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

Sparse matrix computations are ubiquitous in computational science. However, the development of high-performance software for sparse matrix computations is a tedious and error-prone task, for two reasons. First, there are no standard ways of storing sparse matrices, since a variety of formats are used to avoid storing zeros, and the best choice for the format is dependent on the problem and the architecture. Second, for most algorithms, it takes a lot of code reorganization to produce an efficient sparse program that is tuned to a particular format. We illustrate these points by describing two formats — a classical format called Compressed Column Storage (CCS) [6] and a modern one used in the BlockSolve library [7].

CCS format is illustrated in Fig. 2. The matrix is compressed along the columns and is stored using three arrays: `COLP`, `VALS` and `ROWIND`. The array section $\texttt{VALS}(\texttt{COLP}(j) \ldots (\texttt{COLP}(j+1)-1))$ stores the non-zero values of the j-th column and the array section $\texttt{ROWIND}(\texttt{COLP}(j) \ldots (\texttt{COLP}(j+1)-1))$ stores the row indices of the non-zero elements of the j-th column.

This is a very general and simple format. However, it does not exploit any application specific structure in the matrix. The format used in the BlockSolve library exploits structure present in sparse matrices that arise in the solution of PDEs with multiple degrees of freedom. Such matrices often have groups of rows with identical column structure called *i-nodes* ("identical nodes"). Non-zero values for each i-node can be gathered into a dense matrix as shown in Fig. 3. This helps reduce sparse storage overhead and improves performance by making sparse matrix-vector products "rich" in dense matrix-vector products.

* This work was supported by NSF grant CCR-9503199, ONR grant N00014-93-1-0103, and the Cornell Theory Center.

$$\begin{pmatrix} a & 0 & b & 0 \\ c & 0 & d & 0 \\ e & f & 0 & g \\ h & i & 0 & j \end{pmatrix}$$

Fig. 1. A matrix

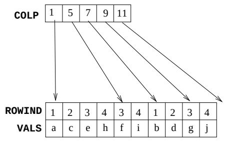

Fig. 2. CCS

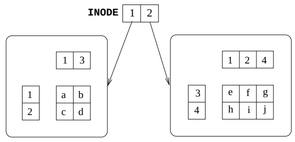

Fig. 3. I-NODE storage

In both cases, computation has to be reorganized in order to exploit the benefits of each format and to avoid inefficient searches. CCS format provides for efficient access to individual columns, but access to rows requires expensive searching. When using BlockSolve format, dense computations are exposed in sparse matrix-vector products $\mathbf{Y} = \mathbf{A} \cdot \mathbf{X}$ by gathering for each i-node the values of $\mathbf{X}$ into a small dense vector and then scattering the result of the dense product into $\mathbf{Y}$.

This demonstrates the difficulty of developing libraries of basic algebraic primitives for sparse matrix computations. Even if we limit ourselves to the formats shown in Tab. 1, we would still have to provide at $6^2 = 36$ versions of sparse matrix-matrix product (assuming that the result is stored in a single format)! The lack of extensibility in such a "sparse BLAS" approach has been addressed by object-oriented solver libraries, like the PETSC library from Argonne [2]. These libraries provide templates for a certain class of solvers (for example, Krylov space iterative solvers) and allow a user to add new formats by providing hooks for the implementations of some algebraic operations (such as matrix-vector product). However, in many cases the implementations of matrix-vector products themselves are quite tedious (as is the case in the BlockSolve library). Also, these libraries are not very useful in developing new algorithms.

One possibility is to give the compiler a *dense* matrix program, declare that some matrices are actually 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 [3, 4], but their approach is limited to simple sparse matrix formats that are not representative of those used in high-performance codes. Intuitively, they trade the ability to handle a variety of formats for the ability to compile arbitrary loop nests.

We have taken a different approach. We focus on the problem of generating efficient sparse *given user-defined storage formats*. In this paper we solve this problem for DOALL loops and loops with reductions. Our approach is based on viewing arrays as *relations*, and the execution of loop nests as evaluation of *relational queries*. Our method of describing storage formats through *access methods* is general enough to specify a variety of formats, yet specific enough to allow important optimizations. Since the class of "DOANY" loops covers not only matrix-vector and matrix-matrix products, but also important kernels within high-performance implementations of direct solvers and incomplete preconditioners, this allows us to address the needs of a number of important applications. One can think of our sparse code generator as providing an exten-

$$
\begin{aligned}
&\text{DO } i = 1, N \\
&\quad \text{DO } j = 1, N \\
&\quad\quad Y(i) = Y(i) + A(i,j) * X(j)
\end{aligned}
$$

Fig. 4. Matrix-vector product

$$
\begin{cases}
1 \le i \le N \wedge 1 \le j \le N \\
A(i,j,a) \wedge X(j,x) \wedge Y(i,y) \\
a \neq 0 \wedge x \neq 0
\end{cases}
$$

Fig. 5. Iteration set constraints

sible set of sparse BLAS codes, which can be used to implement a variety of applications, just like dense BLAS routines!

