A Relational Approach to the Compilation of Sparse Matrix Programs

Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill

Computer Science Department, Cornell University
Ithaca, NY 14853, USA
{*vladimir,pingali,stodghil*}@*@cs.cornell.edu*

Abstract. We present a relational algebra based framework for compiling efficient sparse matrix code from dense DO-ANY loops and a specification of the representation of the sparse matrix. We present experimental data that demonstrates that the code generated by our compiler achieves performance competitive with that of hand-written codes for important computational kernels.

1 Introduction

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

One possibility is to give the compiler dense matrix programs in which some matrices are declared to be sparse, and make the compiler responsible for choosing appropriate storage formats and for generating sparse matrix programs. This idea has been explored by Bik and Wijshoff [2–4], but their approach is limited to simple sparse matrix formats (so-called Compressed Hyperplane Storage) that are not representative of those used in high-performance codes.

In this paper, we focus on the problem of generating efficient sparse code, *given user-specified storage formats*. Our approach is motivated by the fact that sparse matrix formats are highly application and architecture dependent. For example, sparse matrices that arise in the solution of PDEs with many degrees of freedom usually have groups of rows with identical non-zero structure [8]. This property is exploited by the BlockSolve library from Argonne [9]. A simplification of this format is illustrated in Fig. 1. A set of rows with identical structure is called an *inode*. The non-zero values in the rows of an inode can be stored as dense matrices, and the row and column indices are stored just once. The presence of dense blocks in this format makes products of a BlockSolve matrix with a dense vector or a dense matrix "rich" in BLAS-2 and BLAS-3 routines which can be performed efficiently on modern architectures. Without having application-specific knowledge, a compiler could not decide that this representation is profitable.

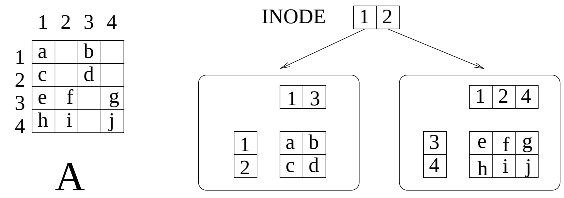

Fig. 1. Simplified BlockSolve format

1.1 Classifying sparse matrix codes

Sparse matrix codes can be classified according to two characteristics: whether there are dependencies in the original dense loop and whether insertions of non-zeroes (called *fill*) occur during execution of the loop. This is shown in Table 1 [1]. In this paper, we focus on DO-ANY loops, which are loops in which iterations can be done in any order. These loops can contain reductions, but are not allowed to have general loop-carried dependences. Focusing on DO-ANY loops allows us to address the needs of many important applications, such as iterative methods for the solution of linear systems (without incomplete factor preconditioners) [12]. The main computations in these algorithms are products of large sparse matrices with dense vectors or dense "skinny" matrices, and vector operations such as dot product, vector scaling and addition.

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

Table 1. Classification of sparse matrix codes

1.2 Our approach

There are two optimization goals in the translation of dense programs into sparse programs: avoiding useless computation and avoiding searching sparse data structures. Both of these goals are achieved by generating code that walks

[1] "MVM" = "matrix-vector multiplication; "MMM" = "matrix-matrix multiplication

over the data structures. The main idea of the work done by Bik and Wijshoff [2–4] is to transform the loop nest so that the *inner* loop index accesses a single dimension of several data structures. The inner loop is transformed to enumerate over one of the data structures, and searches are generated for the rest. The cost of searches is reduced, whenever possible, by scattering some sparse data structures into dense arrays.

The key to our approach is discovering when the data structures can be enumerated together, or *joined*, and then generating efficient code for these *joins*. Instead of a common enumeration in the inner loop, we attempt to discover common enumerations at *all levels* in the loop nest. To this end, we have adopted techniques from relational database query optimization. We view arrays (sparse and dense) as relations which contain both zeroes and non-zeroes. Only non-zeroes are stored explicitly, while zero elements are implicit. We view input loop nests as queries that select a certain set of tuples from a cross product of all relations. Just as is the case in the database query optimization, we combine the selections and cross products to arrive at equi-joins. However, unlike database relations, which are "flat", most sparse matrix formats have a hierarchical structure which must be exploited for efficiency. We require that the programmer provide a description of this structure, using *access methods*. Based on these access methods, we decide how the joins should be ordered and implemented. Finally, code is generated using the implementations of the access methods, provided by the programmer.

