A Generic Programming System for Sparse Matrix Computations

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

Vladimir Kotlyar
IBM T.J. Watson Research Center
30 Saw Mill River Rd.
Hawthorne, NY 10532

Abstract

Sparse matrices are stored in compressed formats in which zeros are not stored explicitly. Writing high-performance sparse matrix libraries is a difficult and tedious job because there are many compressed formats in use and each of them requires specialized code. In this paper, we argue that (i) compressed formats should be viewed as *indexed-sequential access structures* (in the database sense), and (ii) efficient sparse codes exploit such indexing structures wherever possible. This point of view leads naturally to restructuring compiler technology that can be used to synthesize many sparse codes from high-level algorithms and specifications of sparse formats, exploiting indexing structures for efficiency. We show that appropriate abstractions of the indexing structures of commonly used formats can be provided to such a compiler through the type structure of a language like C++. Finally, we describe experimental results obtained from the *Bernoulli Sparse Compiler* which demonstrate that the performance of code generated by this compiler is comparable to the performance of programs in the NIST Sparse BLAS library. One view of this system is that it exploits restructuring compiler technology to perform a novel kind of template instantiation.

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

1 Introduction

Sparse matrix computations are required in many application areas. For example, the finite-element method for solving partial differential equations approximately requires the solution of large linear systems of the form `Ax = b` where `A` is a large sparse matrix. Some web-search engines and data-mining codes compute eigenvectors of large sparse matrices that represent how often certain words occur in documents of interest.

Algorithms for such matrix problems are classified broadly into direct methods and iterative methods [15]. Direct methods such as Cholesky, LU and QR factorizations factorize a matrix into two triangular matrices, and compute the required result from these triangular matrices. Except in special cases, the triangular factors usually have far more non-zeros than the original matrix (a phenomenon called *fill*), so the storage and computational needs of direct methods are usually prohibitive for large sparse matrices [14]. In contrast, iterative methods like Conjugate Gradient and Lanczos methods do not modify the matrix, so they do not suffer from fill. Therefore, iterative methods are increasingly the methods of choice for large sparse matrix problems [32].

In this paper, we will focus on language and systems support for iterative sparse matrix algorithms[1]. The need for this support arises from the fact that sparse matrices are stored in a variety of compressed formats in which zeros are not stored explicitly [28]. The use of compressed formats serves two purposes. First, memory usage becomes more economical; for example, in some fracture mechanics finite-element codes we have developed, matrices have a million rows and columns but have only a few hundred nonzeros in each row. Second, computation time can be reduced because it is not necessary to multiply or add zeros. There are at least forty or fifty formats that are widely used, and it is common to use application-specific formats. Since each format requires its own carefully tuned code, the problem of designing libraries of iterative algorithms which can support all these compressed formats (and which can be easily extended to new formats) is a formidable one.

The approach taken by the numerical analysis community (for example, in the PETSc library from Argonne [5]) is to encapsulate the format-dependent code into a set of *Basic Linear Algebra Subroutines* (BLAS) which are invoked from high-level, format-independent implementations of iterative methods[2]. The high-level iterative codes have to be written just once, but they must be linked with format-specific BLAS. For dense matrices, highly tuned implementations of BLAS are routinely provided by computer vendors [11]. For sparse matrices, the software problem is much more difficult because of the need to support such a large number of formats. Although a number of sparse BLAS libraries have been written [12, 23, 31], they have had limited success because (i) they support only a small number of formats, and (ii) they provide no leverage for people designing new formats.

In this paper, we describe a system that combines *generic programming methodology* [22] with *restructuring compiler technology* [42] to support the development of iterative sparse matrix codes. Generic programming is a methodology for simplifying the development of code libraries in which the same set of algorithms have to be written for different data structures. This methodology requires the design of an API which is supported by all data structure designers, and which is used to write *generic* programs expressing the algorithms of interest in a data-structure-neutral fashion. Linking such a generic program with any data structure implementation that supports the API gives us an implementation of the algorithm for that data structure. Standard compiler optimizations like procedure inlining can be used to make the resulting code efficient. The most well-known example of generic programming is the C++ Standard Template Library (STL) [4] in which algorithms like searching and sorting are implemented for a variety of data structures like arrays, singly-linked lists and doubly-linked lists, using the API of one-dimensional sequences. To apply generic programming ideas to our problem, we note that in our problem domain, the algorithms of inter-

[1]As we argue in the concluding section of this paper, some of our techniques may be applicable to direct methods as well but this remains to be demonstrated.

[2]The BLAS are described in more detail in Section 2; for now, they can be considered to be basic matrix computations which must be coded very differently for different compressed formats.

est are iterative matrix algorithms and the BLAS, while the data structures of interest are the variety of compressed formats. The key problems are to (i) design an API that will be the interface between generic programs and the implementations of compressed formats, and (ii) implement a system that permits the generation of efficient code.

This paper presents such an API and generic programming system. We describe the solution in stages by taking successively more nuanced views of compressed formats. In Section 2, we describe a few important sparse algorithms and compressed formats. We also propose a simple API called the *Strawman API*, and describe a generic programming system designed around it. Intuitively, this API views sparse formats as *random access* data structures. This view is of course inappropriate for sparse formats and therefore leads to very inefficient code, but it does permit us to introduce key ideas simply.

The desire for greater efficiency motivates the *Woodenman API* in Section 3. This API views sparse formats as *sequential access* data structures [38]. We make the case for a generic programming system in which generic algorithm writers code for the Strawman API, but invoke a restructuring compiler that views sparse formats through the Woodenman API and restructures the generic program into efficient code. We give experimental results that show that although this approach improves code efficiency over the use of the Strawman Interface alone, the code produced is still not as efficient as library code for most formats.

In Sections 4 and 5, we present the final API, called the *Ironman API*, that views sparse formats as *indexed-sequential access* data structures [38]. Section 4 describes the indices of interest in compressed formats,

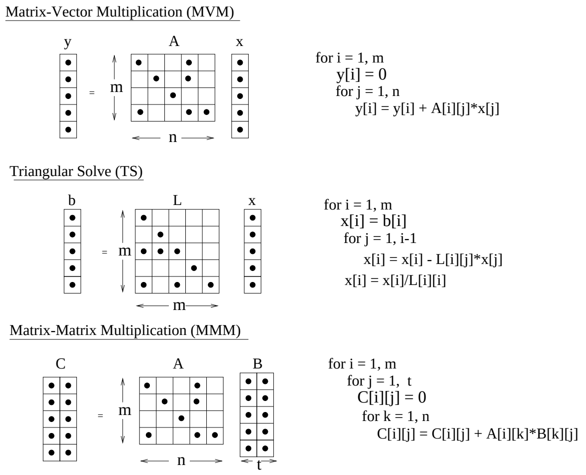


Figure 1: Basic Linear Algebra Subroutines

while Section 5 describes the details of the Ironman API and gives an implementation of a generic programming system that supports this API. We have a prototype of this system implemented, and we are currently reengineering it so that it takes generic programs written in C++ as input. Section 6 gives a brief introduction to the compiler technology used within the system, and Section 7 presents experiments with code generated by our existing system that show that our approach can generate code competitive with handwritten code in the Sparse BLAS library. Section 8 discusses related work, while Section 9 describes ongoing work.

2 A Simple Generic Programming System: the Strawman API

In this section, we introduce some of the algorithms and compressed formats that will be our running examples for the rest of the paper. We also present a simple API called the *Strawman interface* and design a generic programming system for this interface. Although the efficiency of the resulting code is poor, this API lets us introduce the key ideas simply.

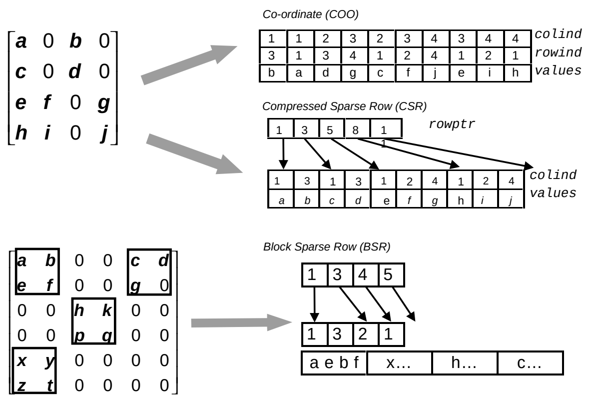


Figure 2: Compressed Formats

2.1 Key algorithms

Although our techniques can be applied to entire iterative sparse matrix algorithms, we will focus on algorithms supported by the NIST Sparse BLAS library, which are the following.

