Compiling Imperfectly-nested Sparse Matrix Codes with Dependences

Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. We present compiler technology for generating sparse matrix code from (i) dense matrix code and (ii) a description of the indexing structure of the sparse matrices. This technology embeds statement instances into a Cartesian product of statement iteration and data spaces, and produces efficient sparse code by identifying common enumerations for multiple references to sparse matrices. This approach works for imperfectly-nested codes with dependences, and produces sparse code competitive with hand-written library code.

1 Introduction

Sparse matrices are usually stored in *compressed formats* in which zeros are not stored explicitly [9]. 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 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. This format organizes the non-zeros of a sparse matrix into a small number of very long “diagonals”. 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, elements are addressed by diagonal and offset.

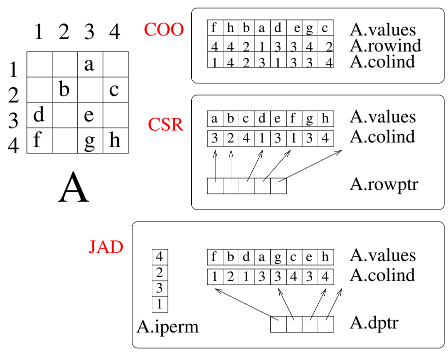

Fig. 1. Sparse Storage Formats

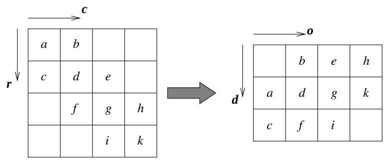

Fig. 2. DIA Storage Format

```
     for j = 1,N
S1:    b[j] = b[j]/L[j,j];
       for i = j+1,N
S2:      b[i] = b[i] - L[i,j]*b[j];
```

Fig. 3. Column Triangular Solve

For dense matrices, highly efficient implementations of the Basic Linear Algebra Subroutines (BLAS) [3] are usually provided by hardware vendors. For sparse matrices, the problem of developing BLAS libraries is complicated by the fact that some forty or fifty compressed formats are used widely, and each format requires customized code. Many attempts at writing sparse BLAS libraries have been confounded by the code explosion problem [11, 4].

Bik and Wijshoff [2] proposed using 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. However, their system is not extensible in the sense that the programmer cannot specify a new format, and modern sparse formats such as JAD format cannot be supported. Pugh and Shpeisman [10] propose an intermediate program representation for sparse codes that allows them to predict asymptotic program efficiency and make decisions about chosing sparse matrix formats.

In our previous work [7], we argued that (i) sparse matrices should be viewed as *sequential-access* data structures [13], and (ii) efficient sparse codes should be organized if possible as *data-centric computations* that enumerate non-zero 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 have an indexing structure and should therefore be viewed as *indexed-sequential-access* structures [13]. 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* [8]. The algorithm is programmed abstractly just once in a data-structure-neutral fashion, and concrete programs are obtained by instantiating this abstract code with different data structure implementations. The most well-known example of this approach is the Standard Template Library (STL) in C++. The MTL [12] is a generic library of matrix computations.

$$E: \; \mathit{Idx} \to E \;|\; \mathit{map}\{F(\mathit{in}) \mapsto \mathit{out} : E\} \;|\; \mathit{perm}\{P(\mathit{in}) \mapsto \mathit{out} : E\} \;|\; E \oplus E \;|\; E \cup E \;|\; v$$
$$\mathit{Idx} : \mathit{attribute} \;|\; \langle \mathit{attribute}, \ldots, \mathit{attribute} \rangle \;|\; (\mathit{attribute} \times \cdots \times \mathit{attribute})$$

Fig. 4. Sparse Matrix Abstraction

In our system, generic programs are dense matrix programs, and they are "instantiated" into efficient sparse matrix programs by our restructuring compiler when it is supplied with specifications of sparse formats. Of course, this instantiation mechanism is considerably more complex than the C++ template instantiation mechanism since it is necessary to restructure the dense code at a deep level to make it data-centric for the desired sparse format. Previously, we showed how this restructuring could be done if the program is a perfectly-nested loop nest in which iterations can be executed in any order [6]. However, many codes of interest, such as the triangular solve in Figure 3 and matrix factorizations, are not perfectly-nested and data dependences do not allow executing statements in arbitrary order. We address this problem in this paper.

