Next-generation Generic Programming and its Application to 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

The contributions of this paper are the following.

1. We introduce a new variety of generic programming in which algorithm implementors use a different API than data structure designers, the gap between the API's being bridged by restructuring compilers. One view of this approach is that it exploits restructuring compiler technology to perform a novel kind of template instantiation.
2. We demonstrate the usefulness of this new generic programming technology by deploying it in a system that generates efficient sparse codes from high-level algorithms and specifications of sparse matrix formats.
3. We argue that sparse matrix formats should be viewed as indexed-sequential access data structures (in the database sense), and show that appropriate abstractions of the index structure of common formats can be conveyed to a restructuring compiler through the type system of a modern language that supports inheritance and templates.

1. INTRODUCTION

Generic programming is a methodology for simplifying the development of libraries in which a set of algorithms have to be implemented for many data structures. Code explosion is avoided by mandating a common API which is (i) supported by all data structures, and (ii) used to express algorithms in a *generic*, data-structure-neutral fashion. For example, the C++ Standard Template Library (STL) [2] uses the API of one-dimensional sequences as the interface between data structures like arrays and lists, and algorithms like searching and sorting. The type systems of modern languages permit the data structure implementations and generic programs to be type-checked and compiled separately; a concrete implementation is produced by linking a generic program with a particular data structure implementation.

There is however a tension in the design of generic programming API's that becomes evident in problem domains such as sparse matrix computations. For dense matrices, highly efficient implementations of the Basic Linear Algebra Subroutines (BLAS) [6] are usually provided by hardware vendors. For sparse matrices, the problem of developing BLAS libraries is complicated by the fact that some forty or fifty *compressed formats* are used to avoid storing zeros. Many attempts at writing sparse BLAS libraries have been confounded by the code explosion problem [15; 5]. Although it appears that generic programming is the solution to this problem, it is not clear that an appropriate API can be designed for sparse matrix libraries. As we explain in this paper, a high-level API that allows the programmer to express generic matrix algorithms in a natural array notation hides details of sparse matrix formats from the compiler, so performance may suffer. On the other hand, a low-level API that exposes the details of compressed formats is not suitable for writing generic programs. This problem is likely to occur in other problem domains in which data structure properties must be exploited for high performance.

In this paper, we discuss one way to solve this problem.

1. We use *dual API's*: a high-level API for expressing generic algorithms, and a low-level API for exposing details of data structures that must be exploited to obtain high performance.

2. We use *restructuring compiler technology* to transform abstract programs written in terms of the high-level API into efficient programs which use the low-level API.

We describe this approach in the context of sparse matrix computations as follows. In Section 2, we present some important sparse algorithms and compressed formats, propose a simple API called the *Strawman API*, and sketch a generic programming system designed around this API. Intuitively, this API views sparse formats as *random access* data structures, which is inappropriate for sparse formats and therefore leads to very inefficient code, but it permits us to introduce key ideas simply.

We motivate the separation of algorithm API and data structure API by taking progressively more nuanced views of compressed formats. The desire for greater efficiency motivates the *Woodenman API* in Section 3. This API views sparse formats as *sequential access* data structures [18]. We make the case for a generic programming system in which generic programmers code for the Strawman API, but invoke a restructuring compiler which views sparse formats through the Woodenman API and restructures the generic program into efficient code. This approach improves code efficiency over the use of the Strawman Interface alone, but for some programs, the efficiency is still poor compared to library code.

In Sections 4 and 5, we present the final API, called the *Ironman API*, that views sparse formats as *indexed-sequential access* data structures [18]. Section 4 describes the indices of interest in compressed formats, while Section 5 describes the details of the Ironman API and gives an implementation of a generic programming