- *Matrix-Vector Multiplication (MVM)*: `y = A*x`: The matrix `A` is sparse, and vectors `y` and `x` are dense.
- *Solution of Triangular Systems (TS)*: `Lx = b`: some problems involve solving multiple systems with the same `L` but different `b`'s.
- *Matrix-Matrix Multiplication (MMM)*: `C = A*B`: `A` is sparse, while `C` and `B` are dense. This is a generalization of matrix-vector product in which a sparse matrix `A` is multiplied by a set of dense vectors represented by the column vectors of matrix `B`.

Figure 1 shows pseudo-code for these algorithms.

2.2 Compressed formats

Figure 2 shows a sparse matrix and three commonly used compressed formats. 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 stored in a particular order such as row-major or column-major order, or they may be ordered arbitrarily. A disadvantage of co-ordinate storage is that it does not permit indexed access to either rows or columns of a matrix. *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. Some of the rows of the matrix may be empty. *Compressed Sparse Column storage* (CSC) is the transpose of CSR in which the non-zeros are stored column-by-column, and it offers indexed access to columns but not rows.

Some sparse matrices have small dense blocks occurring in different positions inside the matrix. It is important to exploit these dense blocks to improve storage and computational efficiency. Figure 2 shows *Block Sparse Row* (BSR) storage which can be viewed as a CSR representation in which the non-zeros are small dense blocks rather than single non-zero elements.

2.3 The Strawman API and generic programming system

As described earlier, the key design decision in a generic programming system is the API that must be supported by the implementations of all data structures. We will use a different class to implement each compressed format, and require such a class to support two methods called `get` and `set`.

- The `get` method takes the row and column co-ordinates as input, and returns the value at that position.

```
template<class ELT>
class StrawmanMatrix {
    int m; //number of rows
    int n; //number of columns
public:
    StrawmanMatrix(int r,int c) {m=r;n=c;}
    int rows() {return m;}
    int columns() {return n;}
    virtual ELT get(int r, int c) = 0;
    virtual void set(int r, int c, ELT v) = 0;
    // Implementation of `A[r][c]' notation.
    class RowRef operator[](int r) { return RowRef(A,r) }
    ....
}
```

Figure 3: The Strawman API: `get/set`

- The `set` method takes a value and row/column co-ordinates as input, and stores the value into that position.

In addition to these methods, there must be methods to return the number of rows and columns in the matrix. Figure 3 shows the Strawman API. Notice that operator-overloading is used to permit programmers to use array syntax rather than invocations of the `get/set` methods.

It is up to the format designer to implement the `get/set` methods as efficiently as possible. For co-ordinate storage, for example, `get` can be implemented by simple linear search, or by binary search if the matrix elements are sorted in lexicographic order. Of course, keeping elements sorted may have its own overheads if many `set` operations into random positions of the matrix are performed, so it is the responsibility of the designer of the compressed format to determine the best strategy. The code in Figure 4 shows one implementation of co-ordinate storage.

To write a generic program in this system, the programmer writes code as though all matrices were dense, but identifies classes that must be used to implement sparse matrices. For example, generic matrix-vector product is coded as shown in Figure 5. To create matrix-vector product for a particular compressed format like co-ordinate storage, the programmer writes template instantiation code, as shown in Figure 5.

```
//co-ordinate storage
template<class ELT>
struct CooStorage {
    vector<int> *rowind;
    vector<int> *colind;
    vector<ELT> *values;
    const int nz;
    CooStorage(vector<int> *_rowind, vector<int> *_colind,
               vector<ELT> *_values)
        : rowind(_rowind), colind(_colind), values(_values),
          nz(rowind->size()) {
    }
};
//Strawman view of storage
template <class ELT>
class CooRandom : public StrawmanMatrix<ELT> {
protected:
    CooStorage<ELT> *A;
public:
    CooRandom(int m, int n, CooStorage<ELT> *A)
        : StrawmanMatrix<ELT>(m,n), A(A) { }
    virtual ELT get(int r, int c) {
        for (int k=0; k < A->nz; k++)
            if((*A->rowind)[k] == r && (*A->colind)[k] == c)
                return (*A->values)[k];
        return 0.0;//zero elements are not stored
    }
    virtual void set(int r, int c, ELT v) {
        for (int k=0; k < A->nz; k++)
            if((*A->rowind)[k] == r && (*A->colind)[k] == c)
                { (*A->values)[k] = v; return; }
        assert(false);//fail if element not allocated
    }
};
```

Figure 4: Co-ordinate Storage: Strawman API

```
template <class T, class ELT>
void mvm(T A, ELT x[], ELT y[])
{
    for (int i=0; i<A.rows(); i++) {
        y[i] = 0;
        for (int j=0; j<A.columns(); j++)
            y[i] += A[i][j] * x[j];
    }
}
//MVM for co-ordinate storage
template void mvm(CooRandom<double> A,
                  double x[], double y[]);
```

Figure 5: Generic Program Instantiation

2.4 Discussion

The Strawman API is very convenient for expressing algorithms in a data-structure-neutral way. Unfortunately, the efficiency of the resulting code is poor. There are two reasons for this.

1. The `get` method is very inefficient because most compressed formats do not support efficient random access.
2. The final code iterates over the bounds of the full matrix and therefore performs computations with both zeros and non-zeros, but the computations with zeros are usually redundant.

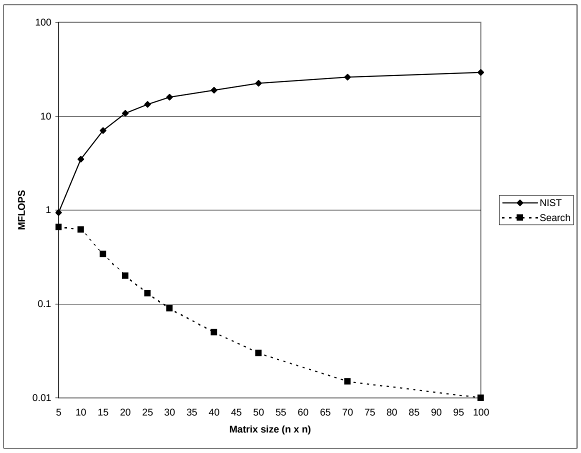


Figure 6: MVM performance for Coordinate Storage: NIST Library vs. Strawman API

In fact, it is easy to see that the coordinate storage MVM code produced by this strategy requires $O(n^2 * NZ)$ time for a $n \times n$ matrix with NZ non-zeros, since $O(n^2)$ floating-point operations are performed and for each operation, a `get` costing $O(NZ)$ time must be executed. The implementation in the NIST Sparse BLAS library takes $O(NZ)$ time which is asymptotically better, as can be seen in Figure 6. These numbers were obtained in the Pentium II platform described in more detail in Section 7.[3]

3 An Enumeration-based API: the Woodenman Interface

One approach to avoiding random accesses and computations with zeros is to recast sparse algorithms in terms of *enumerations* of non-zero elements. Figure 7(a) shows such an enumeration-based algorithm for doing MVM. For each non-zero element `A[r][c]`, we compute the product `A[r][c]*x[c]` and add the result to `y[r]`.

