$(0, 1)$-Matrix-Vector Products via Compression by Induction of Hierarchical Grammars

Aaron Webb
Computer Science Department
St. Cloud State University
St. Cloud, MN 56301
aaronmwebb@gmail.com

Andrew A. Anda
Computer Science Department
St. Cloud State University
St. Cloud, MN 56301
aanda@stcloudstate.edu

Abstract

We demonstrate a method for reducing the number of arithmetic operations within a $(0, 1)$-matrix vector product. We employ an algorithm, SEQUITUR, developed for lossless text compression, which generates a context free grammar derived from an inherent hierarchy of repeated sequences. In this context, the sequences are composed of bit patterns for a set of adjacent columns. This grammar will represent the original matrix as a hierarchical set of rules identifying and exploiting structural redundancies. It is then sufficient to compute the inner product value of that pattern only once. When that pattern reappears in a different row, that inner product value is reloaded rather than recomputed, thus obviating computations for each repetition.

Aaron Webb and Andrew A. Anda
Computer Science Department
St. Cloud State University
St. Cloud, MN 56301
aaronmwebb@gmail.com
aanda@stcloudstate.edu

1 Introduction

A $(0,1)$-matrix (also identified as zero-one or Boolean) is a rectangular matrix for which each element of the matrix has the value of either one or zero. $(0,1)$-matrices arise from problems in a variety of application areas. See [1] for a brief survey.
In fact, any general matrix may be decomposed into the linear combination of conformal (0,1)-matrices. For a general real matrix $A \in \mathbb{R}^{m\times n}$,

$$A = \sum_{i=1}^{m}\sum_{j=1}^{n} \alpha_{ij} E_{ij}, \left\{ E_{ij} = \mathbf{e}_i \mathbf{e}_j^T, 1 \le i \le m; 1 \le j \le m \right\}, \tag{1}$$

where α_{ij} is an element of A, $\mathbf{e}_i$ is the ith unit vector, and E_{ij} is a resultant $(0,1)$-matrix having only one nonzero element. E.g.,

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} = a \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + b \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + c \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} + d \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}.$$

The matrix-vector product operation,

$$Ax = y, \tag{2}$$

represents the product of a matrix, $A \in \{0,1\}^{m\times n}$, by the vector, $x \in \mathbb{R}^n$, yielding the vector, $y \in \mathbb{R}^m$. More generally, the vectors may actually be over any algebraic ring for which addition and multiplication is closed and well defined.
The general matrix-vector product is an example of *level 2* BLAS (Basic Linear Algebra Subroutines) operation, and as such, it exhibits a quadratic complexity. (i.e., a doubly nested loop is required for its evaluation). There is little that can be done to improve the complexity of the general matrix-vector product (unlike the general matrix-matrix product which can be performed in $O(n^\omega)$, where $2 < \omega < 3$. E.g. Strassen's method, where $\omega = \lg 7 \approx 2.807$). The set of operations may be blocked promoting data locality for modest performance gains for large matrices on a hierarchical memory architecture. But, we know of no way to reduce the number of scalar additions and multiplications. For certain classes of structured matrices, however, we can attain $O(n \ln n)$ by applying the FFT (Fast Fourier Transform) to the calculation. A structured matrix is one which can be fully characterized by $O(n)$ parameters. In fact the product of a rank one matrix and a vector can be performed in $O(n)$ if we know the two vectors that any rank-one matrix can be decomposed into, $Ax = yz^T x$ for some vectors y and z. Nonetheless, we will restrict our consideration to only general $(0,1)$-matrices.