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

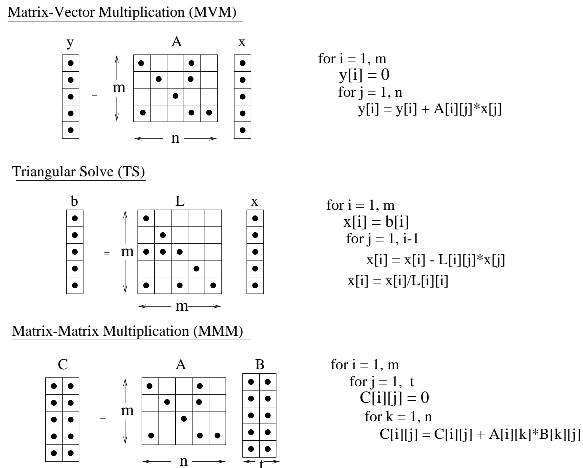


Figure 1: Basic Linear Algebra Subroutines

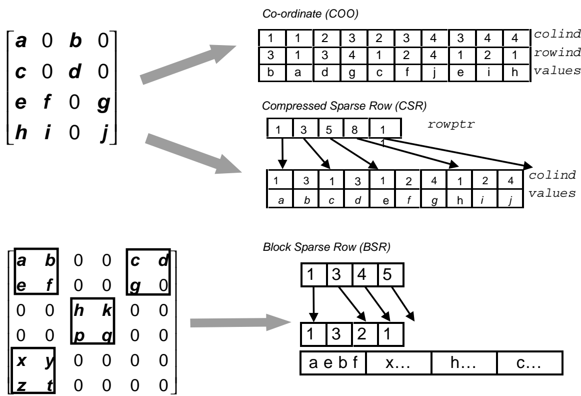


Figure 2: Compressed Formats

system that supports this API. We show that appropriate abstractions of the indexing structures of commonly used formats can be provided to such a system through the type system of a language like C++. Section 6 describes our restructuring compiler technology and presents experimental results that show that our approach can generate code competitive with handwritten code in the NIST Sparse BLAS library [5]. Finally, Section 7 discusses related and ongoing work.

2. A SIMPLE GENERIC PROGRAMMING SYSTEM: THE STRAWMAN API

The following algorithms constitute the Sparse BLAS.

- *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.

As mentioned earlier, there are at least forty or fifty commonly used

```
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`

compressed formats; the NIST Sparse BLAS effort [5] supports 13 of them. 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 ordered arbitrarily. Co-ordinate storage 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. *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.

Some sparse matrices have small dense blocks occurring in different positions inside the matrix. 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.

2.1 The Strawman API and Generic Programming System

The Strawman API requires each class implementing a compressed format to support two methods called `get` and `set`.

- The `get` method takes the row and column co-ordinates of an array element as input, and returns the value at that position.
- The `set` method takes a value and row/column co-ordinates as input, and stores the value into that position in the array.

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 expressed in C++[1]. 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. Figure 4 shows one implementation of co-ordinate storage (the implementation of the `set` method does not allow fill to keep the code simple). To write a generic program in this system, the programmer writes code as though all matrices were dense, but specifies which classes must be used to implement sparse matrices. For example, generic MVM is coded as shown in

[1] C++ has language features (namely, templates and inheritance) that allow us to express our API's and programs concisely. It is certainly possible to take the basic ideas in this paper and to realize them in other modern languages like Java or ML.

```
//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

Figure 5, and MVM for a particular compressed format is created by template instantiation.

2.2 Discussion

The Strawman API is very convenient for expressing algorithms in a data-structure-neutral fashion, but the efficiency of the code is poor for two reasons.

1. The `get` method is very inefficient because most compressed formats do not support efficient random access.
2. The 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 redundant.

As a concrete example of this inefficiency, we note that co-ordinate storage MVM code produced by this strategy requires $O(n^2 * NZ)$ time for a $n \times n$ matrix with NZ non-zeros, while the implementation in the NIST Sparse BLAS library [5] described in Section 3 takes only $O(NZ)$ time. We address these efficiency problems next.

```
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 6: Data-centric Pseudocode

3. A DATA-CENTRIC API: THE WOODEN-MAN INTERFACE

One approach to avoiding random accesses and computations with zeros is to recast algorithms in terms of *enumerations* of non-zero elements. Figure 6(a) shows such an algorithm for doing MVM; for each non-zero element `A[r][c]` of `A`, we compute the product `A[r][c]*x[c]` and add the result to `y[r]`. We call such algorithms *data-centric* [8] because their overall control structure is organized around enumerations of data structure elements.

Even though data-centric algorithms 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* [18] of a compressed format. However, *data-centric algorithms may not be correct if there are dependences between loop iterations*, as in triangular solve. Figure 6(b) shows data-centric pseudocode for triangular solve. From the original 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.

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 an order 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 and which achieve the efficiency of data-centric algorithms whenever possible.

3.1 The Need for Two API's

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 data-centric 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.