The rest of the paper is organized as follows. In Section 2, we sketch how the user can specify sparse matrix formats in our generic programming system. In Section 3, we give an outline of a restructuring framework that we developed for imperfectly-nested loops computing with dense matrices [1]. In Section 4, we discuss how this framework can be used for synthesizing sparse matrix code from dense matrix code and sparse format descriptions. The key ingredients in our solution are a search space of sparse programs, a cost metric for evaluating the quality of a sparse program, and heuristics to restrict the search space. In Section 5, we present experimental results demonstrating that our approach produces code competitive with hand-optimized sparse matrix libraries. 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. For lack of space, we will focus on how this is specified in our system, and omit other details of the generic programming system which can be found in a previous technical report [7].

To appreciate the importance of exploiting the index structure in code restructuring, consider the triangular solve code of Figure 3. 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 from this dense code. 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 4 is used to describe the index structure of a sparse matrix to our system [7]. The most important rule for specifying index structure is the $\mathit{Idx} \to E$ production rule. For example, a CSR matrix is described as $r \to c \to v$, indicating that rows must be accessed first, and within each row, elements within columns can be enumerated. The *map*$\{F(in) \mapsto out : E\}$ and *perm*$\{P(in) \mapsto out : E\}$ rules are used to describe linear and permutation transformations on the matrix indices. A matrix in DIA storage format can be described as *map*$\{d + o \mapsto r, o \mapsto c : d \to o \to v\}$, while the *perm* operator is useful for describing formats like JAD. 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 $E' \cup E''$ (*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''.

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

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

- *Enumeration order*: a description of the order in which coordinate values could 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 $1 \leq c \leq r \leq \mathtt{N}$.

In addition to specifying this index structure, the sparse format designer must write the actual code to perform these enumerations. We omit details of this since it is not relevant to the rest of the paper.

In the running example of Figure 3, 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 *perm*$\{\mathtt{iperm}[r'] \mapsto r : (r' \to c \to v) \oplus (\langle r', c\rangle \to v)\}$. Enumeration properties are used to tell the compiler that $r, r' \geq c$ and that when the $r' \to c \to v$ perspective is used, r' is random-access and c can be enumerated in increasing order. Since `L` can be efficiently accessed either by "diagonal" or by row and the code in Figure 3 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 summarize a *data-centric* framework for restructuring imperfectly-nested dense matrix codes with dependences; details can be found in an associated technical report [1]. In Section 4, we adapt this framework for sparse matrices.

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* i_k of a statement Sk is the execution of statement Sk at iteration i_k of the surrounding loops. We say that there exists a *data dependence* from instance i_s of statement Ss (the *source* of the dependence) to instance i_d of statement Sd (the *destination*) if (i) both instances lie within corresponding loop bounds; (ii) they reference the same memory location; (iii) at least one of them writes to that location; and (iv) instance i_s of statement Ss occurs before instance i_d of statement Sd in program execution order. Dependence constraints can be represented as a matrix inequality of the form $D(i_s, i_d)^T + d \geq 0$. Such an inequality obviously represents a polyhedron. Each such matrix inequality will be called a *dependence class*, and will be denoted by $\mathcal{D}$ with some subscript.

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

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 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.

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 $\mathcal{S}_k$, and the coordinates of instance i_k in $\mathcal{S}_k$ by (i_k, d_k). A *product space* $\mathcal{P}$ for a program is the Cartesian product of its individual statement iteration spaces. For the purposes

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

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.

How do we determine maps F_i to obtain a legal program? We embed statement spaces into a product space using affine embedding functions $F_k : \mathcal{S}_k \rightarrow \mathcal{P}$. Let $F_{k,m}$ denote the dimensions of F_k corresponding to dimensions derived from statement `Sm`, *i.e.* $F_{k,m} : \mathcal{S}_k \rightarrow \mathcal{S}_m$. To keeps matters simple, we only consider embedding functions for which $F_{k,k}$ is identity mapping. As dependence classes are described by systems of linear inequalities, we can use Farkas' Lemma to compute the set of all legal embedding functions. Details are available in [1].