1.1 A Differencing Method

The general Matrix-vector product $Ax = y$, $A \in \mathbb{R}^{m\times n}$, $x \in \mathbb{R}^n$, and $y \in \mathbb{R}^m$, is computed with the following equation,

$$y_i = \sum_{j=1}^{n} \alpha_{ij} x_j, 1 \le i \le m. \tag{3}$$

Computing this requires an algorithm having a doubly nested loop having one of two possible orderings. We'll consider the ordering which performs an inner product in the inner loop (rather than a sum of outer products), so that each outer loop completely calculates one element of the resultant vector y. With either ordering, there are $m(n-1)$ additions and mn multiplications. For a $(0,1)$-matrix though, $\alpha_{ij} \in \{0,1\}, \forall i,j$. So, either the product $\alpha_{ij}x_j$ contributes to the sum as x_j, in the case of $\alpha_{ij} = 1$, or it is skipped, in the case of $\alpha_{ij} = 0$. This implies that the maximum possible amount of work is mn additions which would occur if the matrix were all ones.
Now, lets consider computing the difference between two elements of y, y_i and y_k:

$$\begin{aligned} y_k - y_i &= \sum_{j=1}^{n} \alpha_{kj}x_j - \sum_{j=1}^{n} \alpha_{ij}x_j && (4) \\ &= \sum_{j=1}^{n} (\alpha_{kj}x_j - \alpha_{ij}x_j) && (5) \\ &= \sum_{j=1}^{n} x_j (\alpha_{kj} - \alpha_{ij}) && (6) \end{aligned}$$

Now lets consider computing y_k if y_i has already been computed.

$$\begin{aligned} y_k &= y_i + \sum_{j=1}^{n} x_j (\alpha_{kj} - \alpha_{ij}) && (7) \\ &= y_i + \sum_{j=1}^{n} x_j (d_j), d_j = \alpha_{kj} - \alpha_{ij} && (8) \end{aligned}$$

This implies that apart from the first element of y to be computed, each subsequent element of y can be computed as the sum of a previously computed element with the inner product of x with the difference vector of the two rows of A, d. Let A_k be the kth row of A. Then we have saved $\|A_k\|_1 - \|d\|_1 - 1$ operations in computing y_k. If $\|d\|_1 + 1 < \|A_k\|_1$, the savings is positive. (the offset of 1 is due to the addition of y_i) $\|d\|_1$ is also the *Hamming distance* between two rows of A. E.g. consider a *triangular* $(0,1)$-matrix-vector product,

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} a \\ a+b \\ a+b+c \\ a+b+c+d \end{pmatrix} = \begin{pmatrix} y_0 = a \\ y_1 = y_0 + b \\ y_2 = y_1 + c \\ y_3 = y_2 + d \end{pmatrix}.$$

We can see that in the above example we have halved the number of additions. And, more generally, for a triangular $(0,1)$-matrix-vector product of size n, the number of additions is reduced from $n(n-1)/2$ to $(n-1)$ via the differencing method.
We can deal with any duplicated rows in A by loading its previously computed value at the expense of no additions.
If the remaining rows are unique, what's the lowest number of additions that we need? Because there are no duplicated rows, there must be a Hamming distance of at least one

between each row of A and each other row. The optimum in this case would be for there to be, for every row of A, some other row of A of unit Hamming distance. A sequence of Boolean vectors which satisfies this condition is a Gray code. See [1] for the description of a related algorithm which applies Gray codes to more efficiently perform $(0, 1)$-matrix-vector products.

1.2 SEQUITUR and Hierarchical Structure Identification

In 1994, Nevill-Manning, Witten, and Maulsby [9] introduced an adaptive lossless compression method which develops a model of a given symbol sequence by identifying repeated sequences hierarchically, representing them as a set of rules forming a context free grammar. Nevill-Manning and Witten [4, 6] further developed this method into an algorithm, labeled SEQUITUR, that operates incrementally in linear-time. (For further developments, analyses, and applications of SEQUITUR by its creators, see [2, 5, 3, 7, 8, 10])
The SEQUITUR algorithm is surprisingly simple in its implementation. A hierarchical grammar is created as a linked list of grammar rules and symbols. It can be generated purely by adhering to two constraints while scanning through an input text stream, namely digram uniqueness and rule utility. Digram uniqueness is responsible for the creation of new rules, while rule utility absorbs rules which are unnecessary.
Digram uniqueness ensures that no two character sequence, or digram, may appear more than once in the grammar. This constraint is maintained by means of a hash table containing of all digrams used in the grammar. Should any previously encountered digram be introduced into the grammar, a new rule is generated and all occurrences of the digram will be replaced with a link to the new rule.
Rule utility ensures that every non-root production is used at least twice. This constraint is maintained quite simply through the use of a counter in every rule object. When a rule is used or disused, its counter is incremented or decremented. Should a rule's usage counter be reduced to one, the rule will be expanded within its parent production and be removed.