Even though enumeration-based algo-

[3]The Y-axes of the graphs in this paper show program performance expressed in millions of floating point operations per second (MFLOPS).

```
for r = 1, m
  do
    y[r] = 0
  od
for each <r,c,v> in non-zeros(A)
  do
    y[r] = y[r] + v*x[c]
  od
```

(a) MVM

```
for r = 1, m
  do
    x[r] = b[r]
  od
for each <r,c,v> in non-zeros(L)
  do
   if (r == c) then       //diagonal element
      x[r] = x[r]/v;
   else if (r > c) then   //lower triangle
      x[r] = x[r] - v*x[c];
   else ;                 //upper triangle
  od
```

(b) TS

Figure 7: Enumeration-based Pseudocode

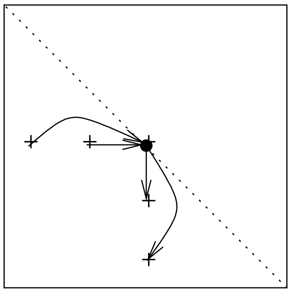


Figure 8: Enumeration Order for TS

rithms look less natural, it might appear that we could use them as a basis for a generic programming system by requiring all matrix classes to support enumeration of non-zeros. Such a class would present a *sequential access* view [38] of a compressed format, rather than a random access view. However, it is easy to see that *enumeration-based algorithms may not be correct if there are dependences between loop iterations*, as there are in triangular solve. Figure 7(b) shows enumeration-based triangular solve. From the dense matrix code in Figure 1, we see that this code is correct only if every diagonal element is enumerated (i) after all the non-zeros within its row and to its left, and (ii) before all the non-zeros within its column and below it. This order is illustrated in Figure 8.

While it is reasonable to require that every sparse format class provide a way of enumerating non-zeros, it is not reasonable to require that these enumerations be in a particular order that is convenient for whatever code is being executed. The challenge therefore is to design a system that permits the writing of generic programs which can work with any compressed format but achieves the efficiency of enumeration-based algorithms whenever possible.

3.1 The Woodenman API

We solve this problem by providing two views of compressed formats—a random access view to the writer of generic programs, and a sequential access view to the compiler. As in Section 2, programs are expressed in a data-structure-neutral fashion by writing them as dense matrix programs. Sparse formats are implemented by classes that provide a way of enumerating the non-zeros of the matrix, in addition to providing `get/set` methods. Our system uses restructuring compiler technology to transform the dense matrix code into enumeration-based code if that is legal; otherwise, it uses the `get/set` methods to generate code as in Section 2.

To enable the compiler to generate efficient code, the sparse format class must specify the following properties of the enumeration to the compiler.

- *Enumeration order*: Intuitively, this is a description of the differences in the row/column co-ordinate values of successive elements in the enumeration. It might be "the entries are visited in $< r, c >$ lexicographical order", or "the entries are visited in an arbitrary order."
- *Enumeration bounds*: This describes the row/column co-ordinate values that can actually occur in the enumeration. For example, some matrices have non-zeros only along their diagonals, while other have non-zeros only in their lower triangular and diagonal parts. Conveying this information to the compiler may enable it to generate better code; for example, in the enumeration-based triangular solve pseudo-code shown above, some of the comparisons of `r` and `c` can be eliminated if the matrix is diagonal or if it does not have non-zeros in its upper triangle.

```
//Matrix abstraction for Woodenman API
template<class I, class E>
class WoodenmanMatrix {
public:
   typedef I iterator_type;
   typedef E value_type;
   virtual I begin() = 0;
   virtual I end() = 0;
};
//Base class for all iterator classes
template<class K, class V>
class WoodenmanIterator {
public:
    typedef K key_type;
    typedef V value_type;
    virtual K operator *() = 0;
    virtual V value() = 0;
    virtual void operator ++(int) = 0;
    ...
};
//Class for unordered iterator
template<class K, class V>
class WoodenmanUnorderedIterator
    : public WoodenmanIterator<K,V>
{ };
//definitions of WoodenmanDecreasingIterator,
//               WoodenmanIncreasingIterator etc.
....
```

Figure 9: Woodenman Interface

Both kinds of information can be expressed as systems of linear inequalities. For example, a matrix that has `m` rows and `n` columns and has no non-zeros in its upper triangle can be described as follows.

$$
\begin{aligned}
1 \leq\ & \mathtt{r} \leq \mathtt{m} \\
1 \leq\ & \mathtt{c} \leq \mathtt{n} \\
& \mathtt{c} \leq \mathtt{r}
\end{aligned}
$$

Since dependence testing and code generation tools for restructuring compilers use polyhedral methods [29], these kinds of polyhedral constraints are easy to integrate into modern compilers.

Figure 9 shows the Woodenman API.

```
template<class ELT> class CooStreamIterator;
// A class for matrices stored in the co-ordinate
// format, in which the entries lie within the lower
// triangle.
#pragma bounds { [i,j] | 0 <= i && i < n-1 \
                          && 0 <= j && j < i-1 }
template<class ELT>
class CooStream
    : public CooRandom<ELT>,
      public virtual WoodenmanMatrix<
          CooStreamIterator<ELT>, ELT >
{
public:
    CooStream(int m, int n, CooStorage<ELT> *A) :
        CooRandom<ELT>(m,n,A) { }
    virtual CooStreamIterator<ELT> begin()
        { return CooStreamIterator<ELT>(A,0); }
    virtual CooStreamIterator<ELT> end()
        { return CooStreamIterator<ELT>(A,A->nz); }
};
template<class ELT>
class CooStreamIterator :
    public WoodenmanUnorderedIterator<pair<int,int>,ELT> {
    friend class CooStream<ELT>;
protected:
    CooStorage<ELT> *A; int jj;
public:
    CooStreamIterator(CooStorage<ELT> *A, int jj)
        : A(A), jj(jj) { }
    virtual void operator ++(int) { jj++; }
    virtual pair<int,int> operator *() {
        return make_pair((*A->rowind)[jj],(*A->colind)[jj]);
    }
    virtual ELT value() { return (*A->values)[jj]; }
};
```

Figure 10: COO: Woodenman API

Enumeration is supported through the use of iterators as in the STL. A class implementing the WoodenmanMatrix interface is like a container class in the STL in the sense that it must implement `begin` and `end` methods that return iterators for enumerating non-zeros. The WoodenmanIterator class is an interface that requires methods for dereferencing the iterator to return the "current" row/column and value, and for advancing the iterator. A method for checking equality of iterators must also be implemented, but we have not shown this for simplicity. Enumeration order and bounds can be incorporated into the program through the use of pragmas, but we have chosen to incorporate order information into the class hierarchy by specifying different classes for enumerations that are unordered/increasing/decreasing etc. The bounds on the stored indices are conveyed to the compiler using a pragma.

Figure 10 shows an implementation of co-ordinate storage for the Woodenman API.

To clarify the meaning of these classes, we show in Figure 11 the code that the sparse compiler might produce if the generic MVM program was instantiated for the CooStream class. After method inlining, this code has the same structure as the code in the NIST library.

```
template <>
void mvm(CooStream<double> &A, double x[], double y[])
{
    for (int i = 0; i < A.rows(); i++)
        y[i] = 0;
    for (CooStreamIterator<double> it = A.begin();
            it != A.end(); it++) {
        int r = (*it).first;
        int c = (*it).second;
        double v = it.value();
        y[r] += v * x[c];
    }
}
```

Figure 11: Compiler-generated code for MVM

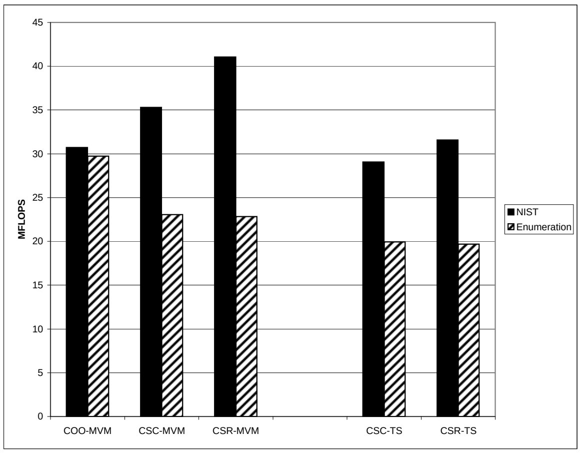


Figure 12: NIST vs. Woodenman API

3.2 Discussion

Figure 12 shows the performance of enumeration-based codes for a number of compressed formats, compared to the performance of handwritten code in the NIST library. For co-ordinate storage, the enumeration-based code is comparable in performance to library code, but for CSR and CSC, the library code is substantially better. To understand this, let us examine the CSR code in more detail. To enumerate the non-zeros of the matrix, the enumeration-based code contains a single loop of the following form.

```
r = 1;
for jj = 1 to NZ do //NZ is the number of non-zeros
    while (jj == rowptr[r+1]) //some rows may be empty
          r++;
    c = colind[jj];
    v = values[jj];
    y[r] = y[r] + v*x[c]
od
```

In contrast, the library code contains a nested loop in which the outer loop enumerates rows and the inner loop enumerates non-zeros within that row. The pseudo-code is shown below.

```
for r = 1 to m do
    for jj = rowptr[r] to rowptr[r+1] - 1 do
      c = colind[jj];
      v = values[jj];
      y[r] = y[r] + v*x[c]
    od
od
```

Although these differences may seem to be minor, there is a fundamental difference in the views of the CSR data structure in these two codes. The Woodenman API views the CSR data structure as a flat, sequential access data structure while the library code exploits the fact that the `rowptr` array permits us to isolate the non-zeros within a row efficiently. In fact, the view taken by the library code is that CSR is an *indexed-sequential access* data structure [38] in which the `rowptr` array is an index (in the database sense) that permits efficient access to the non-zeros within a particular row. This leads naturally to a *nested* view of the data structure. As Figure 12 illustrates, the back-end compiler (`egcs` in this case) performs substantially better on the nested loop code. We experimented with other compilers and found the same result.