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.

For the example of Figure 3, `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. Among the legal embedding functions are $F_1(j_1, l_1^r, l_1^c) = (j_1, l_1^r, l_1^c, j_1, j_1, l_1^r, l_1^c)^T$, $F_2(j_2, i_2, l_2^r, l_2^c) = (j_2, l_2^r, l_2^c, j_2, i_2, l_2^r, l_2^c)^T$, which embed `S1` and `S2` in $\mathcal{P} = j_1 \times l_1^r \times l_1^c \times j_2 \times i_2 \times l_2^r \times l_2^c$.

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 *out* from the data space and add *in* to it. The *perm*$\{P(in) \mapsto out : E\}$ rule does not change the dimensions of the data space.[2] 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 $E' \cup E''$ or $E' \oplus E''$ rule, we split `Sk` 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 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

[2] 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.

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

Fig. 5. Redundant Dimensions

```
for r = enum(iterator_r) (increasing) do
  for c = enum(iterator_c) (increasing) do
    v = currently enumerated value of L
    if (r=c) then b[c] = b[c]/v;
    if (r>c) then b[r] = b[r] - v*b[c];
```

Fig. 6. Data-centric Triangular Solve

$E' = (r' \to c \to v)$, or along "diagonals", using $E'' = (\langle r', c\rangle \to v)$. Since both statements `S1` and `S2` reference `L`, and there are two choices for each reference, the code in Figure 3 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 $F_k(i_k, d_k)$. However, this code will have very poor performance. To improve performance, it is necessary to (i) identify and eliminate redundant dimensions, and (ii) use common enumerations for related dimensions. We illustrate these points with 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 $F_2(j_2, i_2, l_2^r, l_2^c) = (l_2^r, l_2^r, l_2^c, l_2^c, j_2, j_2, i_2)^T$, which embed statements `S1` and `S2` into product space $\mathcal{P} = l_1^r \times l_2^r \times l_1^c \times l_2^c \times j_1 \times j_2 \times i_2$.

All embedding functions are affine, and for each statement instance (i_k, d_k), the data coordinates d_k are affine functions of the loop indices i_k, so we can represent the embedding functions as $F_k(i_k, d_k) = G_k i_k + g_k$, where the matrix G_k defines the linear part of F_k, and the vector 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 5.

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.

Some dimensions must be enumerated in a particular direction in order to ensure legality. If the k^{th} dimension of the difference $F_d(i_d, d_d) - F_s(i_s, d_s)$ for some dependence class $\mathcal{D}$ is the first dimension with non-zero (*i.e.* positive) value, then dimension k of the product space must be enumerated in increasing order to satisfy dependence class $\mathcal{D}$. 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. Similarly, dimension l_1^c must be enumerated in increasing order because of dependence class $\mathcal{D}_2$. All other dimensions of the product space can be enumerated in arbitrary order.

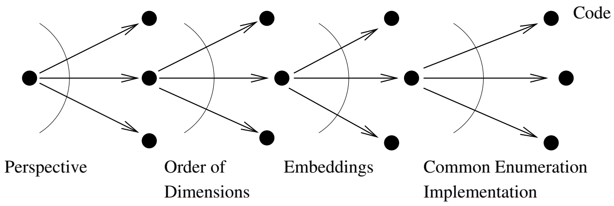

Fig. 7. Search Space

An important optimization is recognizing groups of dimensions that could be enumerated together. In previous work [6], 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 [6].

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 code is shown in Figure 6.

4.2 Search Space and Cost Estimation

We can enumerate all legal enumeration-based codes as illustrated in Figure 7. The syntax of the code is described by the following grammar. The *guard* conditionals arise because of loop bounds.

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

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.