1.1 Outline of our approach

Consider the loop nest for matrix-vector product $\mathbf{Y} = \mathbf{A} \cdot \mathbf{X}$ shown in Fig. 4. Suppose that the matrix $\mathbf{A}$ and the vector $\mathbf{X}$ are sparse, and that the vector $\mathbf{Y}$ is dense. Moreover, the matrix is stored in the CCS format using the arrays `COLP`, `ROWIND` and `VALS`. To execute this code efficiently, it is necessary to perform only those iterations $\langle i, j \rangle$ for which $A(i,j)$ and $X(j)$ are not zero. This set of iterations can be described by the set of constraints shown in Fig. 5. The first row represents the loop bounds. The constraints in the second row associate values with array indices: for example, the predicate $A(i,j,a)$ constraints a to be the value of $A(i,j)$. Finally, the constraints in the third row specify which iterations update $\mathbf{Y}$ with non-zero values. Our problem is to compute an efficient enumeration of the set of iterations specified by these constraints. For these iterations, we need efficient access to the corresponding entries in the matrices and vectors. Since the constraints are not linear and the sets being computed are not convex, we cannot use methods based on polyhedral algebra, such as Fourier-Motzkin elimination [1], to efficiently enumerate these sets.

Our approach is based on relational algebra, and models A, X and Y as *relations* (tables) that hold tuples of array indices and values. Conceptually, the relation corresponding to a sparse matrix contains *both* zero and non-zero values. We view the iteration space of the loop as a relation I of $\langle i, j \rangle$ tuples. To test if elements of sparse arrays A and X are non-zero, we use predicates $NZ(A(i,j))$ and $NZ(X(j))$. Notice that because Y is dense, $NZ(Y(i))$ evaluates to *true* for all array indices $1 \le i \le N$. If we define the *sparsity predicate* $\mathcal{P} \stackrel{\text{def}}{=} NZ(A(i,j)) \wedge NZ(X(j))$, then the constraints in Fig. 5 can be rewritten as the relational query:

$$
Q_{\text{sparse}} = \sigma_{\mathcal{P}} \Big(I(i,j) \bowtie A(i,j,a) \bowtie X(j,x) \bowtie Y(i,y) \Big) \tag{1}
$$

This query is the formalization of the simple statement: "From all the array indices and values that satisfy the array access functions, select the array values and indices that satisfy the sparsity predicate ."

We have now reduced the problem of efficiently enumerating the iterations that satisfy the system of constraints in Fig. 5 to the problem of efficiently computing a relational query involving selections and joins. This problem in turn is solved by determining an efficient order in which the joins in (1) should be performed and determining how each of the joins should be implemented. These decisions depend on the storage formats used for the sparse arrays.

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 outline this in Section 2 (details are in [8]). 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 3.3). 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. Table 1 illustrates specification of the hierarchy of indices for a variety of formats.

In this notation the $\succ$ operator is used to indicate the nesting of the fields within the structure. For example, $I \succ J \succ V$ in the Compressed Row Storage (CRS) format [10] indicates that we have to access a particular row before we can enumerate the column indices and values; and that within a row, we can search on the column index to find a particular value. The notation (I, J) in the specification of the coordinate storage indicates that the matrix is stored as a "flat" collection of tuples.

The $\times$ operator indicates that the indices can be enumerated independently, as in the dense storage format.

Name	Type
CRS	$T_{\mathrm{CRS}} = I \succ J \succ V$
CCS	$T_{\mathrm{CRC}} = J \succ I \succ V$
COORDINATE	$T_{\mathrm{coord}} = (I, J) \succ V$
DENSE	$T_{\mathrm{dense}} = I \times J \succ V$
INODE	$T_{\mathrm{i-node}} = INODE \succ_{\not\cap} (I \times J) \succ V$
ELEMENT	$T_{\mathrm{FE}} = E \succ_{+} (I \times J) \succ V$

Table 1. Hierarchy of indices for various formats

What is the structure of the i-node storage format (Fig. 3)? 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 the $\not\cap$ symbol subscript in $T_{\mathrm{i-node}}$. This will differentiate the i-node storage format from the format often used in Finite Element analysis [12]. In this format the matrix is represented as a sum of element matrices. The element matrices are stored just like the inodes, and the overall type for this format is T_{FE} in Tab. 1, 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.

