the JAD's view is the map operator that describes the permutation. The interface class `term_perm2` refines the general `term_map` class. It takes two template parameters, `Pr` and `Pc`, which are the permutations used on the row and column indices, respectively.

```
////////////////////////////////////////////////////////
//                      term_perm2                    //
////////////////////////////////////////////////////////

template<class Pr, class Pc, class E>
class term_perm2
    : public term_map< pair<int,int>, E >
{
public:
    Pr pr; Pc pc;
    term_perm2() { }
    term_perm2(const Pr &pr, const Pc &pc)
        : pr(pr), pc(pc) { }
    virtual pair<int,int> map(pair<int,int> x) {
        return make_pair(pr.apply(x.first),
                         pc.apply(x.second));
    }
    virtual pair<int,int> unmap(pair<int,int> x) {
        return make_pair(pr.unapply(x.first),
                         pc.unapply(x.second));
    }
};
```

The classes `term_perm_ident` (representing identity permutation) and `term_perm_vector` (permutation vector) are used as the `Pr` and `Pc` arguments to `term_perm2`.

```
////////////////////////////////////////////////////////
//                   term_perm_ident                  //
////////////////////////////////////////////////////////

class term_perm_ident {
public:
    term_perm_ident() { }
    int apply(int x) { return x; }
    int unapply(int x) { return x; }
};

////////////////////////////////////////////////////////
//                   term_perm_vector                 //
////////////////////////////////////////////////////////
class term_perm_vector {
public:
    vector<int> *perm;
    term_perm_vector() : perm(0) { }
    term_perm_vector(vector<int> *perm) : perm(perm) { }
    int apply(int ii) { return (*perm)[ii]; }
    int unapply(int ii) {
        for (int i=0; i<(*perm).size(); i++)
            if ((*perm)[i] == ii) return i;
        assert(false);
    }
};
```

The top class of the JAD format is `Jad`, and it provides the implementation of the row permutation. This is indicated by inheriting from the `term_perm2` interface class, instantiated for the row index with `term_perm_vector`, and with `term_perm_ident` for the column index. The vector `iperm` is used to initialize the instance of `term_perm_vector`.

```
//////////////////////////////////////////////////////
//                        Jad                       //
//////////////////////////////////////////////////////

template<class BASE>
class Jad
    : public JadRandom<BASE>,
      public term_perm2< term_perm_vector,
                         term_perm_ident,
                         JadPers<BASE> >
{
public:
    Jad(int m,int n, JadStorage<BASE> *A)
        : JadRandom<BASE>(m,n,A),
          term_perm2< term_perm_vector, term_perm_ident,
                      JadPers<BASE> >(
                          term_perm_vector(A->iperm),
                          term_perm_ident()) {}
    virtual subterm_type subterm() {
        return JadPers<BASE>(A); }
};
```