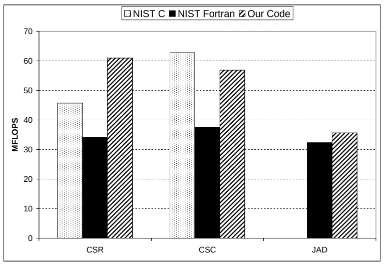

Fig. 8. TS on SGI R12K

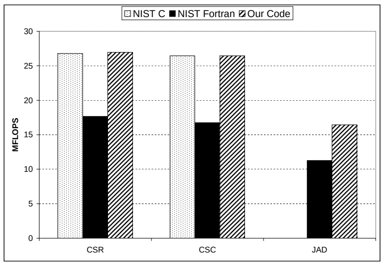

Fig. 9. TS on Intel PII

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). 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 $r' \rightarrow c \rightarrow v$, our compiler does not consider product spaces in which c is enumerated before 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 *before* or *after* the enumeration of the matching dimension.

5 Experimental Results

We have implemented the algorithm presented in this paper in the Bernoulli Sparse Compiler. Here we present performance measurements on an SGI Octane[3] and an Intel Pentium II[4] machines. We compared the code produced by our algorithm with the NIST Sparse BLAS [4] implementations of triangular solve for the CSR, CSC, and JAD sparse formats. Sparse BLAS supports 13 compressed formats. A complete Fortran implementation, as well as a better optimized but incomplete implementation in C, are available. The more complicated formats such as JAD are not supported in the optimized C implementation.

Figure 8 presents the performance of the hand-written NIST C (grey bars) and Fortran (black bars) codes, and the code produced by our algorithm (striped bars). As input we used the matrix `can_1072` from the Harwell-Boeing collection[5]. These results clearly show that the generic programming approach can successfully compete with hand-written library code. Indeed, the performance of our code ranges between 90% and 133% of NIST's C implementation and between 110% and 178% of NIST's Fortran implementation on the R12K. On the Pentium II, our code's performance is prac-

[3] 300MHz R12K processor, 2MB L2 cache, MIPSpro v.7.2 compiler, flags: -O3 -n32 -mips4.
[4] 300MHz, 512KB L2 cache, 256MB RAM, egcs-2.91.66 compiler, flags: -O3 -funroll-loops.
[5] http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/

tically identical with NIST's C implementation and outperforms the NIST's Fortran implementation by about 50%.

6 Conclusions

We have presented a general framework that can be used for modeling execution and restructuring of both sparse and dense imperfectly-nested matrix codes with dependences. We have used this framework to develop an algorithm for synthesizing sparse matrix code from dense matrix code and a specification of sparse matrix formats. The specification language is general enough to capture all sparse formats that we are aware of, and supports user-defined data structures. However, this generality does not come at the expense of performance. Our algorithm is able to exploit the indexing structure of sparse matrix formats and generate code competitive with hand-written library codes.

In this paper we only discussed sequential sparse matrix code generation. In [5] we have investigated the generation of *parallel* sparse matrix code for perfectly-nested loops with no dependences, and we are working on combining the two techniques.

References

1. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loops. Technical Report TR2000-1782, Cornell University, Computer Science, January 2000.
2. 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.
3. 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.
4. BLAS Technical Forum. Sparse BLAS library: Lite and toolkit level specifications, January 1997. Editted by Roldan Pozo and Micheal A. Heroux and Karin A. Remington.
5. Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel code for sparse matrix applications. In *Supercomputing '97*, San Jose, November 15–21, 1997.
6. Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. A relational approach to the compilation of sparse matrix programs. In *Proceedings of EUROPAR*, 1997.
7. Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. A generic programming system for sparse matrix computations. Technical Report TR99-1761, Cornell University, Computer Science, August 1999.
8. 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.
9. Sergio Pissanetsky. *Sparse Matrix Technology*. Academic Press, London, 1984.
10. 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.
11. Yousef Saad. SPARSKIT version 2.0.
12. Jeremy G. Siek and Andrew Lumsdaine. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE '98*, 1998.
13. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1 and 2. Computer Science Press, Rockville, MD, 1988.