We illustrate these points, using the following loop which computes a matrix-vector product (SMVM) of a sparse matrix (A) and a sparse vector (X):

DO $i = 1, N$
    DO $j = 1, N$
        $Y(i) = Y(i) + A(i,j) * X(j)$

For simplicity, assume that the vector Y is dense. For efficiency, we should perform an iteration (i, j) only if $A(i,j) \neq 0$ and $X(j) \neq 0$. This set of iterations can be described using the following system of inequalities:

$$\begin{cases} 1 \leq i \leq N & 1 \leq j \leq N \\ A(i,j) \neq 0 & X(j) \neq 0 \end{cases} \tag{1}$$

Notice that (i) the last two constraints are not linear, and (ii) the set is not necessarily convex. Therefore, methods based on polyhedral algebra, such as Fourier-Motzkin elimination [1], cannot be used to enumerate the points in this set. However, we can look at A as a database relation with tuples $\langle i, j, a \rangle$ of row and column indices and values. Similarly, X can be viewed a database relation of tuples $\langle j, x \rangle$. Finally, the (dense) iteration space, defined by the conjunction of the first two constraints in System 1, can be viewed as a relation R_d of $\langle i, j \rangle$ tuples. The (sparse) iteration set can be viewed as the result of the following relational query expression [14], where equijoins are performed on the common attributes:

$$\sigma_{a \neq 0 \wedge x \neq 0}(R_d(i,j) \bowtie A(i,j,a) \bowtie X(j,x)) \tag{2}$$

In this expression, σ is a *selection* operator which selects those tuples that satisfy the predicate in its subscript [14].

To evaluate this query, we first need to determine whether to do the join on i or the join on j first. In terms of loop transformations, this corresponds to determining whether the i or j loops should be outermost in the final code. For efficiency, it is important to exploit structure in the storage formats of A and X. If A is stored in Compressed Row Storage format (CRS), then the join on i should be done first so that the resulting code enumerates the rows of A in the outer loop, while the inner loop enumerates the elements within a row together with vector X (Figure 2). On the other hand, if the matrix is stored using Compressed Column Storage(CCS), then the join on j should be done first (Figure 3). Given the join order, we must determine how the joins for each variable are to be performed. There are a number of ways to perform these joins, and we will discuss them with reference to Figure 2. If elements in rows of A and in the vector X are sorted, we can use "2-finger" *merge-join* [14]; alternatively, if the vector X is scattered into a hash table, we can use *hash join* [14]. In the context of sparse matrices, the hash table is usually just a dense vector called an *accumulator* [11,7]. The cost of scattering can be amortized by lifting the scatter operation out of the loop nest; this is legal since X is invariant in the loop nest. A third possibility is *nested loop* join in which nested loops are used to enumerate elements in a row of A and in X [11].

$$
\begin{array}{l}
\text{DO } i = 1, n \\
\quad \text{DO } \langle v_A, v_x, j\rangle \in A(i,*) \bowtie X \\
\quad\quad Y(i) = Y(i) + v_A * v_x
\end{array}
$$

Fig. 2. MVM for CRS format

$$
\begin{array}{l}
\text{DO } \langle j, v_x\rangle \in A \bowtie X \\
\quad \text{DO } \langle i, v_A\rangle \in A(*,j) \\
\quad\quad Y(i) = Y(i) + v_A * v_x
\end{array}
$$

Fig. 3. MVM for CCS format

In summary, there are four problems that we must address. The first problem is to describe the structure of storage formats to the compiler. We show how to do this in Section 2. The second problem is to formulate relational queries (Section 3.1), and discover joins. In our example, this step was easy because all array subscripts are loop variables. When array subscripts are general affine functions of loop variables, discovering joins requires computing the echelon form of certain matrices (Section 3.2). The third problem is to determine the most efficient join order, exploiting structure wherever possible (Section 3.3). The final problem is to select the implementations of each join (Section 3.4). To demonstrate that these techniques are practical, we present experimental results in Section 4.

Our approach has the following advantages:

- Most of the compilation approach is independent of the details of sparse storage formats. The compiler needs to know *which* access methods are available and their properties, but not *how* they are implemented.