```
//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 7: Woodenman Interface

- *Enumeration order*: Intuitively, this is a description of the differences in the row/column co-ordinate values of successive elements in the enumeration.
- *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 data-centric 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.

These properties can obviously be expressed as systems of linear inequalities.

3.2 The Woodenman API

Figure 7 shows the Woodenman API. Enumeration is supported through the use of iterators as in the STL. A class implementing the `WoodenmanMatrix` interface is a container that 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 8 shows an implementation of Co-ordinate storage for the Woodenman API. To clarify the meaning of these classes, we show in Figure 9 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<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 8: COO: Woodenman API

```
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 9: Compiler-generated Code for MVM

3.3 Discussion

Figure 10 shows the performance of our enumeration-based codes for a number of compressed formats, compared to the performance of handwritten code in the NIST library, on the Pentium II platform described in detail in Section 6.3. For Co-ordinate storage, our 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, our 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

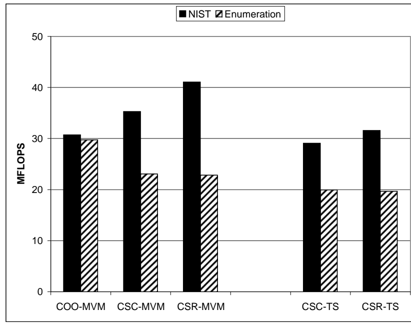


Figure 10: NIST vs. Woodenman API

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 [18] in which the `rowptr` array is an *index* (in the database sense) which permits efficient access to the non-zeros within a particular row. This leads naturally to a *nested* view of the data structure.

A compelling reason for viewing compressed formats as indexed-sequential access data structures is that exploiting indices makes a difference in the asymptotic time complexity of the 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 that permits insertions, such as a hash table. Enumeration-based 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 <r',c',vb> in non-zeros(B) do
    if (r' == c) C[r][c'] = C[r][c'] + va*vb;
  od
od
```

If `B` is viewed 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`.

We conclude that viewing compressed formats as sequential access data structures is a partial solution to the problem of compiling efficient code from generic dense-matrix programs. Improving efficiency further requires exploiting index structures in compressed formats.

4. INDEX STRUCTURE OF COMPRESSED FORMATS

Intuitively, an index structure for a compressed format corresponds to a particular *view* [18] of that data structure. The simplest index structures such as CSR use array co-ordinates themselves as indices. Some formats use indices that are not array co-ordinates but are obtained by applying a simple function to the array co-ordinates. 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 type structure in which there is one interface class for each production in the grammar.

4.1 Index Nesting

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. These expressions can obviously be generalized to arrays of arbitrary dimensions, and are 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.

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

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.

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

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.

Co-ordinate:	$< r, c > \rightarrow v$
CSR:	$r \rightarrow c \rightarrow v$
CSC:	$c \rightarrow r \rightarrow v$
Dense:	$< r \times c > \rightarrow v$

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.

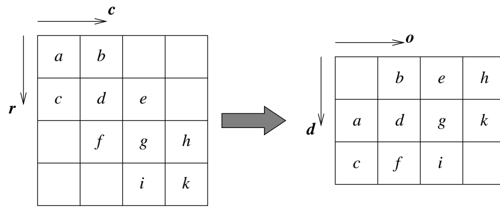


Figure 11: Diagonal Storage

- *Rotation:* In the diagonal storage format found in the Sparse BLAS, matrix elements are grouped and accessed by diagonals, as shown in Figure 11. 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 $d \to o \to v$, where the matrix dimensions, r and c, can be computed from d and o by,

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

- *Blocking:* Consider the Block Sparse Row (BSR) format shown in Figure 2. 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,

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

  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,

$$r = b_r * B + l_r$$
$$c = b_c * B + l_c$$

- *Permutation:* Often, sparse matrices are reordered in order to give their non-zeros 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.

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

For example, the view of the diagonal storage is:

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

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.[2]

$$< i, j > \to v$$
$$i \to j \to v$$

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.

$$E \quad : \quad E \oplus E$$

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.

$$E \quad : \quad E \cup E$$

4.5 Summary

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

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

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 12.

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 12(a). 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 $\to$ 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 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 13. `interval_iterator` and `offset_iterator` are two iterator abstract classes that are described later.