A compelling reason for adopting an indexed-sequential access view of compressed formats is that exploiting indices makes a difference in the asymptotic time complexity of the generated code for some problems. Consider the product of two sparse matrices `C = A*B` where `B` is stored in CSR, and `C` is stored in some format

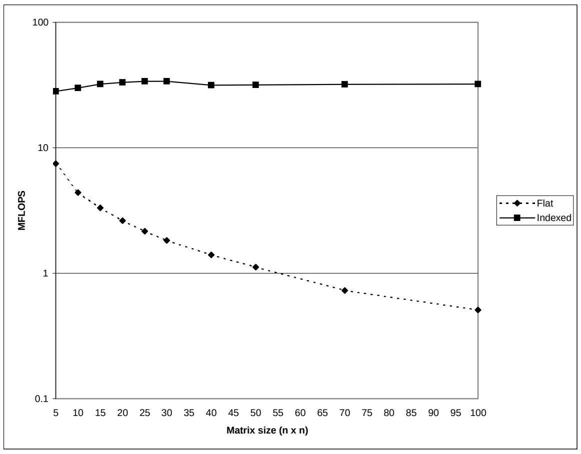


Figure 13: MMM Must Exploit Indices

that permits insertions, such as a hash table. Pseudo-code for this algorithm (assuming `C` is properly initialized) looks like the following.

```
for each <r,c,va> in non-zeros(A) do
  for each <c,c',vb> in non-zeros(B) do
    C[r][c'] = C[r][c'] + va*vb;
  od
od
```

If `B` is treated as a flat, sequential access data structure, the inner loop must scan the entire data structure, so the complexity of the code is $O(NZ(A) * NZ(B))$. If on the other hand, we exploit the index into row `c` of `B`, the complexity of the code is $O(NZ(A) * NZ(B)/n)$ since $NZ(B)/n$ is the average number of non-zeros in a row of `B`. Figure 13 shows an experimental comparison between these two approaches on the Pentium II.

We conclude that viewing compressed formats as sequential access data structures is a partial solution to the problem of compiling efficient code from the generic dense-matrix programs. In the next section, we show that viewing compressed formats as indexed-sequential access data structures permits the compiler to generate more efficient code by exploiting indices.

4 Index Structure of Compressed Formats

Intuitively, an index structure for a compressed format corresponds to a particular *view* of that data structure. The simplest index structure is a hierarchy of indices where each index corresponds to one of the array dimensions. Some formats use indices that are not array dimensions but are obtained by applying a simple function to the array dimensions. One example is a variation of CSR format in which the storage order of rows is a permutation of their order in the actual matrix. The `rowptr` index in this case is a permutation of the row numbers in the actual matrix. Finally, some formats like Jagged Diagonal Storage (JAD) support multiple views.

For the purpose of this paper, we describe these views by using a simple grammar called the *view grammar*. In the next section, we show how this information can be conveyed to the compiler by using an appropriate class structure in which there is one interface class for each production in the grammar.

4.1 Index nesting

In Section 3, we showed that for MMM, it is beneficial to exploit the nested structure of CSR to access elements in a given row of the matrix `B`. If a matrix is considered to be a collection of tuples of the form $< r, c, v >$ where r and c are the row and column co-ordinates and v is the value, then the nested structure of a compressed format can be described by specifying the order in which the fields of these tuples should be accessed. For example, CSR can be specified as follows.

$$CSR : r \rightarrow c \rightarrow v$$

This indicates that the non-zeros within a row of the matrix can be accessed efficiently by using the row co-ordinate as an index into the data structure containing the non-zeros. A similar expression can be written for CSC storage as well. These expressions can obviously be generalized to arrays of arbitrary dimensions, and they can be described formally by the following grammar. In this grammar, *index* may be one of the dimensions of the array, and v denotes array element values.

$$\begin{array}{rcl} E & : & index \rightarrow E \\ & | & v \end{array}$$

In general, an index at a given level may involve multiple array dimensions. One example is provided by co-ordinate storage since neither the row nor the column co-ordinate provides access to a substructure of the compressed format. At the other extreme, both row and column co-ordinates of a dense matrix provide access to substructures. We incorporate these structures into the grammar by enriching what *index* can be.

$$\begin{array}{rcl} index & : & attribute \\ & | & < attribute, \ldots, attribute > \\ & | & < attribute \times \ldots \times attribute > \end{array}$$

For now, attributes may be considered to be array dimensions. The first rule models the case when a single array dimension is used to index a substructure. The second rule models formats like co-ordinate storage for which multiple array dimensions are required to provide access to a substructure. The third rule models formats like dense matrices in which each of a number of array dimensions provides independent access to substructures.

Several sparse matrix formats and their views are given below.

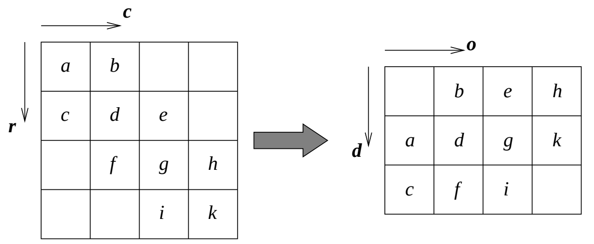


Figure 14: Diagonal Storage

$$\begin{array}{ll} \text{Co-ordinate:} & < r, c > \rightarrow v \\ \text{CSR:} & r \rightarrow c \rightarrow v \\ \text{CSC:} & c \rightarrow r \rightarrow v \\ \text{Dense:} & < r \times c > \rightarrow v \end{array}$$

4.2 Maps

The preceding discussion of sparse matrix views assumed that only array dimensions can be indices. However, this is often not the case.

- *Rotation:* In the diagonal storage format found in the Sparse BLAS, matrix elements are grouped and accessed by diagonals, as shown in Figure 14. In this case, the attributes that are indexed are, d, the diagonal number, and o, the offset within the diagonal. Using these attribute names, the view of the matrix can be expressed as $d \rightarrow o \rightarrow v$, where the matrix dimensions, r and c, can be computed from d and o by,

  $$r = d + o \quad c = o$$

- *Blocking:* The Block Sparse Row (BSR) format is similar to the CSR format except that instead of each matrix element being a scalar, it is a small dense matrix, or block. Each block is accessed by a set of block indices (b_r, b_c), and the scalar elements within each block are accessed by a the offset indices (l_r, l_c). The view of BSR can be expressed in terms of the attribute names, b_r, b_c, l_r and l_c, as,

  $$\text{BSR:} \quad b_r \rightarrow b_c \rightarrow < l_r \times l_c > \rightarrow v$$

  and the block and offset indices are related to r and c by the following, where B is the number of rows and columns in each blocks,

  $$\begin{aligned} r &= b_r * B + l_r \\ c &= b_c * B + l_c \end{aligned}$$

- *Permutation:* Often, sparse matrices are reordered in order to give their nonzeros a particular structure. In these cases, the rows and columns in which the matrix is stored is a permutation of the original row and column indices.