1.3 Applying SEQUITUR to $(0, 1)$-Matrix Vector Product Computation

In order to use the SEQUITUR algorithm for matrix-vector products, two new constraints must be introduced. Patterns must only be detected within individual columns, and patterns must not be matched across row boundaries. SEQUITUR is designed to work on single line text streams where the pattern does not rely on positional data, and lacks the facilities to find patterns which are position dependent as in this usage. Since the patterns required for the optimization lie in the column layout, the column information must be encoded within the data. This is accomplished simply by concatenating a column number with the boolean matrix element datum before grammar generation. Likewise, SEQUITUR will find patterns that span multiple rows, which are irrelevant for this process. To prevent this, unique sentry characters are injected after each row entry. Being unique, these characters ensure that no patterns will be able to include them. As such, they will only be found within the root-production of the grammar. Since both of these constraints are handled by preprocessing

the matrices before grammar generation, the SEQUITUR algorithm requires no changes in order to produce the necessary grammar.
Once the grammar is produced, the matrix-vector multiplication is accomplished simply by tracing through each of the rules and performing the multiplication on each production segment. Since each rule is tied to a corresponding vector substring, it need only be calculated once. Thus by traversing the grammar tree, entire branches are ”pruned” once their values are calculated elsewhere in the tree. This effect is further enhanced when considering that the hierarchy generation occurs independently of the vector, so that any further multiplications involving the original matrix will be able to profit from this process.
SEQUITUR was shown by its developers to have linear time complexity with respect to the size of the input. For our application, that is $O(mn)$. The dynamic space requirements of SEQUITUR grow linearly as well, which proves problematical for significantly long strings. For this reason, Nevill-Manning and Witten suggest two techniques for augmenting SEQUITUR by artificially restricting and bounding its memory requirements [7].

1.4 C++ code to preprocess a $(0,1)$-matrix for SEQUITUR

The following code snippet demonstrates how we are preprocessing the $(0,1)$-matrix for processing by SEQUITUR. Currently the matrix consists of a stream of integers consisting of either 0 or 1. (A future development will be to accept one or more sparse formats)

```
//Routine to preprocess matrix for grammatical compression
void compressMatrix()
{
  ulong i,j,x,y,n,preval,buffval;
//Retrieve matrix width and height
  cin >> x;
  cin >> y;
  mat.push_back(x);
  mat.push_back(y);
                    // Set unique end of line buffer.  Since we are using
                    // unsigned longs in an increasing direction,
                    // decreasing negative numbers will be fairly safe
  buffval= -1;
                    // Initialize the operations count
  total = 0;
                    // Retrieve the matrix values
  for (i=0;i<y;++i)
  {
    for (j=0;j<x;++j)
    {
      cin >> n;
      mat.push_back(n);

                    // Handle the column pattern constraint.
                    // Here we juxtapose the column position data with
                    // the boolean data.
      n = (j << 1) + n;
```

```
                    // Add this element to the compressed grammar.
                    // (SEQUITUR is called at this point)
      S.last()->insert_after(new symbols(n));

                    // Ensure that the hierarchy only gets verified after
                    // the first entry.
      if (i || j)   S.last()->prev()->check();
    }
                    // Handle the end of row pattern constraint.
                    // Here we insert a unique end of row marker to
                    // ensure patterns cannot span rows.
    S.last()-> insert_after(new symbols(buffval--));
  }
...
```

1.5 Exploiting SEQUITUR and the $(0,1)$-Matrix Vector Product

It is doubtful that preprocessing a $(0,1)$-Matrix into a grammar can ever compete with a non-differencing sparse algorithm, or even a naive systolic differencing algorithm, for a single product with a vector having intrinsic data type elements. However, there are numerous significant applications wherein a given matrix is multiplied by a

set of vectors e.g. a $(0,1)$-Matrix times a general matrix, where each column of the general matrix can be considered as a separate vector; (In the context of parallel computing, this allows for a straightforward partitioning and load-balancing if the complete grammar is broadcast.)