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

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()`

(a) Interfaces for Views

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

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

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

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

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

(b) Interfaces for Iterators

Figure 12: Interfaces for Ironman API

An index of the form $< r, c > \rightarrow \cdots$ is specified by inheritance 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 14.

An index like $< r \times c > \rightarrow \cdots$ 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. Figure 15 shows an implementation of dense matrices that uses the `term_nesting2` interface.

By a very simple analysis of these classes, the sparse compiler 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 12(b). 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`

```
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 13: CSR: Ironman API

```
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 14: COO: Ironman API

class. This is done, because whenever two independent iterators appear in a level of the index nesting, (e.g., in 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. The `interval_iterator` is a refinement of `offset_iterator`, which is used to represent all of the integer indices between a fixed lower and upper bound.

6. RESTRUCTURING COMPILER TECHNOLOGY AND PERFORMANCE

We now give a sketch of the restructuring compiler technology that converts programs written using the Strawman API into efficient programs that use the Ironman API. Intuitively, this restructuring must convert a high-level program into a data-centric program which (i) uses enumerations along the preferred directions of the sparse format and (ii) exploits indices, when possible. Our view of sparse matrix formats as indexed-sequential access structures leads naturally to a restructuring technology based on *relational algebra* [18]. For lack of space, we do not give the details of the compiler technology, which can found in an associated technical report [1]. Earlier versions of this compiler technology are also described in other publications [10; 17; 9]. The highlights of our approach are as follows.

