A Generic Programming System for Sparse Matrix Computations

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

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

Abstract

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

1 Introduction

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

Algorithms for such matrix problems are classified broadly into direct methods and iterative methods [15]. Direct methods such as Cholesky, LU and QR factorizations factorize a matrix into two triangular matrices, and compute the required result from these triangular matrices. Except in special cases, the triangular factors usually have far more non-zeros than the original matrix (a phenomenon called *fill*), so the storage and computational needs of direct methods are

---

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

usually prohibitive for large sparse matrices [14]. In contrast, iterative methods like Conjugate Gradient and Lanczos methods do not modify the matrix, so they do not suffer from fill. Therefore, iterative methods are increasingly the methods of choice for large sparse matrix problems [32].

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

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

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

---

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

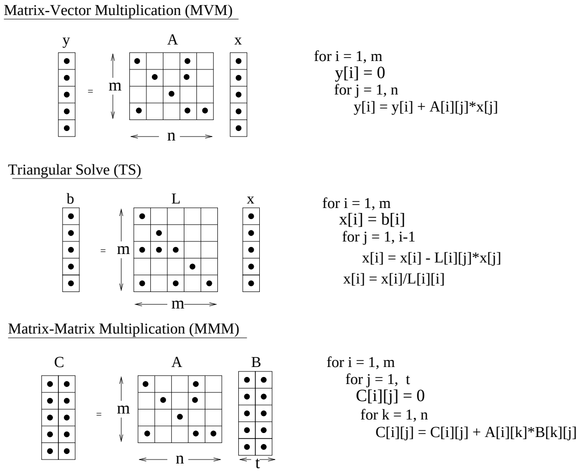
[2] The BLAS are described in more detail in Section 2; for now, they can be considered to be basic matrix computations which must be coded very differently for different compressed formats.

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

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

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

In Sections 4 and 5, we present the final API, called the *Ironman API*, that views sparse formats as *indexed-sequential access* data structures [38]. Section 4 describes the indices of interest in compressed formats, while Section 5 describes the details of the Ironman API and gives an implementation of a generic programming system that supports this API. We have a prototype of this system implemented, and we are currently reengineering it so that it takes generic programs written in C++ as input. Section 6 gives a brief introduction to the compiler technology used within the system, and Section 7 presents experiments with code generated by our existing system that show that our approach can generate code competitive with handwritten code in the Sparse BLAS library. Section 8 discusses related work, while Section 9 describes ongoing work.



```
for i = 1, m
    y[i] = 0
    for j = 1, n
        y[i] = y[i] + A[i][j]*x[j]
```

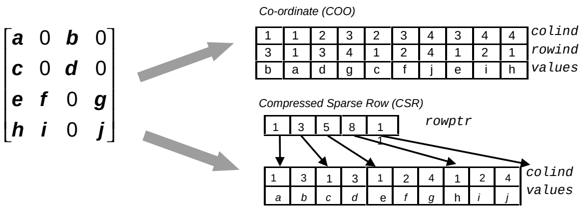
```
for i = 1, m
    x[i] = b[i]
    for j = 1, i-1
        x[i] = x[i] - L[i][j]*x[j]
    x[i] = x[i]/L[i][i]
```

```
for i = 1, m
    for j = 1,  t
        C[i][j] = 0
        for k = 1, n
            C[i][j] = C[i][j] + A[i][k]*B[k][j]
```

Figure 1: Basic Linear Algebra Subroutines

2 A Simple Generic Programming System: the Strawman API

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



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

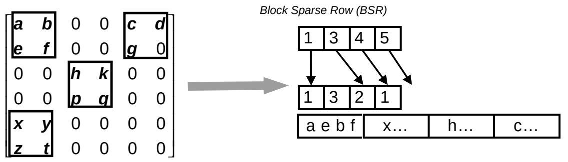
Co-ordinate (COO)

1	1	2	3	2	3	4	3	4	4	`colind`
3	1	3	4	1	2	4	1	2	1	`rowind`
b	a	d	g	c	f	j	e	i	h	`values`

Compressed Sparse Row (CSR)

1	3	5	8	11	`rowptr`

1	3	1	3	1	2	4	1	2	4	`colind`
a	*b*	*c*	*d*	e	*f*	*g*	h	*i*	*j*	`values`



$$\begin{bmatrix} a & b & 0 & 0 & c & d \\ e & f & 0 & 0 & g & 0 \\ 0 & 0 & h & k & 0 & 0 \\ 0 & 0 & p & q & 0 & 0 \\ x & y & 0 & 0 & 0 & 0 \\ z & t & 0 & 0 & 0 & 0 \end{bmatrix}$$

Block Sparse Row (BSR)

1	3	4	5
1	3	2	1
a e b f	x…	h…	c…