The Mailman algorithm: a note on matrix vector multiplication

Edo Liberty *
Computer Science
Yale University
New Haven, CT

Steven W. Zucker *
Computer Science and Applied Mathematics
Yale University
New Haven, CT

Abstract

Given an $m \times n$ matrix A we are interested in applying it to a real vector $x \in \mathbb{R}^n$ in less than the straightforward $O(mn)$ time. For an exact deterministic computation at the very least all entries in A must be accessed, requiring $O(mn)$ operations and matching the running time of naively applying A to x. However, we claim that if the matrix contains only a constant number of distinct values, then reading the matrix once in $O(mn)$ steps is sufficient to preprocess it such that any subsequent application to vectors requires only $O(mn/\log(max\{m,n\}))$ operations. Algorithms for matrix-vector multiplication over finite fields, which save a log factor, have been known for many years. Our contribution is unique in its simplicity and in the fact that it applies also to real valued vectors. Using our algorithm improves on recent results for dimensionality reduction. It gives the first known random projection process exhibiting asymptotically optimal running time. The mailman algorithm is also shown to be useful (faster than naïve) even for small matrices.

keywords

Algorithms, Matrix vector multiplication, Mailman algorithm, Random Projections.

Introduction

A classical result of Winograd ([14]) shows that general matrix-vector multiplication requires $\Omega(mn)$ operations. This matches the running time of the naïve algorithm. However, since matrix-vector multiplication is such a common, basic operation, an enormous amount of effort has been put into exploiting the special structure of matrices to accelerate it. For example, Fourier, Hadamard, Toeplitz, Vandermonde, wavelet, and others can be applied to vectors in $O(npolylog(n))$ operations.

Others have focussed on matrix-matrix multiplication. For two $n \times n$ binary (over $\mathbb{F}_2$) matrices the historical *Four Russians Algorithm* [12] (modified in [10]) gives a log factor improvement over the naïve algorithm, i.e, running time of $O(n^3/log(n))$. Techniques for saving log factors in real valued matrix-matrix multiplications were also found [9]. These methods are reported to be practically more efficient than naïve implementations. Classic positive results, achieving a polynomial speedup, by Strassen [11] and Coppersmith and Winograd [5] as well as lower bounds [4] for general matrix-matrix multiplications are known. These, however, do not extend to matrix-vector multiplication.

We return to matrix-vector operations. Since every entry of the matrix, A, must be accessed at least once, we consider a preprocessing stage. After the preprocessing stage x is given and we seek an algorithm to produce the product Ax as fast as possible. Within this framework Williams [13] showed that an $n \times n$ binary matrix can be preprocessed in time $O(n^{2+\epsilon})$ and subsequently applied to *binary vectors* in $O(n^2/\epsilon \log^2 n)$. Williams also extends his result to matrix operations over finite semirings. However, to the best of the authors' understanding, his technique cannot be extended to real vectors.

*Research supported by AFOSR and NGA

In this manuscript we claim that any $m \times n$ binary matrix[1] can be preprocessed in time $O(mn)$ such that it can be applied to any *real* vector $x \in \mathbb{R}^n$ in $O(mn/\log(max\{m,n\}))$ operations. Moreover, any matrix over a *finite alphabet* Σ can be preprocessed similarly and subsequently applied in $O(mn \log|\Sigma|/\log(max\{m,n\}))$ operations.[2] Such operations are common, for example, in spectral algorithms for unweighted graphs, computing graph transitive closures, nearest neighbor searches, dimensionality reduction and compressed sensing. Our algorithm also achieves all previous results (excluding that of Williams) while using a strikingly simple approach.

The Mailman algorithm

Intuitively, our algorithm multiplies A by x in a manner that brings to mind the way a mailman distributes letters, first sorting the letters by address and then delivering them. Recalling the identity $Ax = \sum_{i=1}^{n} A^{(i)}x(i)$, metaphorically one can think of each column $A^{(i)}$ as indicating one "address", and each entry $x(i)$ as a letter addressed to it. To continue the metaphor, imagine that computing and adding the term $A^{(i)}x(i)$ to the sum is equivalent to the effort of walking to house $A^{(i)}$ and delivering $x(i)$. From this perspective, the naive algorithm functions by delivering each letter individually to its address, regardless of how far the walk is or if other letters are going to the same address. Actual mailmen, of course, know much better. First, they arrange their letters according to the shortest route (which includes all houses) without moving; then they walk the route, visiting each house regardless of how many letters should be delivered to it (possibly none).