- The access method abstraction is general enough to be able to describe a variety of data structures to the compiler, yet it is specific enough to enable some important optimizations.
- By considering different implementation strategies for the joins, we are able to explore a wider spectrum of time/space tradeoffs than is possible with existing techniques.

2 Describing data structures to the compiler

Since our compiler does not have a fixed set of formats "hard-wired" into it, it is necessary to present an abstraction of storage formats to the compiler for use in query optimization and code generation. We require the user to specify (i) the hierarchical structure of indices, and (ii) the methods for searching and enumerating these indices. To enable the compiler to choose between alternative code strategies, the cost of these searches and enumerations must also be specified. We restrict attention to two-dimensional matrices for simplicity.

2.1 Hierarchical Structure of Indices

Assume that the dense matrix is a relation with three fields named I, J and V where the I field corresponds to rows, the J field corresponds to columns and the V field is the value. We start by specifying the index structure of CRS in which the outer index corresponds to rows, and the inner index corresponds to columns.

$$T_{\mathrm{CRS}} = I \succ J \succ V \tag{3}$$

In this definition, the $\succ$ operator is used to indicate the nesting of the fields within the structure. In contrast, the index structure of CCS storage can be specified as follows.

$$T_{\mathrm{CRS}} = J \succ I \succ V \tag{4}$$

Dense matrices, by contrast, do not have hierarchical indexing structure. We can specify this by using $\times$ to indicate that there is no nesting of structure between I and J.

$$T_{\mathrm{DENSE}} = I \times J \succ V \tag{5}$$

What is the structure of the simplified BlockSolve storage (BSS) format (Fig. 1)? The problem here is that a new *INODE* field is introduced in addition to the row and column fields. Fields like inode number which are not present in the dense array are called *external* fields. An important property that we need to convey is that inodes partition the matrix into disjoint pieces. We denote it by

$$T_{\mathrm{BSS}} = INODE \succ_{\not\cap} (I \times J) \succ V \tag{6}$$

The $\not\cap$ symbol subscript indicates that the sets of indices stored in the sub-arrays are disjoint. This will differentiate the BSS format from the format used often in Finite Element analysis [13]. In this format the matrix is represented as a sum of element matrices. The elements matrices are stored just like the inodes, and the overall type for this format is

$$T_{\mathrm{FE}} = E \succ_{+} (I \times J) \succ V \tag{7}$$

where E is the field of element numbers. Our compiler is able to recognize the cases when the matrix is used in additive fashion and does not have to be explicitly constructed.

We need another building block to represent "flat" collections of tuples such as coordinate storage format, which simply has three arrays for row, column, and value. This format is denoted as follows:

$$T_{\mathrm{coord}} = (I, J) \succ V \tag{8}$$

The (I, J) notation means that we store these fields as a set of $\langle i, j \rangle$ tuples and that there are no fields available for indexing these tuples.

A grammar for building these specifications is given below.

$$\begin{aligned} T := {} & V \\ \mid {} & F \succ_{\mathrm{op}} T \\ \mid {} & F \times F \times F \times \ldots \succ T \\ \mid {} & (F, F, F, \ldots) \succ T \end{aligned}$$

V is the value field, and F denotes the set of index fields.

2.2 Access Methods

For each level of the index hierarchy, *access methods* for searching and enumerating the index must be provided to the compiler.

For the index structure $I_1 \times \ldots \times I_n$ (if n is 1, this is just the base case of a single field), the following methods should be provided to the compiler for each I_k:

$$\begin{aligned} \langle b, h_k \rangle &= \mathtt{Search}(x_k) \\ \{\langle x_k, h_k \rangle\} &= \mathtt{Enum}() \\ t &= \mathtt{Deref}(\langle h_1, \ldots, h_n \rangle) \end{aligned}$$

Here, x_k is the value of the index of type I_k, b is a boolean flag indicating whether the value was found or not and h_k is the *handle* used to dereference the values t of the result type T. For example, a handle can be an integer offset into an underlying array or a pointer.

Notice that we can separately search and enumerate the components of a product type. For example, an implementation of the cartesian storage would provide a way to enumerate all column indices and all row indices. Also notice that the `Enum`() method returns a set of tuples and it is actually represented as several methods, that implement a *stream* interface: `Open`(), `Valid`(), `Advance`(), `Close`().