In all of these cases, the view of the storage is most naturally expressed in terms of a different set of indices, and r and c can be easily computed by applying a simple function to the storage indices. This can be expressed by adding a production of the following form to the grammar.

$$E \;\; : \;\; map\{F(in) \mapsto out : E\}$$

For example, the view of the diagonal storage is:

$$map\{d + o \mapsto r, o \mapsto c : d \rightarrow o \rightarrow v\}$$

4.3 Perspective

It may be the case that a compressed format can be viewed in multiple ways. For instance, the Jagged Diagonal format (JAD) found in the Sparse BLAS can be viewed in the following two ways,[4]

$$\begin{aligned} < i, j > &\rightarrow v \\ i \rightarrow j &\rightarrow v \end{aligned}$$

[4]For simplicity, we ignore the permutation that occurs in JAD.

The two views represent the fact that two different sets of methods can be used to access the storage. The first case represents a particularly efficient method for enumerating the the elements of the matrix, which does not provide any ordering guarantees. The second case represents a set of methods that can be used to give random access to the rows, and to enumerate the elements within a row in increasing order by column. The first view is appropriate for MVM, in which a fast enumeration of the whole matrix is desired, and in which no constraints are placed on the order of that enumeration. For TS, this method cannot be used because it violates dependences, so the methods of the second view must be used.

We refer to each of the different views for a single storage format as different "perspectives" on the format, and we represent perspective with our grammar as follows.

$$E \; : \; E \oplus E$$

4.4 Aggregation

Finally, some formats are simply collections of two or more compressed formats. Triangular solve, for instance, might be implemented efficiently if a sparse matrix format provided efficient random access to its diagonal elements, and indexed access to the off-diagonal elements by either rows or columns. This is accomplished sometimes by using different formats to store the different regions of the matrix — the diagonal of the matrix might be stored in a dense vector, and the elements in the lower triangle might be stored in CSR.

In our grammar, we will represent the aggregation of two or more storage formats into a single sparse matrix with the $\cup$ operator.

$$E \; : \; E \cup E$$

4.5 Summary

Below is the complete grammar for expressing views of a sparse matrix format,

$$\begin{array}{rcl} E & : & index \rightarrow E \\ & | & map\{F(in) \mapsto out : E\} \\ & | & E \oplus E \\ & | & E \cup E \\ & | & v \\ \\ index & : & attribute \\ & | & < attribute, \ldots, attribute > \\ & | & < attribute \times \cdots \times attribute > \end{array}$$

5 The Ironman API

As before, we deal with two different API's. The generic programmer views matrices as random access data structures, but the compiler views them through the *Ironman API* as indexed-sequential access data structures whose index structure was described in the previous section. The Ironman API is summarized in Figures 15 and 19.

5.1 Interfaces for Views

Each production in the view grammar given in Section 4 has an associated interface, which we have implemented in C++ as a small number of abstract classes described in Figure 15. The programmer conveys views of a storage format to the sparse compiler by writing a set of classes that inherit from the appropriate interfaces.

The `term_nesting` abstract class denotes an occurrence of the $\rightarrow$ operator within the view. This abstract class takes two template parameters. The first specifies the implementation of the iterator that can be used to enumerate the index at this level. The second specifies the implementation of the

Abstract class	Methods
`term_scalar<V>`	`operator V()`
`term_nesting<I,E>`	`I begin()`, `I end()` `E subterm(I)`

`term_nesting2<I1,I2,E>`	`I1 begin1()`, `I1 end1()`
	`I2 begin2()`, `I2 end2()`
	`E subterm(I1, I2)`

...	
`term_map<K,E>`	`K map(E::index_type)` `E subterm()`
`term_aggregation2<E1,E2>`	`E1 subterm1()` `E2 subterm2()`
...	
`term_perspective2<E1,E2>`	`E1 subterm1()` `E2 subterm2()`
...	

Figure 15: Interfaces for view productions

substructure below this level. An implementation of CSR, in which the entries within each row are stored in order, that inherits from `term_nesting` is shown in Figure 16. `interval_iterator` and `offset_iterator` are two iterator abstract classes that are described later.

```
template<class ELT>
class Csr
    : public term_nesting< interval_iterator<int>,
                           CsrRow<ELT> > {
    // ...
};
template<class ELT>
class CsrRow
    : public term_nesting< CsrRowIterator<ELT>,
                           ELT > {
    /// ...
};
template<class ELT>
class CsrRowIterator :
    public offset_iterator<int> {
    // ...
};
```

Figure 16: CSR: Ironman API

A flat hierarchy, like $< \ r, c \ > \to \cdots$ is specified by inheriting from the `term_nesting` abstract class and specifying that its iterator enumerates indices of type `pair<int,int>`. This is illustrated by the implementation of Co-ordinate storage shown in Figure 17.

```
template<class ELT> class CooIterator;
template<class ELT>
class Coo
    : public CooRandom<ELT>,
      public term_nesting< CooIterator<ELT>,
                           ELT > {
    // ...
};
template<class ELT>
class CooIterator :
    public unordered_iterator< pair<int,int> > {
    // ...
};
```

Figure 17: COO: Ironman API

```
// Dense matrix storage
template<class ELT>
class Dense
    : public term_nesting2< interval_iterator<int>,
                            interval_iterator<int>,
                            ELT > {
    // ...
};
```

Figure 18: Dense: Ironman API

A term, like $< r \times c > \to \cdots$, has two independent iterators. To specify these sorts of views, `term_nesting2`, etc., abstract classes are provided which allow the implementation of each independent iterator to be specified. An implementation of dense matrices that uses the `term_nesting2` interface is shown in Figure 18.

By a very simple analysis of these classes, the sparse compiler

Abstract class	Methods	Properties
`unordered_iterator<K>`	`K operator *()` `void operator ++()`	no ordering

`increasing_iterator<K>`,
`decreasing_iterator<K>`
	`K operator *()`
	`void operator ++()`, or
	`void operator --()`
		one-way ordering

inherits from ↑ `ordered_iterator<K>`		bi-directional ordering

inherits from ↑
`offset_iterator<K>`
	`int operator -(iterator)`
	`void operator +=(int)`
	`void operator -=(int)`
		ordered, with distance

inherits from ↑ `interval_iterator<K>`		range of keys

Figure 19: Interfaces for iterators

can infer the following relationships,

```
Coo: // <r,c> -> v
    term_nesting< unordered_iterator< pair<int,int> >,
                  ELT >
Csr: // r -> c -> v
    term_nesting< interval_iterator<int>,
                  term_nesting< offset_iterator<int>,
                                ELT > >
Dense: // <r x c> -> v
    term_nesting2< interval_iterator<int>,
                   interval_iterator<int>,
                   ELT >
```

which clearly indicate the nested structure of these formats, and the properties of the iterators that are used at each level.

Interfaces for expressing perspective, aggregation and map are also available.

5.2 Interfaces for Iterators, revisited

The abstract classes for the iterators are described in Figure 19.

Unlike the iterators in Section 3, iterators in the Ironman API are used for enumerating indices only. That is, they do not provide the methods for accessing the substructures. Instead, the substructures are obtained via the `subterm` method in each `term_nesting` class. This is done, because whenever two independent iterators appear in a level of the hierarchy, (e.g., the dense matrix storage format), the matrix elements are associated with two indices from two different iterators. Since, in this case, the value is not associated with a single iterator, it cannot be accessed via a method in either iterator. Thus, the method for accessing the value is placed in the `term_nesting` classes.

We also refine the iterators discussed in Section 3 to account for more ordering properties. In addition to unordered, increasing, and decreasing iterators, we provide the `offset_iterator` interface for iterators whose positions can be randomly accessed, similar to the `random_access_iterator`'s found in the STL. A further refinement of `offset_iterator` is the `interval_iterator`, which is used to represent all of the integer indices between a fixed lower and upper bound.

6 Overview of the Compiler

We now give an overview of the restructuring techniques used in our compiler. The interested reader can refer to our other papers ([41], [20], [26]) for more detailed descriptions of these techniques.

Our view of sparse matrix formats as indexed-sequential access structures leads naturally to a restructuring technology based on *relational algebra* [38]. The highlights of this approach are as follows.

