A Generic Programming Case Study: Sparse Matrix Computations

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

Abstract

Generic programming looks like the perfect solution for writing sparse matrix programs as there are some forty or fifty compressed formats widely used in practice. However, it is unclear how to design a generic programming API for sparse matrices. A high-level API appropriate for generic matrix algorithms hides details of sparse matrix formats from the compiler, resulting in poor performance; while a low-level API that exposes the details of compressed formats is not suitable for writing generic programs.

We present 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.

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* fashion. For example, the C++ Standard Template Library (STL) [3] 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 some problem domains such as sparse matrix computations. For dense matrices, highly efficient implementations of the Basic Linear Algebra Subroutines (BLAS) [8] 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 [7, 18]. 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 [14], 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 [14], we proposed 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.

In our system, generic programs are dense matrix programs; *i.e.* the Programmer (high-level) API views sparse matrices as *random-access* data structures. The Compiler (low-level) API, on the other hand, views sparse matrices as *indexed-sequential-access* data structures. Our restructuring compiler [2] "instantiates" the generic programs into efficient sparse matrix programs.

The rest of the paper is organized as follows. In Section 2 we describe the abstract index structure of sparse

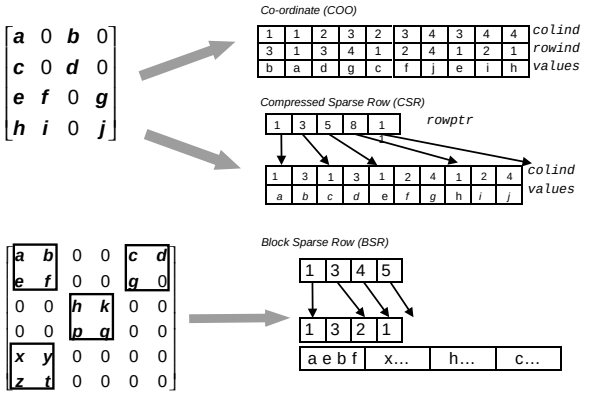

Figure 1: Compressed Formats

matrices used by our compiler. In Section 3 we introduce our generic programming system. We discuss our implementation of three important compressed formats in Section 4. In Section 5, we present the restructuring compiler technology that instantiates generic programs into efficient sparse codes. We present experimental results in Section 6. We conclude with related work review in Section 7.

2 Matrix Abstraction

As mentioned earlier, there are at least forty or fifty commonly used compressed formats; the NIST Sparse BLAS effort [7] supports 13 of them. Figure 1 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. Some of the rows of the matrix may be empty.

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 1 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.

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

Figure 2: Sparse Matrix Abstraction

2.1 Index Structure

The grammar in Figure 2 is used to describe the index structure of a sparse matrix to our system [14]. The most important rule for specifying index structure is the $\mathit{Index} \rightarrow E$ (*nesting*) production rule. For example, a CSR matrix is described as $r \rightarrow c \rightarrow v$, indicating that rows must be accessed first, and within each row, elements within columns can be enumerated. The $\mathit{map}\{F(\mathit{in}) \mapsto \mathit{out} : E\}$ rule is used to describe linear and permutation transformations on the matrix indices. The $E' \oplus E''$ (*perspective*) rule means that the matrix can be accessed in different ways, using either of the index structures E' or E''. The $E' \cup E''$ (*aggregation*) rule is used to describe a matrix that is a collection of two formats, such as a format in which the diagonal elements are stored separately from the off-diagonal ones. Enumerating the elements of such matrix requires enumerating both E' and E''.

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

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

- *Enumeration order*: a description of the order in which coordinate values could be enumerated efficiently. For the CSR format above, r is random-access, and within each row, c can be enumerated efficiently in increasing order.
- *Enumeration bounds*: a description of the coordinate values that actually occur in the enumeration. A lower triangular matrix, for example, could be annotated $1 \leq c \leq r \leq \mathtt{N}$.

In addition to specifying this index structure, the

sparse format designer must write the actual code to perform these enumerations.

3 Generic Programming System

The key feature of our generic programming system is that it uses *dual API's* for sparse matrices. The Programmer API provides a dense matrix (or random-access) view of a sparse matrix, and is used by the generic program writer. The Compiler API is a low-level API that describes a sparse matrix to the optimizing compiler as an indexed-sequential-access data structure. The restructuring compiler technology described in Section 5 instantiates the generic programs into efficient sparse matric codes.

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++[1] 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 of the the template definitions. The programmer uses pragmas, as shown in Figure 3, 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.