For the index structure $(I_1, \ldots, I_n)$, only one set of functions need be provided:

$$\begin{aligned} \langle b, h \rangle &= \texttt{Search}(x_1, \ldots, x_n) \\ \{\langle x_1, \ldots, x_n, h \rangle\} &= \texttt{Enum}() \\ t &= \texttt{Deref}(h) \end{aligned}$$

As we describe in Section 3, our compiler generates *plans* which are programs expressed in terms of access methods. Below is a plan for sparse matrix-vector product (the matrix stored in CRS). To generate efficient code from such a plan,

DO $\langle i, h_i \rangle \in \texttt{Enum}_i(A)$
    Row $= \texttt{Deref}_i(A, h_i)$
    DO $\langle j, h_j \rangle \in \texttt{Enum}_j(\text{Row})$
        $v_A = \texttt{Deref}_j(\text{Row}, h_j)$
        $v_x = \texttt{Deref}(X, \texttt{Search}(X, j))$
        $v_y = \texttt{Deref}(Y, \texttt{Search}(Y, i))$
        $v_y = v_y + v_A * v_x$

Fig. 4. Plan for Sparse MVM

it is desirable to have additional information about indices and access methods.

- *Cost of searching:* We differentiate between $O(1)$ lookups, $O(\log n)$ binary searches and $O(n)$ linear searches.
- *Ordering of indices:* Ordered or Unordered when enumerated
- *Range of the indices:* Dense or Sparse. Notice that $O(1)$ lookup does not necessarily imply that an index is dense: a sparse array can be stored using a hash table.

2.3 Discussion

The set of access methods described above does not specify how sparse matrices are created or how non-zero elements (fill) are inserted. It is relatively easy to come up with insertion schemes for simple formats like CRS and CCS which insert entries at a very fine level – for example, for inserting into a row or column as it is being enumerated. More complicated formats, like BlockSolve, are more difficult to handle: the BlockSolve library [9] analyzes and reorders the whole matrix in order to discover inodes.

At this point, we have taken the following position: each data structure should provide a method to *pack* it from a hash table. This is enough for DO-ANY loops, since we can insert elements into the hash table as they are generated, and pack them later into the sparse data structure.

3 Organization of the Compiler

3.1 Obtaining relational queries

Suppose we have a perfectly nested loop with a single statement:

DO $\mathbf{i} \in \mathcal{B}$
    $S : A_0(\mathbf{F}_0\mathbf{i} + \mathbf{f}_0) = \ldots A_k(\mathbf{F}_k\mathbf{i} + \mathbf{f}_k)$

where $\mathbf{i}$ is the vector of loop indices and $\mathcal{B}$ are the loop bounds. We make the usual assumption that the loop bounds are polyhedral, and that the arrays A_k, $k = 0 \ldots N$, are addressed using affine access functions. A_0 is the array being written into. Since we deal only with DO-ALL loops in this paper, we assume that the iterations of the loop nest can be arbitrarily reordered.

If some of the arrays are sparse, then some of the iterations of the loop nest will execute "simpler" versions of the original statement S. In most cases, the simpler version is just a NOP. Bik and Wijshoff [2,4] describe an attribute grammar for computing guards, called *sparsity predicates*, that determine when non-trivial computations must be performed in the loop body. If $\mathcal{P}$ is the sparsity predicate, the resulting program is the following.

DO $\mathbf{i} \in \mathcal{B}$
    IF $\mathcal{P}$ THEN
        $S' : A_0(\mathbf{F}_0\mathbf{i} + \mathbf{f}_0) = \ldots A_k(\mathbf{F}_k\mathbf{i} + \mathbf{f}_k)$

The predicate $\mathcal{P}$ is a boolean expression in terms of individual $NZ(A_k(\mathbf{F}_k\mathbf{i} + \mathbf{f}_k))$ predicates, where the predicate $NZ(A_k(\mathbf{F}_k\mathbf{i} + \mathbf{f}_k))$ evaluates to true if and only if the array element in question is explicitly stored.