- Sparse matrices are modeled as relations in which the array

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

Figure 15: Dense: Ironman API

```
#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 16: Generic MVM with Instantiation

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 [18].
- Efficient evaluation strategies for these relational algebra expressions are found using relational query optimization.
- The indexing structure of a sparse matrix format is exposed to the query optimizer through the type structure discussed in Section 5.

We are building our system as a source-to-source transformation tool. The user runs his program though our sparse compiler which instantiates some of the the template definitions. The programmer uses pragmas, as shown in Figure 16, 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 like inlining.

6.1 Restructuring Technology

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.

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

This intermediate program says that the relations "A", "X" and "Y" are to be *joined*[3] 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.

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

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:

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

The relative efficiency of the two strategies 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 index structure.

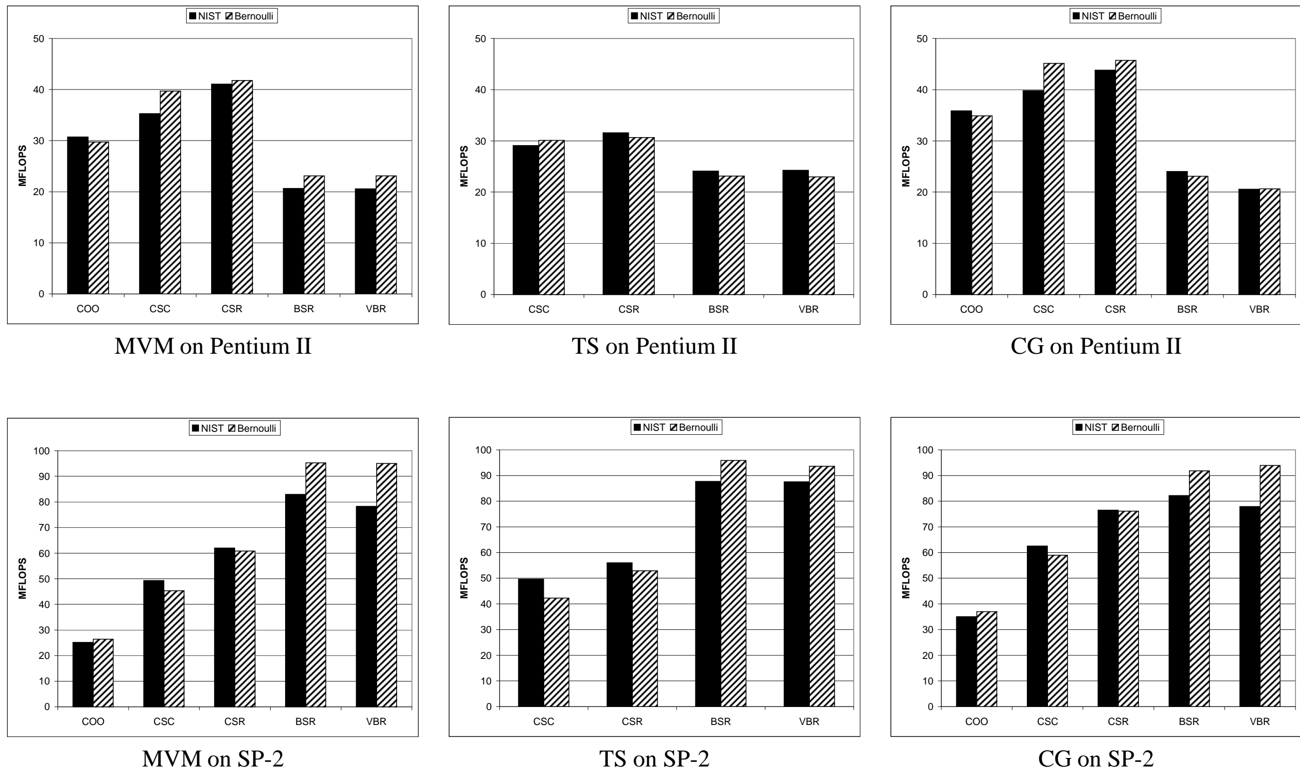
Join Implementation Once the order in which the joins are to be evaluated is determined, implementation strategies must be selected for each join. The choice of strategy depends on what index structures are available for searching the join field, and what properties hold for the enumerating the join field. Our compiler can obtain this information directly from the term of the index structure 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 merge-join [18] between 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 [18] 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.

6.2 Discussion

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 index 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 [10].
- 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, so the performance models are very different.



MVM on Pentium II | TS on Pentium II | CG on Pentium II

MVM on SP-2 | TS on SP-2 | CG on SP-2

Figure 17: Performance Measurements

6.3 Experimental Results

Our implementation of the Bernoulli Sparse Compiler is ongoing, but we have enough of it implemented to produce the following results.

First, we compared code produced by the Bernoulli compiler with the NIST Sparse BLAS 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). We used the matrix `can_1072` from the Harwell-Boeing collection [11] as input. 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 larger computations like CG.

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 17 presents the performance of the handwritten NIST C code (dark bars) and the code generated by the Bernoulli Sparse Compiler (shaded bars). These results demonstrate that the generic programming approach can successfully compete with handwritten 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 in most cases produces code that is structurally identical to the handwritten one. There are minor syntactic differences—for example, the handwritten 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 handwritten 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 code generated by the compiler. The handwritten 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 handwritten matrix-vector multiplication for the block formats contains a questionable guard that tries to avoid computation for zero entries in the vector. The absence of this guard improves the performance of the compiler-generated code by up to 21%.

7. RELATED WORK AND CONCLUSIONS

Generic Programming Our work is in the spirit of generic programming which is "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" [12]. An important difference from existing generic programming systems is that in our system, the API used in writing generic algorithms is different from the API that is supported by the implementors of compressed formats. *Supporting dual API's effectively requires advanced restructuring compiler technology and can be viewed as a sophisticated form of template instantiation.* In the terminology of *aspect-oriented programming* [7], index structures in compressed formats are *aspects* that cross-cut the get/set abstractions of the basic API. However, the existing effort on using aspect-oriented programming for sparse matrix computations [14] does not provide an API for supporting user-defined data structures.

Restructuring Compilers Bik and Wijshoff were the first to apply restructuring compiler technology to *synthesize* sparse matrix programs from dense matrix programs [3]. Their compiler restructured input codes to match a *Compressed Hyperplane Storage* (CHS) format (CSR and CSC are special cases of this format) whenever possible. The main limitation is that the compiler has a small set of simple formats built into it, so it cannot be extended to new formats.

Sparse Matrix Libraries POOMA [13] and Blitz++ [20] are two recent packages for matrix computations whose API is essentially the Strawman API described earlier. A rich set of C++ templates is provided, using which a programmer can assemble matrix implementations. Some optimizations can be performed by the compiler by relying on Template Expressions [19], but the range of such optimizations is limited. In particular, programmers must provide their own implementations of operations like MVM or triangular solve.
The MTL [16] is another C++ matrix library in which matrices are viewed as containers of containers. This idea is analogous to indexed sequential access, but not as rich as the structures that we discuss in this paper. Also, MTL does not have high- and low-level API's, as we do.

Ongoing Work We are currently investigating the applicability of the techniques described in this paper to direct methods like Cholesky factorization. Codes for sparse direct methods usually exploit a lot of domain-specific tricks to obtain efficiency [4], and it is unclear how many of these can be incorporated into a restructuring compiler. One solution might be to lower the semantic level of the input code, but these issues remain to be investigated.

8. REFERENCES

[1] Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. Compiling imperfectly-nested sparse matrix codes with dependences. Technical Report TR2000-1788, Cornell University, Computer Science, March 2000.

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

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

[4] 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.

[5] BLAS Techinal Forum. Sparse BLAS library: Lite and toolkit level specifications, January 1997. Edited by Roldan Pozo and Micheal A. Heroux and Karin A. Remington.

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

[7] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008 P9710042, Xerox Palo Alto Research Center, February 1997.

[8] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 16–18, 1997.

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

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

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

[12] 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.

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

[14] 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.

[15] Yousef Saad. SPARSKIT version 2.0.

[16] Jeremy G. Siek and Andrew Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE '98*, 1998.

[17] 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.

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

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

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

APPENDIX

A. AN EXAMPLE: JAGGED DIAGONAL

A.1 Overview

In this appendix, we present an extended example in order to illustrate the use of the interfaces that were presented in this paper.

The sparse matrix format that we will use for our example is the Jagged Diagonal (JAD) format. An instance of a JAD matrix may be constructed as follows. First, the rows of the matrix, as in Figure 18(a), are "compressed" so that zero elements are eliminated. This requires introducing an auxiliary array, `colind`, to maintain the original column indices. This is shown in Figure 18(b). Next, the rows of the compressed matrix are sorted by the number of non-zeros within each row in *decreasing* order. This requires introducing a permutation vector, `iperm`, as shown in Figure 18(c). Finally, the columns of the compressed and sorted matrix, which are called the "diagonals", are stored contiguously in two vectors, `colind` and `values`. The vector `dptr` is used to record the first index of the entries of each diagonal within `colind` and `values`. The final storage is shown in Figure 18(d).

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

A.2 Strawman API for JAD

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

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

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