[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 other modern languages like Java or ML.

```
#pragma instantiate with Bernoulli
template <class T, class BASE>
void mvm(T A, BASE x[], BASE 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 3: Generic MVM with Instantiation

```
template<class BASE>
class matrix {
    int m; //number of rows
    int n; //number of columns
public:
    matrix(int r,int c) {m=r;n=c;}
    int rows() {return m;}
    int columns() {return n;}
    virtual BASE get(int r, int c) = 0;
    virtual void set(int r, int c, BASE v) = 0;

    // Implementation of 'A[r][c]' notation.
    class RowRef operator[](int r)
        { return RowRef(A,r) }
    ....
}
```

Figure 4: The Programmer API: `get/set`

3.1 The Programmer API

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

- The `get` method takes the row and column coordinates of an array element as input, and returns the value at that position.
- The `set` method takes a value and row/column coordinates 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 4 shows the Programmer API expressed in C++. Notice that operator overloading is used to permit programmers to use array syntax rather than invocations of the `get/set` methods.

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 Figure 3, and MVM for a particular compressed format is created by template instantiation.

3.2 The Compiler API

The Compiler API is summarized in Figures 5. Enumeration is supported through the use of iterators as in

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::index_type unmap(K)`
	`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 5: Interfaces for Compiler API

the STL. 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.

3.2.1 Interfaces for Views

Each production in the view grammar given in Figure 2 has an associated interface, which we have implemented in C++ as a small number of abstract classes described in Figure 5(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 $\rightarrow$ 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 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 Section 4.2.

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 Section 4.1.

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. The BSR implementation in Section 4.3 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> >,
              term_scalar< BASE > >

Csr: // r -> c -> v
term_nesting< interval_iterator<int>,
              term_nesting< increasing_iterator<int>,
                            term_scalar< BASE > > >
```

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.

3.2.2 Interfaces for Iterators

The abstract classes for the iterators are described in Figure 5(b).

Iterators in the Compiler 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 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.

In addition to `unordered_iterator`, `increasing_iterator`, and `decreasing_iterator` 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.

4 Compressed Formats

In this section we present C++ implementation of the API's of the three compressed formats introduced in Section 2. The formats were chosen to illustrate different features of our sparse matrix abstraction. The Co-ordinate format does not allow indexed access to either rows or columns. The Compressed Sparse Row format permits indexed access to rows but not columns. The Block Sparse Row format is a more complex format that illustrates linear maps and dense sub-matrices.

4.1 Co-ordinate Storage (COO)

Co-ordinate storage (COO) shown in Figure 1 is a simple format 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. Its index structure can be viewed as $< r, c > \rightarrow v$.

4.1.1 COO Programmer API

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

```
//////////////////////////////////////////////////
//                CooStorage                    //
//////////////////////////////////////////////////

template<class BASE>
struct CooStorage {
    vector<int> *rowind;
    vector<int> *colind;
    vector<BASE> *values;
    const int nz;

    CooStorage(vector<int> *_rowind,
               vector<int> *_colind,
               vector<BASE> *_values)
        : rowind(_rowind), colind(_colind),
          values(_values), nz(rowind->size()) {
    }
};
```

The `CooRandom` 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. As the non-zero elements in COO can appear in any order, `ref` uses linear search.

```
//////////////////////////////////////////////////
//                CooRandom                     //
//////////////////////////////////////////////////

template <class BASE>
class CooRandom : public matrix<BASE> {
protected:
    CooStorage<BASE> *A;

public:
    CooRandom(int m, int n, CooStorage<BASE> *A)
        : matrix<BASE>(m,n), A(A) { }

    virtual ~CooRandom() { }

    BASE *ref(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;
    }
    virtual BASE get(int r, int c) {
        BASE *p = ref(r,c);
        if (p) { return *p; }
        else { return 0.0; }
    }
    virtual void set(int r, int c, BASE v) {
        BASE *p = ref(r,c);
        assert(p);
        *p = v;
    }
};
```

4.1.2 COO Compiler API

The classes `Coo` and `CooIterator` implement the compiler API for the COO format. This API provides fast enumeration of the non-zero entries. As the COO format has no index structure, there is a single (unordered) iterator implementing it. That iterator corresponds to the $< r, c >$ term in the abstract view of COO.

```
//////////////////////////////////////////////////
//                     Coo                      //
//////////////////////////////////////////////////

template<class BASE>
class Coo
    : public CooRandom<BASE>,
      public term_nesting< CooIterator<BASE>,
                           term_scalar<BASE> >
{
protected:
    typedef CooIterator<BASE> iterator_type;
    typedef term_scalar<BASE> subterm_type;
public:
    Coo(int m, int n, CooStorage<BASE> *A) :
        CooRandom<BASE>(m,n,A) { }
    virtual iterator_type begin()
        { return CooIterator<BASE>(A,0); }
    virtual iterator_type end()
        { return CooIterator<BASE>(A,A->nz); }
    virtual subterm_type subterm(iterator_type it)
        { return (*A->values)[it.jj]; }
};

//////////////////////////////////////////////////
//                   CooIterator                //
//////////////////////////////////////////////////

template<class BASE>
class CooIterator :
    public unordered_iterator<pair<int,int> > {
    friend class Coo<BASE>;
protected:
    CooStorage<BASE> *A; int jj;
public:
    CooIterator(CooStorage<BASE> *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 bool equal(
        const proto_iterator<pair<int,int> > &y)
        const
        { return jj ==
              dynamic_cast<const CooIterator &>
              (y).jj; }

};
```

4.2 Compressed Sparse Row (CSR)

Compressed Sparse Row storage (CSR) in Figure 1 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. Using the view grammar in Figure 2, CSR can be described as $r \rightarrow c \rightarrow v$.