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

$$\mathbf{H} = \begin{pmatrix} \mathbf{I} \\ \mathbf{F}_0 \\ \vdots \\ \mathbf{F}_n \end{pmatrix} \quad \mathbf{a} = \begin{pmatrix} \mathbf{i} \\ \mathbf{a}_0 \\ \vdots \\ \mathbf{a}_n \end{pmatrix} \quad \mathbf{f} = \begin{pmatrix} \mathbf{0} \\ \mathbf{f}_0 \\ \vdots \\ \mathbf{f}_n \end{pmatrix} \tag{9}$$

Following [10], the matrix $\mathbf{H}$ is called a *data access matrix*. Notice that the following *data access equation* holds:

$$\mathbf{a} = \mathbf{f} + \mathbf{H}\mathbf{i} \tag{10}$$

Furthermore, we view the arrays A_k as relations with the following attributes:

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

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

$$\sigma_{\mathcal{P}} \sigma_{(\mathbf{a}=\mathbf{f}+\mathbf{H}\mathbf{i})} (R_I \times \ldots \times A_k(\mathbf{a}_k, v_k) \times \ldots) \tag{11}$$

3.2 Discovering joins

The key to efficient evaluation of relational queries like (11) is to perform equi-joins rather than cross products followed by selections. Intuitively, this involves "pushing" the selections $\sigma_{(\mathbf{a}=\mathbf{f}+\mathbf{H}\mathbf{i})}$ through the cross-products to expose joins. In the matrix-vector product example discussed in Section 1, the joins were simple equijoins of the form $a = b$. More generally, array subscripts are affine functions of loop variables, and we should look for affine joins of the form $a = \alpha b + \beta$ for some constants α and β.

It is useful to look at this in terms of the data access equation. Let a_j, f_j and $\mathbf{h}_j^T$ be the j-th element of $\mathbf{a}$, the element of $\mathbf{f}$ and the row of $\mathbf{H}$, respectively. The following result tells us when array dimensions a_j and a_k are related by an affine equality:

$$\left(\forall \mathbf{i} : a_j = \alpha a_k + \beta\right) \Leftrightarrow \left(\left(f_j = \alpha f_k + \beta\right) \wedge \left(\mathbf{h}_j = \alpha \mathbf{h}_k\right)\right) \tag{12}$$

This suggests that we look for rows of $\mathbf{H}$ which are multiples of each other. Consider the following variation on matrix-vector product, where X and A are sparse, and Y is dense.

DO $i = 1, n$
  DO $j = 1, n$
    $Y(i) = Y(i) + A(i - j, j) * X(j)$

The data access equation for this loop is the following:

$$\begin{pmatrix} i \\ j \\ s \\ t \\ i_y \\ j_x \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & -1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{13}$$

In this equation, s and t are the row and column indices for accessing A, while i_y and j_x are the indices for accessing X. One equijoin is clear from (13): $i = i_y$. It seems that we are left with two more joins: s (a trivial join of one variable) and $t = j_x = j$. However, for any fixed value of $i = i_0$, we get

$$s = i_0 - j \tag{14}$$

which is an affine join on s and j! In other words, we can exploit the order in which variables are bound by joins to expose more joins than are evident in the data access matrix.

To do this systematically, suppose that the data access matrix is in the block form shown in Figure 5, where all entries in the column vectors c_1, c_2 etc are non-zero. It is trivial to read off affine joins: there is an affine join corresponding to each column $\mathbf{c}_i$ of this matrix. The entries in $\mathbf{L}'_i$ are the coefficients in the affine joins of variables bound by previous joins.

$$\begin{pmatrix} c_1 & 0 & 0 & & 0 \\ L'_2 & c_2 & 0 & \dots & 0 \\ \multicolumn{2}{c}{L'_3} & c_3 & & 0 \\ & \vdots & & \ddots & \\ \multicolumn{4}{c}{L'_r} & c_r \end{pmatrix}$$

Fig. 5. Permuted Column Echelon Form of Data Access Matrix

We can get our data access matrix into this form by permuting equations, as shown below. Affine joins can be read off trivially from this form.

$$\begin{pmatrix} i \\ i_y \\ j \\ s \\ t \\ j_x \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 1 & -1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{15}$$

It is easy to show that we can get a general data access equation into this form in two steps.

1. Apply column operations to reduce the data access matrix to column echelon form. This is equivalent to multiplying the matrix $\mathbf{H}$ on the right by a unimodular matrix $\mathbf{U}$, which can be found using standard algorithms [6].
2. Apply row permutations as needed. This is equivalent to multiplying the matrix produced in the previous step by a permutation matrix $\mathbf{P}$.