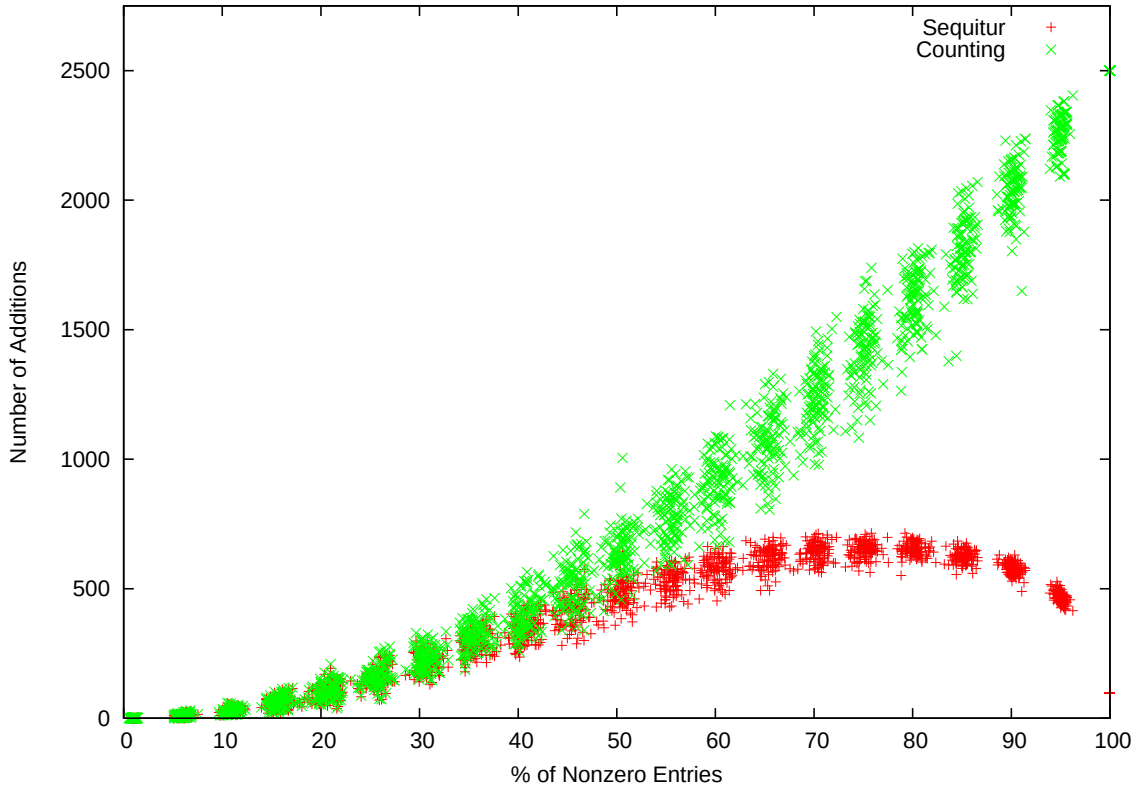
sequence of vectors e.g. iterative methods: $x_p \Leftarrow A^p x_0 \equiv x_p \Leftarrow A x_{p-1}$;

vector with structured elements e.g. very high precision, polynomial, quaternion, matrix, etc. elements, where the cost per operation is relatively high.

In these cases, the expense of the grammar generation can be amortized across the entire set of calculations, and it should prove competitive in many instances.

1.6 Results

Testing the algorithm with large sets of random matrices exibiting a full range a matrix sparse densities, we produced the following results which demonstrate the operational savings of exploiting SEQUITUR for calculating the $(0,1)$-matrix vector product. Each matrix was randomly generated of size 50x50, where the probability of each element being nonzero increased by 5 percent for each 100 element test group. This resulted in 2,100 50x50 matrices being calculated. The resulting comparison of the number of additions utilized versus the sparse density of the matrices is shown below.