4.2.1 CSR Programmer API

As with the Co-ordinate format, the structure `CsrStorage` is used to hold all of the components of the CSR storage within a single object.

```
//////////////////////////////////////////////////
//                  CsrStorage                  //
//////////////////////////////////////////////////

template<class BASE>
struct CsrStorage {
public:
    vector<int> *rowptr;
    vector<int> *colind;
    vector<BASE> *values;
    const int n;
    const int nz;

    CsrStorage(vector<int> *_rowptr,
               vector<int> *_colind,
               vector<BASE> *_values)
        : rowptr(_rowptr), colind(_colind),
          values(_values), n(rowptr->size()-1),
          nz(colind->size()) {
    }
};
```

The `CsrRandom` class implements the programmer interface for the matrix by implementing the `get` and `set` abstract methods. The method `ref` shown here finds a particular (r, c) entry within the matrix by linear search within the row r. A binary search could also be used, as entries within a row are sorted by column index.

```
//////////////////////////////////////////////////
//                 CsrRandom                    //
//////////////////////////////////////////////////

template <class BASE>
class CsrRandom : public matrix<BASE> {
protected:
    CsrStorage<BASE> *A;

public:
    CsrRandom(int m, int n, CsrStorage<BASE> *A)
        : matrix<BASE>(m,n), A(A) { }

    virtual ~CsrRandom() { }

    virtual BASE *ref(int r, int c) {
        for (int jj=(*A->rowptr)[r];
             jj<(*A->rowptr)[r+1];
             jj++)
            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 0.0; }
    }
    virtual void set(int r, int c, BASE v) {
        BASE *p = ref(r,c);
        assert(p);
        *p = v;
    }
};
```

4.2.2 CSR Compiler API

Class `Csr` is the top-level class implementing the compiler interface for the CSR format. It provides access to the rows of the sparse matrix and corresponds to the $r \rightarrow \cdots$ term in the abstract view. As the array `rowptr` provides random access to a particular row in the matrix, this nesting level is described to the compiler as `interval_iterator`.

```
//////////////////////////////////////////////////
//                      Csr                     //
//////////////////////////////////////////////////

template<class BASE>
class Csr
    : public CsrRandom<BASE>,
      public term_nesting< interval_iterator<int>,
                           CsrRow<BASE> >
{
protected:
    typedef interval_iterator<int> iterator_type;
    typedef CsrRow<BASE> subterm_type;
public:
    Csr(int m, int n, CsrStorage<BASE> *A)
        : CsrRandom<BASE>(m,n,A) { }
    virtual iterator_type begin()
        { return interval_iterator<int>(0); }
    virtual iterator_type end()
        { return interval_iterator<int>(rows()); }
    virtual subterm_type subterm(iterator_type it)
        { return CsrRow<BASE>(A,*it); }
};
```

The classes `CsrRow` and `CsrRowIterator` provide access to the non-zero elements within a row of the CSR matrix. They implement the $c \rightarrow v$ part of the abstract view. The `increasing_iterator` tells the compiler that elements within a row are sorted.

```
//////////////////////////////////////////////////
```

```
//                    CsrRow                          //
////////////////////////////////////////////////////////

template<class BASE>
class CsrRow
    : public term_nesting< CsrRowIterator<BASE>,
                           term_scalar<BASE> >
{

protected:
    CsrStorage<BASE> *A; int r;
    typedef CsrRowIterator<BASE> iterator_type;
    typedef term_scalar<BASE> subterm_type;
public:
    CsrRow(CsrStorage<BASE> *A, int r)
        : A(A), r(r) { }
    virtual iterator_type begin()
        { return CsrRowIterator<BASE>(
            A,(*A->rowptr)[r]); }
    virtual iterator_type end()
        { return CsrRowIterator<BASE>(
            A,(*A->rowptr)[r+1]); }
    virtual subterm_type subterm(iterator_type it)
        { return (*A->values)[it.jj]; }
};

////////////////////////////////////////////////////////
//                  CsrRowIterator                    //
////////////////////////////////////////////////////////

template<class BASE>
class CsrRowIterator :
    public increasing_iterator<int> {
    friend class CsrRow<BASE>;
protected:
    CsrStorage<BASE> *A; int jj;
public:
    CsrRowIterator(CsrStorage<BASE> *A, int jj)
        : A(A), jj(jj) { }
    virtual void operator ++(int) { jj++; }
    virtual int operator*()
        { return (*A->colind)[jj]; }
    virtual bool equal(const proto_iterator<int> &y)
        const
        { return jj ==
              dynamic_cast<const CsrRowIterator &>
              (y).jj; }
};
```

4.3 Block Sparse Row (BSR)

The *Block Sparse Row* (BSR) in Figure 1 is a generalization of CSR in which the non-zeros are small dense blocks rather than single non-zero elements. 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 (o_r, o_c). The view of BSR can be expressed as $map\{b_r * B + o_r \mapsto r, b_c * B + o_c \mapsto c : b_r \rightarrow b_c \rightarrow < o_r \times o_c > \rightarrow v\}$.