We did not need the first step in our example since the matrix in Equation 13 is already in column echelon form.

More formally, we have

$$\mathbf{H}' = \mathbf{PHU} \tag{16}$$

If $\mathbf{H}$ has rank r, then $\mathbf{H}'$ can be partitioned into blocks $\mathbf{L}_m$ for $m = 1, \ldots, r$, such that in each block $\mathbf{L}_m$ the columns after m are all zero and the m-th column $(\mathbf{c}_m)$ is *all* non-zero:

$$\mathbf{H}' = \begin{pmatrix} \mathbf{L}_1 \\ \vdots \\ \mathbf{L}_r \end{pmatrix} \qquad\qquad \mathbf{L}_m = \begin{pmatrix} \mathbf{L}'_m & \mathbf{c}_m & \mathbf{0} \end{pmatrix} \tag{17}$$

Define $\mathbf{j} = \mathbf{U}^{-1}\mathbf{i}$ and $\mathbf{b} = \mathbf{P}(\mathbf{a} - \mathbf{f})$. Then the data access equation (10) is transformed into:

$$\mathbf{b} = \mathbf{H}'\mathbf{j} \tag{18}$$

Now if we partition $\mathbf{b}$ according to the partition (17) of $\mathbf{H}'$, then for each $m = 1, \ldots, r$ we get:

$$\mathbf{b}_m = \mathbf{L}_m\mathbf{j} = \mathbf{L'}_m\mathbf{j}(1 : m-1) + \mathbf{c}_m * \mathbf{j}(m) \tag{19}$$

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

$$\mathbf{b}_m = \text{invariant} + \mathbf{c}_m * \mathbf{j}(m) \tag{20}$$

3.3 Ordering joins

In Section 3.2, we permuted rows of the data access matrix *after* finding its column echelon form. However, there is nothing sacrosanct about the order in which equations appear in the data access matrix. In fact, good code generation requires that the permutation be interleaved with the computation of the echelon form of the data access matrix. For example, in the sparse MVM from Section 3.2, the order of variables in (15) is not suitable if the matrix is in CRS form since the access to matrix A in the inner loop is along diagonals. A better strategy is to generate code which enumerates the rows s, and then the columns t. This corresponds to the following permuted echelon form:

$$\begin{pmatrix} s \\ i \\ j \\ i_y \\ j_x \\ t \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 1 \\ 0 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \qquad\qquad \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} \tag{21}$$

To get this echelon form, we permute the rows so that s appears first, and we add the first column of $\mathbf{H}$ in (13) to the second. The new loop variable u runs over the first join, which is just the enumeration of $s \in A$. The second variable v joins the rest of the variables for a fixed $u = u_0$:

$$v = i - u_0 = j = i_y - u_0 = j_x = t \tag{22}$$

Our compiler obtains this join order as follows. Since A is in CRS format, we conclude from the specification of its index structure given in Section 2 that it is desirable to have the following join orders

$$s \succ t \succ a \tag{23}$$

$$j_x \succ x \tag{24}$$

That is, it is desirable to bind s, the row index, in an outer loop before binding t, the column index etc. One approach is to search exhaustively for a permutation $\mathbf{P}$ for which the resulting join order violates the fewest $\succ$ constraints. Our compiler uses a heuristic to avoid the search. In this example, this heuristic produces the "right" join order shown above. For lack of space, we omit details.

Figures 2 and 3 are examples of code produced by this step.

3.4 Implementing joins

The example in the previous section has two joins. One join is a degenerate one variable join on s which simply enumerates over the rows of A. The second join is more interesting: for each $s = s_0$ it joins R_I (the relation determining the dense iteration space), X, Y and the row $A(s_0, *)$. How is it implemented?

There are three basic algorithms for implementing joins between two relations R and S on a common attribute a.

- Enumerate $a_1 \in S$, $a_2 \in R$ and test whether $a_1 = a_2$. This is called *nested loop* join. A variation is to search R instead of enumerating it. We call this *enumerate-search* join.
- If the tuples in R and S can be enumerated in a common sorted order of a, then the join can be computed using a well known "two-finger" *merge-join* algorithm.
- We can insert the tuples of R into a hash table, enumerate S and search into the hash table. This is called *hash join*.