Some formats can be seen as providing several alternative index hierarchies. This is denoted by $T \bigcup T$ rule in the grammar for building specifications of index hierarchies:

$$T := V \;\Big|\; F \;\Big|\; F \succ_{\mathrm{op}} T \;\Big|\; F \times F \times F \times \ldots \succ T \;\Big|\; (F, F, F, \ldots) \succ T \;\Big|\; T \bigcup T \tag{2}$$

where the terminal V indicates an array value field, and F indicates an array index field.

2.2 Access Methods

For each level of the index hierarchy (such as I and (J, V) is the case of CRS storage), *access methods* for searching and enumerating the indices must be provided to the compiler, as described in [8].

This set of access methods does not specify 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 (this is the approach taken by Bik and Wijshoff [3, 4]). More complicated formats, like BlockSolve, are more difficult to handle: the BlockSolve library [7] 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 $\mathbf{i} \in \mathcal{B}$
$\quad S : A_0(\mathbf{F}_0\mathbf{i} + \mathbf{f}_0) = \ldots A_k(\mathbf{F}_k\mathbf{i} + \mathbf{f}_k)$

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

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.

$$\mathbf{H} = \begin{pmatrix} \mathbf{I} \\ \mathbf{F}_0 \\ \vdots \\ \mathbf{F}_n \end{pmatrix} \quad \mathbf{a} = \begin{pmatrix} \mathbf{i} \\ \mathbf{a}_0 \\ \vdots \\ \mathbf{a}_n \end{pmatrix} \quad \mathbf{f} = \begin{pmatrix} \mathbf{0} \\ \mathbf{f}_0 \\ \vdots \\ \mathbf{f}_n \end{pmatrix} \tag{3}$$

Following [9], the matrix $\mathbf{H}$ is called a *data access matrix*. Notice that the following *data access equation* holds:

$$\mathbf{a} = \mathbf{f} + \mathbf{H}\mathbf{i} \tag{4}$$

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):

$$\sigma_{\mathcal{P}}\sigma_{(\mathbf{a}=\mathbf{f}+\mathbf{H}\mathbf{i})}\left(R_I \times \ldots \times A_k(\mathbf{a}_k, v_k) \times \ldots\right) \tag{5}$$

Permutations and linear index transformation are easily incorporated in our framework. Linear transformations on array indices can be folded into the data access equation (4). This issue of handling various data structure *orientations* has also been previously addressed by Bik and Wijshoff. Matrices that are permuted by rows and/or columns can be represented by relational queries. We can view a permutation as a relation $P(i, i')$, where i' is the permuted index. Then $P(i, i') \bowtie_i A(i, j, a)$ represents the matrix A permuted by rows. This expression can then be used in the query (5).

3.2 Discovering joins

The key to efficient evaluation of relational queries like (5) is to perform equijoins 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:

$$\left(\forall \mathbf{i} : a_j = \alpha a_k + \beta\right) \Leftrightarrow \left(\left(f_j = \alpha f_k + \beta\right) \wedge \left(\mathbf{h}_j = \alpha \mathbf{h}_k\right)\right) \tag{6}$$

This suggests that we look for rows of $\mathbf{H}$ which are multiples of each other. Consider the variation on matrix-vector product shown in Fig. 6, where X and A are sparse, and Y is dense. The data access equation for this loop is shown in Fig. 7. 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 equi-join is clear from this data access equation: $i = i_y$. 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 $s = i_0 - j$. This is an affine join on s and j! In other words, we can exploit the order in which variables are bound by joins to join more variables than is evident in the data access matrix.

To do this systematically, suppose that the data access matrix is in the block form shown in Fig. 9, 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.

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 [5].

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

Fig. 6. The loop nest

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

Fig. 7. Data access equation

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

Fig. 8. Echelon form for $s \succ t$ hierarchy

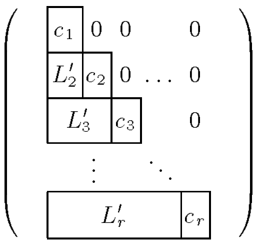

Fig. 9. Block structure of the echelon form

2. Apply row permutations as needed. This is equivalent to multiplying the matrix produced in the previous step by a permutation matrix $\mathbf{P}$.