1.7 Further Developments

Additionally, with little additional development, SEQUITUR could be applied to a more general matrix for the special case where there is a relatively high multiplicity for some of the elements. This is common for matrices generated by, e.g. graphics and graph analysis applications.

Equation (1) has little practical value unless one considers the special case where the number of distinct α_{ij} terms is smaller than the smallest of the two indices. Then all of the E_{ij} matrices will be added together forming denser $(0, 1)$-matrices for each set of identical α_{ij} terms. We can then refactor equation (1) as:

$$A = \sum_{i=1}^{k} \beta_i B_i, \tag{9}$$

Where there are k distinct entries, $\beta_{1:k}$, in A, and each B_i represents a distinct $(0, 1)$-matrix corresponding to a specific β_i. For a discussion of how to scale and translate these problems to admit efficient solution by means of $(0, 1)$-matrices, see [1].

At our present stage of development, there are some classes of matrices which SEQUITUR does not handle optimally – those which require a systematic subtraction of some elements using the differencing method, e.g, banded matrices.

Additional directions for development:

- Develop an optimization for symmetric matrices.
- Integrate SEQUITUR with a Minimum Spanning Tree algorithm.

- Apply SEQUITUR after preprocessing with a graph partitioning algorithm such as METIS, or other sparse matrix reordering algorithms.
- Investigate the effectiveness of $(0, 1)$-matrices for pre-conditioning iterative methods.

2 Conclusion

We described an effective method for applying SEQUITUR, which generates a context free grammar derived from an inherent hierarchy of repeated sequences, to the problem of reducing the number of operations required by the computation of a $(0, 1)$-matrix vector product. A differencing method was described which could be used in conjunction with SEQUITUR to significantly reduce the number of recomputed sequences thus reducing the overall number of operations. We then described how $(0, 1)$-matrix vector products may be applied to the performance of certain restricted classes of more general matrix vector products.

References

[1] A.A. Anda, *A bound on matrix-vector products for (0,1)-matrices via gray codes*, Proceedings of the 37th Midwest Instruction and Computing Symposium (MICS) (University of Minnesota, Morris), 2004.

[2] C.G. Nevill-Manning, *Inferring sequential structure*, Ph.D. thesis, University of Waikato, Hamilton, New Zealand, 1996.

[3] C.G. Nevill-Manning and I.H. Witten, *Compression and explanation using hierarchical grammars*, Computer Journal **40** (1997), no. 2/3, 103–116.

[4] ———, *Identifying hierarchical structure in sequences: A linear-time algorithm*, Journal of Artificial Intelligence Research **7** (1997), 67–82.

[5] ———, *Inferring lexical and grammatical structure from sequences*, Proc. Compression and Complexity of Sequences 1997 (Positano) (J. A. Storer and M. Cohn, eds.), June 1997.

[6] ———, *Linear-time, incremental hierarchy inference for compression*, Proc. Data Compression Conference (Los Alamitos, CA) (J. A. Storer and M. Cohn, eds.), IEEE Press, 1997, pp. 3–11.

[7] ———, *Phrase hierarchy inference and compression in bounded space*, Proc. Data Compression Conference (Los Alamitos, CA) (J. A. Storer and M. Cohn, eds.), IEEE Press, 1998, pp. 179–188.

[8] ———, *Online and offline heuristics for inferring hierarchies of repetitions in sequences*, Proc. Institute of Electrical and Electronic Engineers, vol. 88, November 2000, pp. 1745–1755.

[9] C.G. Nevill-Manning, I.H. Witten, and D.L. Maulsby, *Compression by induction of hierarchical grammars*, Proc. Data Compression Conference (Los Alamitos, CA) (J. A. Storer and M. Cohn, eds.), IEEE Press, 1994, pp. 244–253.

[10] I.H. Witten, *Adaptive text mining: Inferring structure from sequences*, J. Discrete Algorithms **2** (2004), no. 2, 137–159.