4.3.1 BSR Programmer API

The structure `BsrStorage` holds all of the components of the BSR storage within a single object.

```
////////////////////////////////////////////////////////
//                  BsrStorage                        //
////////////////////////////////////////////////////////

template<class BASE>
struct BsrStorage {
public:
    vector<int> *browptr;
    vector<int> *bcolind;
    vector<BASE> *values;
    const int blk_sz;
    const int num_blocks;
    const int bnz;
    BsrStorage(vector<int> *_browptr,
               vector<int> *_bcolind,
               vector<BASE> *_values, int _blk_sz)
        : browptr(_browptr), bcolind(_bcolind),
          values(_values), blk_sz(_blk_sz),
          num_blocks(browptr->size()-1),
          bnz(bcolind->size()) {
    }
};
```

The `BsrRandom` class shows one possible implementation of the `get` and `set` which provide the programmer interface for the matrix. The block row b_r, as well as the offsets o_r and o_c, are accessed directly. Linear search is used for the block column b_c. Binary search could also be used as the blocks are sorted by column index within a block row.

```
////////////////////////////////////////////////////////
//                  BsrRandom                         //
////////////////////////////////////////////////////////

template <class BASE>
class BsrRandom : public matrix<BASE> {
protected:
    BsrStorage<BASE> *A;

public:
    BsrRandom(int m, int n, BsrStorage<BASE> *A)
        : matrix<BASE>(m,n), A(A) { }
    virtual ~BsrRandom() { }
    BASE *ref(int r, int c) {
        int bi = r / A->blk_size;
        int oi = r % A->blk_size;
        for (int bjj=(*A->browptr)[bi];
             bjj<(*A->browptr)[bi+1];
             bjj++) {
            int bj = (*A->bcolind)[bjj];
            int oj = c % A->blk_size;
            int j = bj*A->blk_sz + oj;
            int jj = (bjj * A->blk_sz*A->blk_sz) +
                     oi*A->blk_sz + oj;
            if (j == 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 0.0; }
    }
    virtual void set(int r, int c, BASE v) {
        BASE *p = ref(r,c);
        assert(p);
        *p = v;
    }
};
```

4.3.2 BSR Compiler API

The following classes are used to implement the compiler interface for BSR:

- `Bsr`: $map\{b_r * B + o_r \mapsto r, b_c * B + o_c \mapsto c : \cdots\}$
- `BsrHier`: $b_r \rightarrow \cdots$
- `BsrRow`: $b_c \rightarrow \cdots$

- `BsrBlock`: $< o_r \times o_c > \rightarrow v$

The top-level class `Bsr` is responsible for translating between the two-dimensional row/column view of the matrix and the four-dimensional internal representation.

```
////////////////////////////////////////////////
//                    Bsr                     //
////////////////////////////////////////////////

template<class BASE>
class Bsr
    : public BsrRandom<BASE>,
      public term_map< pair<int,int>, BsrHier<BASE> >
{
protected:
    typedef BsrHier<BASE> subterm_type;
public:
    Bsr(BsrStorage<BASE> *A)
        : BsrRandom<BASE>(A->num_blocks*A->blk_sz,
                          A->num_blocks*A->blk_sz,
                          A) { }
    virtual subterm_type subterm()
        { return BsrHier<BASE>(A); }
    virtual pair<int,int>
    map(quad<int,int,int,int> p) {
        return make_pair(
            p.first*A->blk_sz+ p.third,
            p.second*A->blk_sz+ p.fourth);
    }
    virtual quad<int,int,int,int>
    unmap(pair<int,int> p) {
        int i = p.first, j = p.second;
        int bi = i / A->blk_sz,
            oi = i % A->blk_sz;
        int bj = j / A->blk_sz,
            oj = j % A->blk_sz;
        return make_quad(bi,bj,oi,oj);
    }
};
```