In our example, we can search cheaply into R_I and Y, and we need an efficient way of enumerating X (which we have assumed to be sparse) and the row $A(s_0, *)$. If X and the row can be enumerated in increasing order of $j_x = t$, then we can apply merge join. Whether such an enumeration is possible can be determined from the information in Section 2.2. The search into R_I, which is nothing more that a test against loop bounds, can in most cases be eliminated by projecting the loop bounds onto the surrounding loop variables (s and $j_x = t$ in this example).

Figure 4 is an example of the code generated by this step. This code is "instantiated" using the methods provided by the user, to generate the final output code.

The algorithms for performing two-relation joins can be easily generalized to many-relation joins, and to affine joins. Generalizations to the case when the sparsity predicate requires enumeration of zeroes as well as non-zeroes (as is required for the addition of sparse vectors) are also straight-forward. For lack of space, we omit the details.

4 Experiments

4.1 Different join implementations

We have claimed in the introduction that different implementations of joins have different time/space tradeoffs. Figure 6 plots the times for merge join and hash join implementations of a dot product of two sparse vectors with 50 non-zeroes each. The number of common indices varies on the X-axis. Y-axis is the time to perform the dot product on a single node of an IBM SP-2. The "hash-join (amortized)" graph shows the times for hash-joins with hashing done once and its cost amortized over 5 iterations.

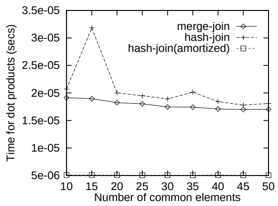

Fig. 6. Merge join vs. hash join

These results demonstrate that using merge join is advantageous when memory is limited. Unlike Bik and Wijshoff, we are able to explore this alternative to hash join in our compiler.

4.2 Join ordering

It is also interesting to compare the performance of the code generated by our compiler with the code produced by the Sparse BLAS generator [5] for the sparse matrix-matrix product $C = A \cdot B^T$, when all matrices are stored in CCS.

DO $i = 1, N$
    DO $j = 1, N$
        DO $k = 1, N$
            $C(i, j) = C(i, j) + A(i, k) * B(j, k)$

The relational query for the sparse iteration set is (assuming C is preallocated):

$$\sigma_{NZ(A(\ldots)) \wedge NZ(B(\ldots)) \wedge NZ(C(\ldots))}(R_I(i, j, k) \bowtie A(i, k, a) \bowtie B(j, k, b) \bowtie C(i, j, c)) \tag{25}$$

The joins are on the common attributes i, j and k. We need to determine the permutation $\mathbf{P}$, exploiting the hierarchical storage formats for the relations. Since the matrices are in CCS format, the specification in Section 2 suggests that the following constraints on join order should be respected:

$$j \succ i,\ k \succ i,\ k \succ j \tag{26}$$

Without doing any search, our compiler generates the join order (k, j, i), which satisfies all three constraints. The Sparse BLAS generator produces a (j, k, i) loop order and thus only removes searches from the innermost loop and leaves a search into B in the second loop.

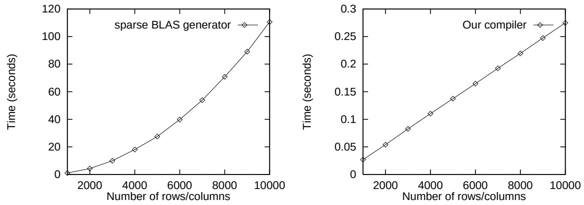

Fig. 7. MMM performance

Figure 7 plots the performance of the two implementations (notice the difference in time scales). We used the same sparse data structures – an implementation of Compressed Column Storage in Bik's sparse run-time library – and we preallocated $\mathbf{C}$ to exclude the overhead of insertions. To be able to predict and preallocate the storage for the product we have used banded matrices (stored in CCS, of course) for our experiments. The A and B matrices have the half-bandwidth of 3 (i.e. most rows have 7 non-zeroes). The half-bandwidth for C is 6 – it has 13 non-zeroes in most rows. As is expected the time for the (j, k, i) ordering is quadratic in N, because the k loop has to run from 1 to N. The (k, j, i) loop is linear, because both j and i loops enumerate over sparse data structures.