- Sparse matrices are modeled as relations in which the array indices and value are the fields of the relation, and each non-zero entry of the matrix has an associated tuple in the relation.
- The loops of the computation are modeled as expressions in a relation algebra [8, 38].
- Efficient evaluation strategies for these relational algebra expressions are found using query optimization techniques [33].
- The indexing structure of a sparse matrix format is exposed to the query optimizer through the type structure discussed in Section 5. This separation of algorithm and data structure allows changes to be made to the formats without requiring changes to the input program.

We sketch our compiler technology using the simple example of matrix-vector product in which A is stored in CRS, and X and Y are stored as sparse vectors.

Query Formulation The first task of the compiler is to translate the input generic program into a suitable intermediate representation. The intermediate representation of a loop describes the iterations in which there is work to do, but does not take a position on the order in which these iterations should be done.

$$\texttt{for } < a, x, y > \in \Pi_{<a,x,y>}(A(i,j,a) \bowtie X(j,x) \bowtie Y(i,y)) \ \{$$
$$y \texttt{ = } y + a * x$$
$$\}$$

This intermediate program says that the relations "A", "X" and "Y" are to be *joined*[5] on their common fields (i between A and Y, j between A and Y), and the resulting tuples are to have all fields except the value fields, a, y, and x, projected away. This computation produces another relation, and the body of the loop is to be executed for each tuple in that relation with appropriate bindings for a, x and y.

Join Scheduling The next task is to determine the order in which the joins must be performed. The $\bowtie$ operator is associative and commutative, so there are several possibilities. In our example, there are two basic, non-trivial strategies:

$$(A \bowtie_j X) \bowtie_i Y$$
$$(A \bowtie_i Y) \bowtie_j X$$

Which strategy is more efficient depends on the formats used to store the sparse data structures. If the compiler were to select the first strategy, then the join between A and X on the j field would be performed first, and then the join between the intermediate result and Y on i. However, in our example, the CRS format in which A is stored allows efficient access to the i index before the j index. Therefore, our compiler will pick the second strategy, which performs the join on i first.

The order in which a format's indices can be accessed is obtained directly from the format's hierarchy.

Join Implementation Once the order in which the joins are to be evaluated is determined, implementation strategies must be selected for each join. The choice of strategy depends on what indexing structures are

[5] To be precise, this is the *natural join* in database terminology.

available for searching the join field, or what properties hold for the enumerating the join field. Our compiler can obtain this information directly from the term of the hierarchy in which the join index appears.

In our example, the choice of join implementations depends upon the details of the formats used to store A, X, and Y. If, for instance, the elements of X and each row of A are stored in sorted order, then a constant time merge-join between the X and each row of A is possible. Otherwise, the elements of X could be scattered into a dense vector that, for the cost of $O(n)$ storage, would provide a constant time index for a hash-join with A.

Method Instantiation The final step of the query optimization process is to replace method invocations within the query evaluation plan with code provided by the storage format to implement the access. The result of this step, which is essentially procedure inlining, is an executable program for evaluating the query.

While there are many similarities between our restructuring techniques and database query optimization, there are also many profound differences. Some of these differences are the following:

- In databases, multiple, separate but simple indices are usually provided for accessing a relation. In contrast, sparse matrix formats usually provide a single, multi-level indexing structure.

- Complicated array references, such as $A[3j + 10, 4i - k]$, can appear in matrix programs, and these give rise to joins with general affine constraints [41]. Such constraints are handled using integer linear programming techniques in our compiler. These kinds of complicated constraints are called θ-joins in the database literature, but they are quite rare in database applications.

- In database systems, the dominant cost is usually disk I/O. In a sparse matrix computation, the dominant cost is usually cache and memory access. As a result, in order to produce efficient code, our compiler must employ a different set of low-level code optimizations than are found in a RDBMS.

7 System Design and Experiments

In designing our system, we set the following goals.

- The end-user of our system, namely the programmer who selects the specific sparse matrix format for which a generic algorithm is to be implemented, should be presented with a simple mechanism with which to invoke our sparse compiler.

- Our sparse compiler should work as a single tool within a suite of tools of a larger generic programming system. In particular, our sparse compiler should work cooperatively with an underlying C++ compiler. Our sparse compiler should handle the sparse matrix computations, and leave the other generic programming problems to the C++ compiler.

- Our sparse compiler should knit implementations for sparse matrix computations that are as efficient, and hopefully more so, than those that the programmer might have written by hand.

We are building our system as a source-to-source transformation tool. That is, the user will first run their program though our sparse compiler, which will instantiate some

```
#pragma instantiate with Bernoulli
template <class T, class ELT>
void mvm(T A, ELT x[], ELT y[])
{
    for (int i=0; i<A.rows(); i++) {
        y[i] = 0;
        for (int j=0; j<A.columns(); j++)
            y[i] += A[i][j] * x[j];
    }
}
// Will be instantiated with the Bernoulli compiler.
template void mvm(Csr<double> A, double x[], double y[]);
```

Figure 20: Generic MVM with instantiation

of the the template definitions. The programmer uses pragmas, as shown in Figure 20, to indicate which template definitions are to be instantiated by the sparse compiler; the rest are left untouched. The sparse compiler will generate a transformed C++ program to be run through the underlying C++ compiler, which will perform the remaining instantiation and usual optimizations.

7.1 Experimental results

We have implemented a prototype of the Bernoulli Sparse Compiler that provides the core restructuring and optimizations required for generating efficient code. The restructuring algorithms are described in [26, 40, 41]. Our current work is focused on integrating this compiler with a C++ front-end.

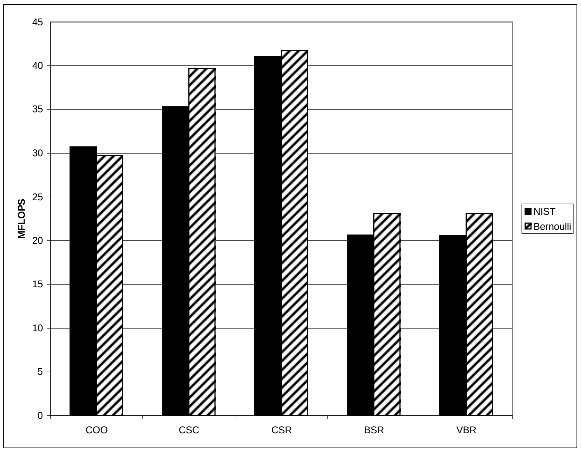
Even though the implementation is not complete, it can automatically instantiate most of the Sparse BLAS codes.

First, we compared code produced by the Bernoulli compiler with the NIST C implementations of two algorithms—matrix-vector multiplication and unit-diagonal triangular solve for five sparse matrix formats—co-ordinate (COO), Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), Block Sparse Row (BSR), and Variable-size Block sparse Row (VBR). As input we used the matrix `can_1072` from the Harwell-Boeing collection [7]. It arises in finite-element structures problems in aircraft design and has 1072 rows and columns and 12444 nonzero entries. For the block formats we used the sparsity pattern of the same matrix but expanded each entry into a 15×15 block.

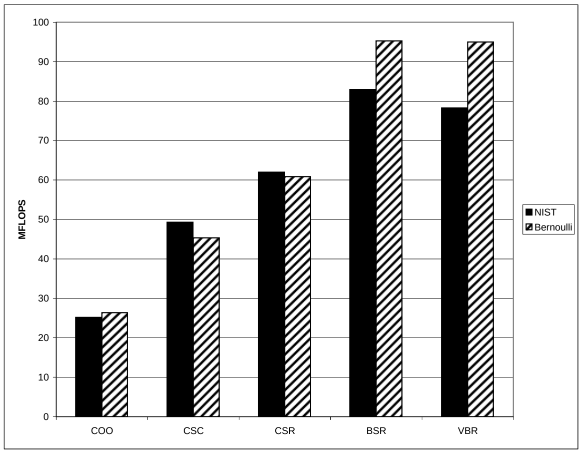
We also used the Bernoulli Sparse Compiler to generate code for the entire Conjugate Gradient (CG) iterative solver. Since the NIST C Sparse BLAS does not provide this routine, we also hand-wrote versions of CG for each of the storage formats, which called the appropriate NIST C Sparse BLAS routines to perform the kernel computations. The point that we wish to make here is that our approach scales from simple loop nests, like MVM and TS, to much larger computations.