The classes `BsrHier`, `BsrRow`, and `BsrRowIterator` implement the $b_r \rightarrow b_c \rightarrow \cdots$ part of the index structure of BSR. As expected, they are almost identical to the classes implementing the compiler API for CSR in Section 4.2.

```
////////////////////////////////////////////////
//                   BsrHier                  //
////////////////////////////////////////////////

template<class BASE>
class BsrHier
    : public term_nesting< interval_iterator<int>,
                           BsrRow<BASE> >
{
protected:
    BsrStorage<BASE> *A;
    typedef interval_iterator<int> iterator_type;
    typedef BsrRow<BASE> subterm_type;
public:
    BsrHier(BsrStorage<BASE> *A) : A(A) { }
    virtual iterator_type begin()
        { return interval_iterator<int>(0); }
    virtual iterator_type end()
        { return interval_iterator<int>(
            A->num_blocks); }
    virtual subterm_type subterm(iterator_type it) {
        return BsrRow<BASE>(A,*it); }
};

////////////////////////////////////////////////
//                    BsrRow                  //
////////////////////////////////////////////////

template<class BASE>
class BsrRow
    : public term_nesting< BsrRowIterator<BASE>,
                           BsrBlock<BASE> >
{
protected:
    BsrStorage<BASE> *A; int r;
    typedef BsrRowIterator<BASE> iterator_type;
    typedef BsrBlock<BASE> subterm_type;
public:
    BsrRow(BsrStorage<BASE> *A, int r)
        : A(A), r(r) { }
    virtual iterator_type begin()
        { return BsrRowIterator<BASE>(
            A,(*A->browptr)[r]); }
    virtual iterator_type end()
        { return BsrRowIterator<BASE>(
            A,(*A->browptr)[r+1]); }
    virtual subterm_type subterm(iterator_type it)
        { return BsrBlock<BASE>(A,it.jj); }
};

////////////////////////////////////////////////
//                BsrRowIterator              //
////////////////////////////////////////////////

template<class BASE>
class BsrRowIterator :
    public increasing_iterator<int> {
    friend class BsrRow<BASE>;
protected:
    BsrStorage<BASE> *A; int jj;
public:
    BsrRowIterator(BsrStorage<BASE> *A, int jj)
        : A(A), jj(jj) { }
    virtual void operator ++(int) { jj++; }
    virtual int operator*()
        { return (*A->bcolind)[jj]; }
    virtual bool equal(const proto_iterator<int> &y)
        const
        { return jj ==
              dynamic_cast<const BsrRowIterator &>
              (y).jj; }
};
```

The `BsrBlock` class implements the dense-matrix blocks in BSR. The independence of the o_r and o_c indices is described by `term_nesting2`. Both o_r and o_c are implemented as `interval_iterator`.

```
////////////////////////////////////////////////
//                  BsrBlock                  //
////////////////////////////////////////////////

template<class BASE>
class BsrBlock
    : public term_nesting2< interval_iterator<int>,
                            interval_iterator<int>,
                            term_scalar<BASE> >
{
protected:
    BsrStorage<BASE> *A;
    int bjj;
    typedef interval_iterator<int> iterator1_type,
        iterator2_type;
    typedef term_scalar<BASE> subterm_type;
public:
    BsrBlock(BsrStorage<BASE> *A, int bjj)
        : A(A), bjj(bjj) { }
    virtual iterator1_type begin1()
        { return interval_iterator<int>(0); }
    virtual iterator1_type end1()
        { return interval_iterator<int>(
            A->blk_sz);; }
    virtual iterator2_type begin2()
        { return interval_iterator<int>(0); }
    virtual iterator2_type end2()
```

```
        { return interval_iterator<int>(
            A->blk_sz);; }
    virtual subterm_type subterm(iterator1_type it1,
                                 iterator2_type it2){
        return (*A->values)[
            bjj*A->blk_sz*A->blk_sz +
            *it1*A->blk_sz + *it2]; }
};
```

5 Compiler Technology

The key to efficiency in sparse matrix computations is performing the computation in a *data-centric* way, *i.e.*, enumerating the non-zero elements of sparse matrices and performing computations with these elements as they are enumerated.

We illustrate our restructuring compiler technology on the generic matrix-vector multiplication in Figure 3. For the running example, we assume that the sparse matrix `A` is stored in CSR format, and has `M` rows and `N` columns. After the generic program is instantiated by our compiler, we have the C++ code shown in Figure 6.

5.1 Restructuring Imperfectly-nested Loops

The code in Figure 3 is imperfectly-nested—statement `S1: y[i] = 0;` is nested in the `i` loop but not in the `j` loop. We have developed compiler technology for restructuring imperfectly-nested loops in [1] and [2], here we present a short summary.

Our framework makes the usual assumptions about programs: (i) programs are sequences of statements nested within loops, (ii) all memory accesses are through array references, and there is no array aliasing, and (iii) all loop bounds and array indices are affine functions of surrounding loop indices and symbolic constants.

5.1.1 Dependences

At the very least, the restructured program must preserve the semantics of the original code. For our example, if we execute statement `S2: y[i] += A[i][j] * x[j];` before `y[i]` is initialized in statement `S1`, the resulting code will be wrong. Dependence analysis states that semantics of a program are preserved if all dependences are respected in the transformed program.