```
/////////////////////////////////////////////////////////////
//                         JadRandom                       //
/////////////////////////////////////////////////////////////
template <class BASE>
class JadRandom : public matrix<BASE> {
protected:
    JadStorage<BASE> *A;
public:
    JadRandom(int m, int n, JadStorage<BASE> *A)
        : matrix<BASE>(m,n), A(A) { }
    virtual ~JadRandom() { }
    BASE *ref (int r, int c) {
        int rr = -1;
        for (rr=0; rr<A->n; rr++)
            if ((*A->iperm)[rr] == r) break;
        assert(rr != A->n);
        for (int d=0; d<A->nd; d++) {
            int jj_lo = (*A->dptr)[d];
            int jj_hi = (*A->dptr)[d+1];
            int jj = jj_lo + rr;
            if (jj >= jj_hi) break;
            if ((*A->colind)[jj] == c)
                return &(*A->values)[jj];
        }
        return 0;
    }
    virtual BASE get(int r, int c) {
        BASE *p = ref(r,c);
        if (p) { return *p; }
        else { return 0; }
    }
    virtual void set(int r, int c, BASE v) {
        BASE *p = ref(r,c);
        assert(p);
        *p = v;
    }
};
```

A.3 Ironman API for JAD

Using the grammar presented in Section 4, the following view can be used to describe the indexing structure of the JAD format.

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

The following classes implement the different pieces of the view.

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

We present the classes "inside-out".

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

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

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

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

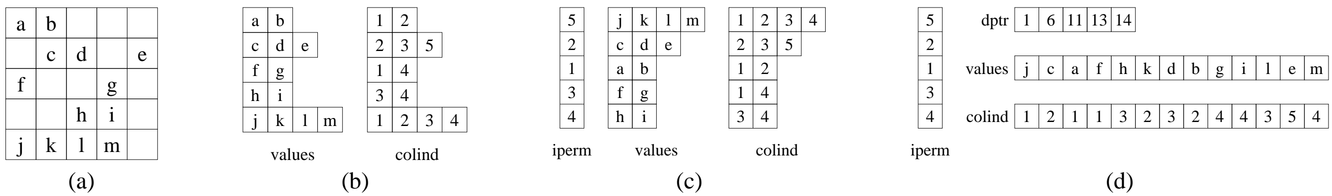


Figure 18: Building JAD Storage

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

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

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

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

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

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

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

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

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

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

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

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