Formally, we have $\mathbf{H}' = \mathbf{PHU}$. 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:

$$\mathbf{H}' = \begin{pmatrix} \mathbf{L}_1 \\ \vdots \\ \mathbf{L}_r \end{pmatrix} \qquad \mathbf{L}_m = \begin{pmatrix} \mathbf{L}'_m & \mathbf{c}_m & \mathbf{0} \end{pmatrix} \tag{7}$$

Define $\mathbf{j} = \mathbf{U}^{-1}\mathbf{i}$ and $\mathbf{b} = \mathbf{P}(\mathbf{a} - \mathbf{f})$. Then the data access equation (4) is transformed into: $\mathbf{b} = \mathbf{H}'\mathbf{j}$. Now if we partition $\mathbf{b}$ according to the partition (7) of $\mathbf{H}'$, then for each $m = 1, \ldots, r$ we get: $\mathbf{b}_m = \mathbf{L}_m\mathbf{j} = \mathbf{L'}_m\mathbf{j}(1 : m-1) + \mathbf{c}_m * \mathbf{j}(m)$

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

3.3 Ordering and Implementing Joins

The final permutation of the rows of the data access matrix gives us the nesting order of the enumeration of the attributes of the arrays. We would like this order to be consistent with the index hierarchy. Suppose that in our example the matrix is stored using CRS format. Then we would like the enumeration of s to be nested before the enumeration of t. One such ordering and the corresponding echelon form is shown in Fig. 8. In the resulting loop nest the 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 $u = u_0$: $v = i - u_0 = j = i_y - u_0 = j_x = t$.

DO $i = 1, n$
  DO $\langle v_A, v_x, j \rangle \in A(i, *) \bowtie X$
    $Y(i) = Y(i) + v_A * v_x$

Fig. 10. MVM for CRS format

DO $\langle j, v_x \rangle \in A \bowtie X$
  DO $\langle i, v_A \rangle \in A(*, j)$
    $Y(i) = Y(i) + v_A * v_x$

Fig. 11. MVM for CCS 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)

In general, we build a precedence graph for the nesting order of attributes out of the specification of the index structure of the arrays. Our compiler heuristically tries to find a permutation which would satisfy as many constraints as possible. Of course, if the precedence graph is cyclic, then searches are unavoidable. Figures 10 and 11 are examples of join orderings produced by this step.

Once we have found the nesting order of the joins, we have to select an algorithm for performing each of the joins. The basic algorithms for performing joins can be found in database literature [11, 13]. Our compiler selects an appropriate algorithm based on the properties of access methods of the joined relations. It is at this point that the sparsity predicate is "folded" into join implementations in order to produce enumerations over a correct combination of zeros and non-zeros. This way we can treat disjunctive predicates (as in vector addition) as well as conjunctive predicates (as in vector inner products). Also, the basic algorithms for performing two-relation joins can be easily generalized to many-relation joins, and to affine joins. 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. We have compared the performance of hash-join (scatter) and merge-join implemetations of a dot product of two sparse vectors with 50 non-zeros each. We have run our experiments on a single (thin) node of an IBM SP-2. Merge-join has outperformed hash-join by 10-30%. However, if the cost of hashing (scattering) is amortized over many iterations of an enclosing loop, then hash-join outperforms merge-join by an order of magnitude. These results suggest that using merge join is advantageous when memory is limited and when there is no opportunity to hoist hashing outside of an enclosing loop. Unlike Bik and Wijshoff, we are able to explore this alternative to hash join in our compiler.

4.2 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.

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.

References

1. Corinne Ancourt and Franois Irigoin. Scanning polyhedra with do loops. In *Principle and Practice of Parallel Programming*, pages 39–50, April 1991.
2. Argonne National Laboratory. *PETSc, the Portable, Extensible Toolkit for Scientific Computation.* http://www.mcs.anl.gov/petsc/petsc.html.
3. Aart J.C. Bik and Harry A.G. Wijshoff. Advanced compiler optimizations for sparse computations. *Journal of Parallel and Distributed Computing*, 31:14–24, 1995.
4. Aart J.C. Bik and Harry A.G. Wijshoff. Automatic data structure selection and transformation for sparse matrix computations. *IEEE Transactions on Parallel and Distributed Systems*, 7(2):109 – 126, 1996.
5. Henri Cohen. *A Course in Computational Algebraic Number Theory.* Graduate Texts in Mathematics. Springer-Verlag, 1995.
6. Alan George and Joseph W-H Liu. *Computer Solution of Large Sparse Positive Definite Systems.* Prentice Hall, Inc., 1981.
7. 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.
8. Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel sparse code for user-defined data structures. In *Proceedings of Eights SIAM Conference on Parallel Processing for Scientific Computing*, March 1997. Available as Cornell Computer Science Technical Report from http://cs-tr.cs.cornell.edu.
9. Wei Li and Keshav Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 11(4):353–375, November 1993.
10. Sergio Pissantezky. *Sparse Matrix Technology.* Academic Press, London, 1984.
11. Raghu Ramakrishnan. *Database Management Systems.* College Custom Series. McGraw-Hill, Inc, beta edition, 1996.
12. Gilbert Strang. *Introduction to applied mathematics.* Wellesley-Cambridge Press, 1986.
13. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, v. I and II.* Computer Science Press, 1988.