We will use `S1`, `S2`, ..., `Sn` to name the statements in the program in syntactic order. An *instance* $\vec{i}_k$ of a statement `Sk` is the execution of statement `Sk` at iteration $\vec{i}_k$ of the surrounding loops. We say that there exists a *data dependence* from instance $\vec{i}_s$ of statement `Ss` (the *source* of the dependence) to instance $\vec{i}_d$ of statement `Sd` (the *destination*) if (i) both instances lie within corresponding loop bounds; (ii) they reference the same

```
template <>
void mvm(Csr<double> &A, double x[], double y[])
{
    typedef Csr<double>::subterm_type row_type;

    for (int i = 0; i < A.rows(); i++)
        y[i] = 0.0;

    for (Csr<double>::iterator_type it_r = A.begin();
         it_r != A.end(); it_r++) {
        int r = *it_r;
        row_type Ar = A.subterm(it_r);
        for (row_type::iterator_type
             it_c = Ar.begin();
             it_c != Ar.end(); it_c++) {
            int c = *it_c;
            double v = Ar.subterm(it_c);
            y[r] += v * x[c];
        }
    }
}
```

Figure 6: Compiler-generated Code for MVM

memory location; (iii) at least one of them writes to that location; and (iv) instance $\vec{i}_s$ of statement `Ss` occurs before instance $\vec{i}_d$ of statement `Sd` in program execution order. Dependence constraints can be represented as a matrix inequality of the form $D(\vec{i}_s, \vec{i}_d)^T + \vec{d} \geq 0$. Such an inequality obviously represents a polyhedron. We call each such matrix inequality a *dependence class*, and denote it by $\mathcal{D}$ with some subscript.

For our running example in Figure 3, it is easy to show that there is one relevant dependence class[2] $\mathcal{D} = \{0 \leq i_1 < \mathtt{M}, 0 \leq i_2 < \mathtt{M}, 0 \leq j_2 < \mathtt{N}, i_1 = i_2\}$. It arises because statement `S1` writes to a location `y[i]` which is then read by statement `S2`.

5.1.2 Modeling Program Transformations

We model program transformations as follows. We map dynamic instances of statements to points in a Cartesian space $\mathcal{P}$. We then enumerate the points in $\mathcal{P}$ in lexicographic order, and execute all statements mapped to a point when we enumerate that point. If there are more than one statement instances mapped to a point, we execute these statement instances in original program order. Intuitively, the Cartesian space $\mathcal{P}$ models a perfectly-nested loop, and the maps model transformations that embed individual statements into this perfectly nested loop. It should be understood that this perfectly-nested loop is merely a logical device—the code generation phase produces an imperfectly-nested loop from the space and the maps.

For $\mathcal{P} = i \times j$, we can embed the code in Figure 3 into $\mathcal{P}$ using the embedding functions $F_1 = (i, 0)$ for statement `S1`, and $F_2 = (i, j)$ for statement `S2`. That embedding preserves the original program order.

[2] There is also a reduction dependence coming from the multiple writes to `y[i]` in statement `S2`. Reduction dependences are not important for preserving semantics, they are easy to recognize and eliminate.

Clearly, not all spaces and maps correspond to legal transformations. However, if the execution order of the transformed program respects all dependences (i.e. for each dependence, the source statement instance is enumerated and executed before the destination statement instance), then the resulting program is semantically equivalent to the original program. We must therefore address two problems.

What is the Cartesian space $\mathcal{P}$ for the transformed program? Each statement has an *iteration space* and a *data space*. The iteration space is a Cartesian space whose dimension is equal to the number of loops surrounding that statement. For our example, the iteration space of S1 is i, and the the iteration space of S2 is $i \times j$. The data space is a Cartesian space whose dimensions are the dimensions of all references to arrays on which we might want to be data-centric. In our context, these are the references in the statement to sparse arrays. In the example, matrix A is sparse and is stored in CSR format, so the data space of statement S2 is $a^r \times a^c$. We use the actual sparse data dimensions, *i.e.* if A was stored in BSR format, the data space of S2 would be $a^{b_r} \times a^{b_c} \times a^{l_r} \times a^{l_c}$, where the name of each dimension has been chosen to reflect its pedigree. The *statement space* of a statement is the product of its iteration space and data space. We denote the statement space of statement Sk by $\mathcal{S}_k$, and the coordinates of instance $\vec{i}_k$ in $\mathcal{S}_k$ by $(\vec{i}_k, \vec{d}_k)$. The statement space of S2 is $i_2 \times j_2 \times a_2^r \times a_2^c$. A *product space* $\mathcal{P}$ for a program is the Cartesian product of its individual statement iteration spaces. For the purposes of this paper, the order in which individual dimensions appear in this product is left unspecified, and each order corresponds to a different product space. A product space has 5 dimensions, and there are a total of $5!$ product spaces, corresponding to the different orders of dimensions of $\mathcal{P} = i_1 \times i_2 \times j_2 \times a_2^r \times a_2^c$.

Product spaces can be computed easily from the original code and the abstract index structure of sparse matrices, details are available in [2].