We ran the experiments on two platforms—a Pentium II and a wide node of the IBM SP-2 at Cornell Theory Center. The Pentium II runs at 300 MHz and has 512 KB of L2 cache and 256 MB of RAM. The operating system is RedHat Linux 5.2. We compiled the code with `egcs` version 1.1.1 with `-O4 -malign-double -mpentiumpro` compiler flags. The wide node of the SP2 has a POWER2 Super Chip processor running at 135 MHz clock speed, 128 KB data cache, 256 bit memory bus, and 1 GB of memory. We used the `xlc` compiler version 3.1.4.7 with `-O3 -qarch=pwr2 -qmaxmem=-1` flags on AIX 4.2.

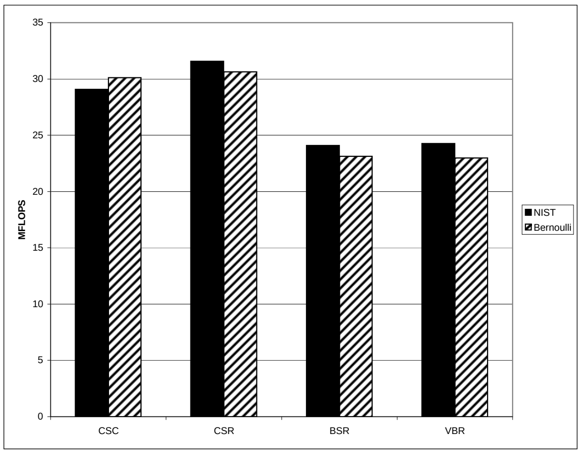
Figure 21 presents the performance of the hand-written NIST C code (dark bars) and the code generated by the Bernoulli Sparse Compiler (shaded bars). These results clearly show that the generic programming approach can successfully compete with hand-written library code. Indeed, Bernoulli-generated code performance ranges between 96% and 113% of NIST's on the Pentium II and between 85% and 121% on the IBM SP-2. Moreover, examining the C code reveals that the Bernoulli compiler



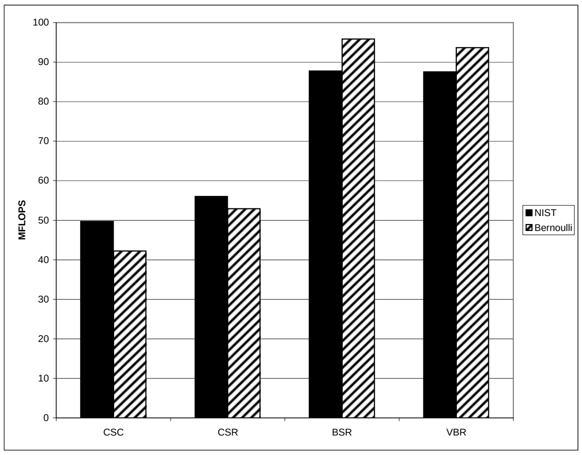
MVM on Pentium II



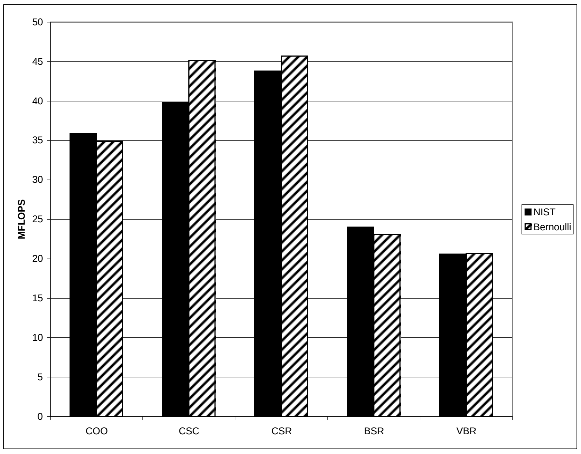
MVM on SP-2



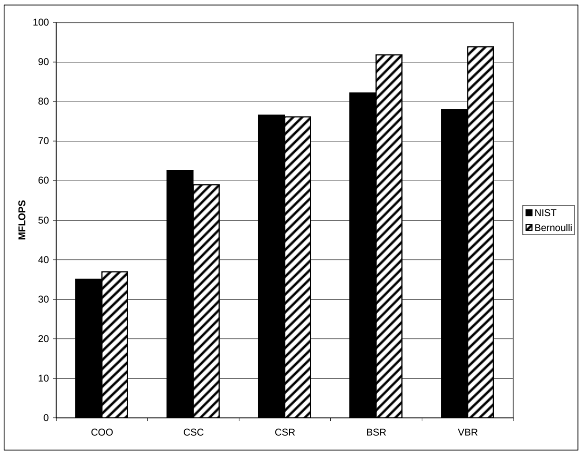
TS on Pentium II



TS on SP-2



CG on Pentium II



CG on SP-2

Figure 21: Performance measurements

in most cases produces code that is structurally identical to the hand-written one. There are minor syntactic differences—for example, the hand-written code would use `for (i=0;i!=m;i++) *pc++ = 0;` while the compiler generated `for (i=0;i<=m-1;i++) c[i]=0;`. These differences result in the hand-written code performing slightly better than the compiler-generated one when compiled with `egcs` and slightly worse when compiled with `xlc`.

We observed only three structural differences in the compiler-generated code. The hand-written implementation of CSC matrix-vector multiplication does not hoist a loop invariant. That omission is penalized by `egcs` and rewarded by `xlc`. The NIST implementation of triangular solve for CSR restructures the code in order to avoid initializing the output vector which gives it a small advantage on both platforms. The hand-written matrix-vector multiplication for the block formats contains a questionable guard that tries to avoid computation for zero entries in the vector. That improves the performance of the compiler-generated code by up to 21%.

8 Related Work

Generic Programming Our work is in the spirit of generic programming which has been described as "the idea of abstracting from concrete, efficient algorithms to obtain generic algorithms that can be combined with different data representations to produce a wide variety of useful software" [22]. The C++ STL library is an implementation of this philosophy in the realm of standard data structures like lists and trees [4]. Not only is our problem domain different, but at a deeper level, a key difference is that there is a single API in STL used by generic programmers and supported by data structure implementors, while in our system, the API used in writing generic algorithms is substantially different from the API that is supported by the implementors of compressed formats. *Supporting the dual API's effectively requires advanced restructuring compiler technology.*

Multiple *views* of a data structure are used extensively in database systems where it is common to support multiple views for relations and collections of relations [38]. Novak has argued for supporting multiple views for data structures as well, and has implemented a system that provides roughly the same functionality as the *map* view described in Section 4 [24].

Other researchers have recognized that the level of abstraction of programs can be raised by combining generic programming with more sophisticated compiler technology than is usually available for template instantiation. Our work is close in spirit to that of Batory and co-workers [34, 35] who have used similar ideas in designing the DiSTiL system, a software generator for container data structures. DiSTiL is a declarative language that extends C with constructs for specifying complex data structures declaratively. Data structures are specified by type equations that permit composition of DiSTiL components. When a DiSTiL program is compiled, these declarative specifications are replaced with efficient C implementations by the DiSTiL compiler. DiSTiL's goal is to support standard data structures, not sparse matrices, and no restructuring of code is done during the compilation process.

Aspect-oriented programming As we discussed in Section 2, the Strawman API presents a simple view of compressed formats that permits programmers to write generic code, but it does not by itself permit the compiler to generate efficient code. Subsequent developments in the paper were concerned with conveying additional information about compressed formats to the compiler in order to permit it to generate more efficient code. These additional properties *cross-cut* the `get/set` abstractions of the basic API, and are *aspects* in the terminology of Kiczales [3, 16].

Kiczales and others have designed *aspect-oriented extensions* to Java [21] to permit the expression of such aspects in Java classes in a modular fashion, using compiler technology to exploit aspects for generating efficient code. The key advantage is that resulting programs are simpler to read and maintain because algorithms and aspects are coded separately, and the algorithm is not cluttered with what are essentially implementation details. There are ongoing efforts to write sparse matrix factorization codes using these ideas [18, 30].