4.3 BlockSolve

Table 2 shows the performance of the MVM code from the BlockSolve library and code generated by our compiler, for 12 matrices. Each matrix was stored in the *clique/inode* storage format used by the BlockSolve library and was formed from a 3d grid with a 27 point stencil with a varying number of unknowns, or components, associated with each grid point. The grid sizes are given along the left-hand side of the table; the number of components is given across the top. The left number of each pair is the performance of the BlockSolve library; the right is the performance of the compiler generated code. The computations were performed on a thin-node of an SP-2. These results indicate that the performance of the compiler-generated code is comparable with the hand-written code, even for as complex a data structure as BlockSolve storage format.

	1	3	5	7
$10 \times 10 \times 10$	4.16/4.65	16.43/17.55	23.03/24.23	28.04/28.89
$17 \times 17 \times 17$	4.15/4.40	16.24/17.32	23.52/24.26	26.21/27.00
$25 \times 25 \times 25$	4.22/4.40	16.19/17.14	22.85/23.05	—/—

Table 2. Hand-written/Compiler-generated (Mflops)

5 Conclusions and future work

We have presented a novel approach to compiling sparse codes: we view sparse data structures as database relations and the execution of sparse DO-ANY loops as relational query evaluation. By abstracting the details of sparse formats as access methods, we are able to generate efficient sparse code for a variety of data structures.

For lack of space, there are some details we have not addressed in the paper.

- Sometimes we might want to store a matrix A in indices other than row i and column j. For example, a matrix might be stored by diagonal. We can view this as a linear transformation from the index space of $\langle i, j\rangle$ into the index space $\langle s, t\rangle$ with $s = i - j$ and $t = j$. It is quite straight-forward to augment the data access equation (10) of Sec. 3.1 with this kind of information. This issue of handling different data structure *orientation* has also been previously addressed by Bik and Wijshoff.
- Some storage formats permute the rows or columns (or both) of a matrix. A good example is the *jagged diagonal* format [12], which sorts the rows by the number of non-zeroes. The rows of the matrix are numbered now using a new index i' and the format supplies a permutation that relates the new index to the old row index i. If we view the permutation as a relation with attributes $P(i, i')$, then the old matrix can be expressed a join and a projection

$$A = \pi_{i,j,v}(A_{\text{jagged}}(i', j, v) \bowtie P(i, i')) \tag{27}$$

  Our compiler can handle queries of this form, although we have not explored the necessary language required to specify this.

We are currently working on extending these techniques to loops with dependences.

References

1. C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Principle and Practice of Parallel Programming*, pages 39–50, April 1991.
2. Aart Bik. *Compiler Support for Sparse Matrix Computations.* PhD thesis, Leiden University, the Netherlands, May 1996.
3. Aart J.C. Bik, Peter M.W. Knijnenburg, and Harry A.G. Wijshoff. Reshaping access patterns for generating sparse codes. In *Seventh Annual workshop on Languages and Compilers for Parallel Computing*, August 1994.

4. Aart J.C. Bik and Harry A.G. Wisjhoff. 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. Peter Brinkhaus, Aart Bik, and Harry A.G. Wijshoff. Subroutine on demand-service. sparse blas 2 & 3. http://hp137a.wi.leidenuniv.nl:8080/blas-service/blas.html.
6. Henri Cohen. *A Course in Computational Algebraic Number Theory*. Graduate Texts in Mathematics. Springer-Verlag, 1995.
7. John R. Gilbert and Cleve Moler adn Robert Schreiber. Sparse matrices in matlab: Design and implementation. Technical Report CSL-91-4, Xerox PARC, June 1991.
8. Mark T. Jones and Paul E. Plassmann. The efficient parallel iterative solution of large sparse linear systems. In A. George, J. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *IMA Volumes in Mathematics and Its Applications*, pages 229 – 245. Springer-Verlag, 1993. Also ANL MSC Preprint P314-0692.
9. Mark T. Jones and Paul E. Plassmann. BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems. Technical Report ANL–95/48, Argonne National Laboratory, December 1995.
10. Wei Li and Keshav Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 11(4):353–375, November 1993.
11. Sergio Pissantezky. *Sparse Matrix Technology*. Academic Press, London, 1984.
12. Youcef Saad. Kyrlov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, November 1989.
13. Gilbert Strang. *Introduction to applied mathematics*. Wellesley-Cambridge Press, 1986.
14. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, v. I and II*. Computer Science Press, 1988.