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

$$G = \begin{array}{c} \begin{matrix} i_1 & i_2 & j_2 \end{matrix} \\ \left[\begin{array}{c|cc} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right] \end{array} \begin{array}{c} \text{dimension} \\ a_2^r \\ a_2^c \\ i_1 \\ i_2 \\ j_2 \end{array}$$

Figure 7: Identifying Redundant Dimensions

Among the legal embedding functions for the product space $\mathcal{P} = i_1 \times i_2 \times j_2 \times a_2^r \times a_2^c$ are $F_1(i_1) = (i_1, i_1, 0, i_1, 0)^T$, $F_2(i_2, j_2, a_2^r, a_2^c) = (i_2, i_2, j_2, a_2^r, a_2^c)^T$, which correspond to the original execution order; as well as $F_1(i_1) = (i_1, -1, -1, -1, -1)^T$, $F_2(i_2, j_2, a_2^r, a_2^c) = (M+1, i_2, j_2, a_2^r, a_2^c)^T$, which correspond to initializing vector y before executing any instance of statement S2.

5.2 Generating Data-centric Code

We can think of a product space and embeddings as representing a perfectly-nested loop nest with guarded statements where we enumerate the values of all dimensions, and execute statement Sk when the values being enumerated match the embedding $F_k(\vec{i}_k, \vec{d}_k)$. However, this code will have very poor performance. To improve performance, it is necessary to (i) identify and eliminate redundant dimensions, and (ii) use common enumerations for related dimensions.

Redundant dimensions To give ourselves more flexibility to restructure the code, we introduced many dimensions in the product space. Now, after we have determined the embeddings, we can identify the dimensions we do not need and eliminate them. For our example, consider the (ordered) product space $\mathcal{P} = a_2^r \times a_2^c \times i_1 \times i_2 \times j_2$, and the embedding functions $F_1(i_1) = (-1, -1, i_1, -1, -1)^T$ and $F_2(i_2, j_2, a_2^r, a_2^c) = (a_2^r, a_2^c, M+1, i_2, j_2)^T$. As $a_2^r = i_2$ and $a_2^c = j_2$, the values of dimensions i_2 and j_2 of the product space are determined by the values of the preceding dimensions a_2^r and a_2^c. More generally, we identify redundant dimensions as follows.

Embedding functions are affine, and for each statement instance $(\vec{i}_k, \vec{d}_k)$, the data coordinates $\vec{d}_k$ are affine functions of the loop indices $\vec{i}_k$. We can therefore represent the embedding functions as $F_k(\vec{i}_k, \vec{d}_k) = G_k \vec{i}_k + \vec{g}_k$, where the matrix G_k defines the linear part of F_k, and the vector $\vec{g}_k$ is the affine part. We can use the matrix $G = [G_1 G_2 \ldots G_n]$ to identify redundant dimensions in the product space. We use G^k to refer to the k^{th} row of the matrix G. For our example, this matrix is shown in Figure 7.

If a row of the G matrix is a linear combination of preceding rows, the corresponding dimension of the

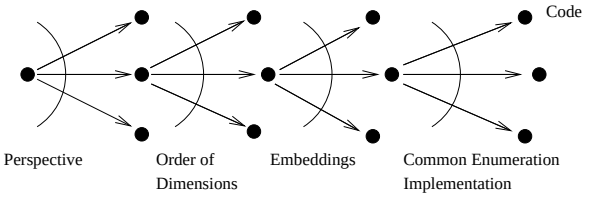

Figure 8: Search Space

product space is said to be *redundant*. In our example, dimensions i_2 and j_2 are redundant. It is not necessary to enumerate redundant dimensions since code is executed only for a single value in that dimension, and that value is determined by values of preceding dimensions; instead we can generate code to search for this value. For the example, the search for a particular value in dimensions i_2 and j_2 is a simple lookup.

Common enumerations An important optimization is recognizing groups of dimensions that could be enumerated together. In previous work [11], we developed technology for common enumeration of dimensions which are related through a single parametric variable (we called these *joinable* dimensions). We use common enumerations for groups of dimensions consisting of a non-redundant dimension, and redundant dimensions that immediately follow it and are linearly dependent on it. There are a number of ways of performing common enumerations which are closely related to join strategies in database systems such as merge-join and hash-join [11]. In the example, there is a single reference to a sparse matrix, and common enumerations do not arise.

The resulting data-centric code is the one in Figure 6.

5.3 Search Space and Cost Estimation

In theory, we can enumerate all legal enumeration-based codes as illustrated in Figure 8, then estimate the cost of each code, and select the best one.

Cost Figure 9 describes syntax for enumeration-based pseudo-codes.[3] Each syntax rule is annotated with its associated cost. *EnumCost* depends on whether we are enumerating the dimension in a direction supported by the format, or whether dependences force us to enumerate in a different direction. *SearchCost* depends on the type of enumeration method available for that dimension (*e.g.*, whether it is an interval, or whether the values are sorted). *CommonEnumCost* depends on what common enumeration implementations are available for the corresponding data dimensions.

[3]The *guard* conditionals arise because of loop bounds.