Restructuring Compilers Traditionally, restructuring compiler technology has been used to restructure dense matrix programs to enhance parallelism or locality of reference, but it cannot be used directly to restructure sparse matrix programs. This is because program analysis techniques are based on integer linear programming, and can be used only if all array subscripts are affine functions of loop index variables. Such subscripts are common in dense matrix programs in which arrays are accessed by row, column or diagonals, but are the exception in sparse matrix programs since sparse arrays are accessed through indirection arrays.

Bik and Wijshoff at Leiden University were the first to apply restructuring compiler technology to *synthesize* sparse matrix programs from dense matrix programs [1, 2]. Their compiler had knowledge of small number of formats built into it. The formats they considered can be called *Compressed Hyperplane Storage* (CHS) formats since they are obtained by doing a basis transformation on the dense array index space and then compressing out the non-zeros along one or more dimensions. CSR and CSC are therefore special cases of CHS formats. Their compiler analyzed and restructured the input code to match a CHS format, and generated sparse code for that format. The main limitation of this system is that it has a small set of relatively simple formats built into it, and it cannot be extended to new formats.

The PEI system of Perrin and coworkers [39] has taken essentially the same approach, but with more mathematically-oriented restructuring technology. Formats like CSR and CSC are described algebraically using basis transformations of dense array index spaces, spread/gather operations etc. The compiler then attempts to transform PEI programs to match these formats. As with the Leiden work, it is not clear that these techniques can be generalized to formats that are not CHS formats.

Sparse Matrix Libraries A number of projects in the numerical analysis community have exploited generic programming to support sparse matrix computations. PETSc [5, 27] is a successful library from Argonne which has a large collection of iterative solvers. These solvers must be linked with user-supplied BLAS that must be written for the particular sparse format of interest. The BLAS are invoked directly by PETSc code, so no special compiler support is needed for PETSc. In contrast, our system permits even the BLAS to be written in a generic, data-structure-neutral fashion, although at the cost of requiring aggressive restructuring compiler technology for generating efficient code.

POOMA [9, 25] and Blitz++ [6, 37] are two more recent packages for matrix computations. The API for both packages is essentially the Strawman API described in this paper. A rich set of C++ templates are provided in both packages, using which a programmer can assemble matrix implementations and produce matrix programs. Some optimizations can be performed by the compiler by relying on Template Expressions [36], but the range of such optimizations is limited, and they can be cumbersome to use. In particular, programmers must provide their own implementations of operations like MVM or triangular solve.

Similar efforts are ongoing in the functional languages community [13, 17]. Some of these projects such as Mona [17] are using the idea of *higher-order functors* to build Computational Mathematics libraries. Functors are parameterized program modules that take modules as arguments and return modules as results (modules are similar to virtual classes in C++). Therefore, higher-order functors in SML are similar to templates in C++, but they have more formal semantics.

9 Conclusions

Some of the ideas in this paper can be applied to *dense matrix* programs to obtain code with good locality of reference. Intuitively, the connection is that on machines with a memory hierarchy, dense matrices are not random-access data structures either! We have developed restructuring technology called *data-centric transformations* for producing enumeration-based code for dense matrix problems, and we have shown that on many problems, it outperforms more conventional locality-enhancement techniques by a wide margin [19]. This data-centric technology has been incorporated into SGI's MIPSPro compiler product-line as of January 1999.

We have investigated the generation of parallel code for iterative methods, but we have not discussed this in the paper for lack of space. The interested reader is referred to the PhD dissertation of one of the authors [40].

We are currently investigating the applicability of the techniques described in this paper to direct methods like Cholesky factorization. Even though direct methods are inappropriate for large sparse problems, some preconditioning strategies for iterative methods, such as Incomplete Cholesky factorization, perform computations similar to direct methods. Codes for sparse direct methods usually exploit a lot of tricks to obtain efficiency [10], and it is unclear how many of these can be incorporated into restructuring compilers. One solution might be to lower the semantic level of the input code, so that the compiler does not have to start with dense matrix code but with code in which some restructuring has been done by the programmer. These issues remain to be investigated.

References

[1] Aart J.C. 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.

[2] Aart J.C. Bik and Harry A.G. Wijshoff. Simple quantitative experiments with a sparse compiler. In *Parallel algorithms for irregularly structured problems: third international workshop, IR-*

REGULAR '96, pages 249–262, Santa Barbara, CA, August 19–21 1996.

[3] Aspect-oriented programming, May 2, 1998. `www.parc.xerox.com/spl/projects/aop/`.

[4] Matthew Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, MA, 1998.

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

[6] Blitz++ home page, May 2, 1998. `http://monet.uwaterloo.ca/blitz/`.

[7] R.F. Boisvert, Roldan Pozo, K. Remington, R.F. Barrett, and J.J. Dongarra. *The Quality of Numerical Software: Assessment and Enhancement*, chapter Matrix Market: a web resource for test matrix collections, pages 125–137. Chapman and Hall, London, 1997.

[8] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[9] James A. Crotinger, Julian Cummings, Scott Haney, William Humphrey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy J. Williams. Generic programming in POOMA and PETE. In *Proceedings from Dagstuhl Seminar on Generic Programming*, April 27–May 1, 1998.

[10] James W. Demmel, Stanley C. Eisenstat, John R. Gilbert, Xiaoye S. Li, and Joseph W. H. Liu. A supernodal approach to sparse partial pivoting. Technical Report CSD-95-883, Computer Science Division, University of California, Berkeley, 1995.

[11] 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.

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

[13] Stephen Fitzpatrick, M. Clint, and P. Kilpatrick. The automated derivation of sparse implementations of numerical algorithms through program transformation. Technical Report 1995/Apr-SF.MC.PLK, Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, Northern Ireland, U.K., April 1995.

[14] Alan George and Joseph W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[15] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[16] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc

Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox Palo Alto Research Center, February 1997.

[17] H. Hong and W. Schreiner. HPGP: High-performance generic programming for computational mathematics by compile-time instantiation of higher order functors. Technical Report HPGP Project Report 96-01, Johannes Kepler University, 1997.

[18] John Irwin, Jean-Marc Loingtier, John Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. Technical Report SPL97-007 P9710045, Xerox Palo Alto Research Center, February 1997.

[19] Induprakas Kodukula and Keshav Pingali. An experimental evaluation of shackling and tiling for memory hierarchy management. In *Proceedings of the 1999 International Conference on Supercomputing*, Rhodes, Greece, June 20–25, 1999.

[20] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. Compiling parallel code for sparse matrix applications. In *Supercomputing '97*, San Jose, November 15–21, 1997.

[21] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ. In *ECOOP'98 Workshop Reader*, 1998. Springer-Verlag LNCS 1543.

[22] 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.

[23] NIST sparse BLAS: Sourceservice code request, November 4, 1998.

[24] Gordon Novak. Creation of views for reuse of software with different data representations. *IEEE Transactions on Software Engineering*, 21(5):993–1005, December 1995.

[25] Parallel object-oriented methods and applications, October 23, 1998. `http://www.acl.lanl.gov/pooma/`.

[26] Paul Stodghill. *A Relational Approach to the Automatic Generation of Sequential Sparse Matrix Codes*. PhD thesis, Cornell University, July 1997. also as Technical Report CORNELLCS:TR97-1635.

[27] PETSc – the Portable, Extensible Toolkit for Scientific computation, May 5, 1998. `http://www.mcs.anl.gov/petsc/`.

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

[29] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[30] 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.

[31] Yousef Saad. SPARSKIT version 2.0.

[32] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing, Boston, MA, 1996.

[33] P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proceedings ACM-SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[34] Yannis Smaragdakis and Don Batory. DiSTiL: A transformation library for data structures. In *1997 Usenix Conference on Domain-Specific Languages*, Santa Barbara, CA, October 1997.

[35] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin layers. In *ECOOP '98*, July 1998.

[36] Todd Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[37] Todd Veldhuizen. The Blitz++ array model. In *ISCOPE '98*, 1998. to appear.

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

[39] E. Violard and G.-R. Perrin. PEI: a language and its refinement calculus for parallel programmming. *Parallel Computing*, 18:1167–1184, 1992.

[40] Vladimir Kotlyar. Relational Algebraic Techniques for the Synthesis of Sparse Matrix Programs. Technical Report TR99-1732, Department of Computer Science, Cornell University, March 1999.

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

[42] Michael Wolfe. *High Performance Compilers*. Addison-Wesley, Redwood City, CA, 1996.