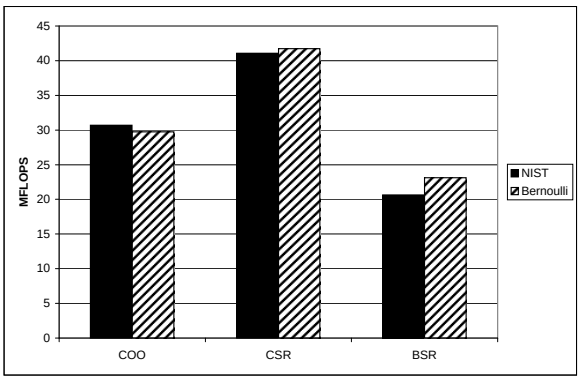

Figure 10: Performance Measurements: MVM

Heuristics to limit the search space Searching the full space of enumeration-based codes is impractical, but the following heuristics make the search space manageable.

Data-centric Execution Order: We only consider data-centric orders of dimensions of the product space (*i.e.,* orders in which all data dimensions come before any iteration space dimensions). The indexing structure of sparse matrices puts further restrictions on the dimensions orderings we need to consider. In our example, A is accessed through the index structure $r \rightarrow c \rightarrow v$, so our compiler does not consider product spaces in which c is enumerated before r.

Common Enumerations: Efficient sparse code enumerates the data as few times as possible, so our goal is to use a single enumeration of a sparse matrix, and execute all statements which reference that matrix. That restricts our choice of embedding functions to just three per dimension: a common enumeration with a matching dimension of another statement, or, if that is not legal, embedding the statement *before* or *after* the enumeration of the matching dimension.

6 Experimental Results

Our implementation of the Bernoulli Sparse Compiler is ongoing. The performance measurements we present here are from our earlier implementation that produces C code. For the final version of the paper, we will present the numbers from the C++ code generated by our compiler.

Additionally, the generic programming system that we have implemented does not use `virtual` methods, as do the examples in this paper. Instead, we use the Barton and Nackman trick [24] to ensure that all method invocations can be resolved to method definitions at compile time. See [13] for a more detailed discussion of the performance issues involved in implementing our sys-

S	:	`for` $i \in$ `enum`(*iterator*) `do` S	: *EnumCost*(*iterator*) $*$ *Cost*(S)
	|	`for` $i \in$ `enum`(*itr*$_1$, *itr*$_2$) `do` S	: *CommonEnumCost*(*itr*$_1$, *itr*$_2$) $*$ *Cost*(S)
	|	`if` ($i \in$ `search`(*iterator*)) `then` S	: *SearchCost*(*iterator*) $+$ *Cost*(S)
	|	`if` (*guard*) `then` $x = y$	: 1
	|	$S_1; S_2$	: *Cost*(S_1) $+$ *Cost*(S_2)

Figure 9: Cost Estimation

tem.

We compared code produced by the Bernoulli compiler with the NIST Sparse BLAS C implementations of matrix-vector multiplication, for the three compressed formats discussed in this paper. We used the matrix `can_1072` from the Harwell-Boeing collection [15] as input. It arises in finite-element structures problems in aircraft design and has 1072 rows and columns and 12444 nonzero entries. For the BSR format we used the sparsity pattern of the same matrix but expanded each entry into a 15×15 block.

The numbers presented here were collected on a Pentium II running at 300 MHz, with 512 KB of L2 cache and 256 MB of RAM. The operating system was RedHat Linux. We compiled the code with `egcs` version 1.1.1 with `-O4 -malign-double -mpentiumpro` compiler flags.

Figure 10 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. Bernoulli-generated code performance ranges between 97% and 112% of NIST's performance.

7 Related Work

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" [16]. 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.*

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 [20, 21] 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 The Programmer 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. the Compiler API conveys 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 [10].

Kiczales and others have designed *aspect-oriented extensions* to Java [12] 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 [9, 17]; however, they do not provide an API for supporting user-defined data structures.

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 [5]. 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.

Sparse matrix libraries A number of projects in the numerical analysis community have exploited generic programming to support sparse matrix computations. PETSc [4] 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 [6] and Blitz++ [23] are two more recent packages for matrix computations. The API for both packages is essentially the Programmer API described in this paper. A rich set of C++ templates are provided in both packages, with which a programmer can assemble matrix implementations and produce matrix programs. Some optimizations can be performed by the compiler by relying on Template Expressions [22], 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.

The MTL [19] 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.

References

[1] Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000. To appear.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] Nikolay Mateev, Keshav Pingali, and Paul Stodghil. Performance issues in generic programming with sparse matrices, March 2000. `http://www.cs.cornell.edu/Info/Projects/Bernoulli/papers/poocs00.ps`.

[14] Nikolay Mateev, Keshav Pingali, Paul Stodghill, and Vladimir Kotlyar. Next-generation generic programming and its application to sparse matrix computations. In *Proceedings of the 2000 International Conference on Supercomputing*, Santa Fe, New Mexico, May 8–11, 2000. To appear.

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

[16] 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.

[17] 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.

[18] Yousef Saad. SPARSKIT version 2.0.

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

[20] 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.

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

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

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

[24] Todd Veldhuizen. Techniques for scientific C++, version 0.3, August 1999. `http://extreme.indiana.edu/~tveldhui/papers/techniques/`.