To extend this idea to matrix-vector multiplication, our algorithm decomposes A into two matrices, U and P, such that $A = UP$. The matrix P is the "address-letter" correspondence matrix. Applying P to x is analogous to arranging the letters. U is the matrix containing all possible columns in A. Applying U to (Px) is analogous to walking the route. Hence, we name our algorithm after the age-old wisdom of the men and women of the postal service.

For simplicity, we describe the algorithm for an $m \times n$ matrix A, where $m = \log_2(n)$ and $A(i,j) \in \{0,1\}$. Later we shall generalize it to include other matrix dimensions and possible entry values. There are precisely $2^m = n$ possible columns in the matrix A, by construction, since each of the m column entries can be only 0 or 1. Define the universal columns matrix, U_n, as the matrix containing *each* possible column $\{0,1\}^m$ once. By definition, for any column $A^{(j)}$ there exists exactly one index $1 \le i \le n$ such that $A^{(j)} = U_n^{(i)}$. Define the $n \times n$ correspondence matrix P as $P(i,j) = \delta(U^{(i)}, A^{(j)})$. Here δ stands for the Kronecker delta function. $\delta(U_n^{(i)}, A^{(j)}) = 1$ if $U_n^{(i)} = A^{(j)}$ and zero otherwise.

$$\begin{aligned}(U_nP)(i,j) &= \sum_{k=1}^{n} U_n(i,k)P(k,j) \\ &= \sum_{k=1}^{n} U_n^{(k)}(i)\delta(U_n^{(k)}, A^{(j)}) \\ &= A^{(j)}(i) = A(i,j)\end{aligned}$$

The Mailman algorithm simply uses the associativity of matrix multiplication to compute $Ax = (UP)x = U(Px)$. The fact that computing Px requires $O(n)$ operations is clear since P is a sparse matrix containing only n non-zeros. We later show that applying U_n to any vector also requires $O(n)$ operations. Thus, although A is of size $\log(n) \times n$, the product Ax can be computed via $U(Px)$ while performing only $O(n)$ operations. This gains a $\log(n)$ factor over the naïve application, requiring $O(n\log(n))$ operations. If the number of rows, m, in A is more than $\log(n)$ we partition A into at most $\lceil m/\log(n) \rceil$ submatrices each of size at most $\log(n) \times n$. Since each of the submatrices can be applied in $O(n)$ operations, the entire matrix can be applied in $O(mn/\log(n))$ operations. We now describe how to construct the correspondence matrix P and how to apply the universal column matrix U_n efficiently.

[1] A real valued matrix taking entry values from the set $\{0,1\}$

[2] A matrix A is said to be over a *finite alphabet* if $\forall\, i,j\ A(i,j) \in \Sigma$, and $|\Sigma|$ is a finite constant.

Preprocessing: constructing the correspondence matrix

An entry in the correspondence matrix $P(i,j)$ is set to 1 if $A^{(j)} = U^{(i)}$; otherwise $P(i,j) = 0$. In the next section we construct U such that each column i encodes in binary the value $i-1$. Thus, if a column of $A^{(j)}$ contains the binary representation of the value $i-1$, it is equal to $U^{(i)}$. We therefore construct P by reading A and setting $P(i,j)$ to 1 if $A^{(j)}$ represents the value $i-1$. This concludes the preprocessing stage and can clearly be achieved in $O(mn)$ steps. Notice that since U is fixed, P encodes all the information about A.

Application: universal column matrices

Denote by U_n the $\log_2(n) \times n$ matrix, which contains all strings $\{0,1\}^{\log(n)}$ as columns. Also denote by $\mathbf{0}_n$ and $\mathbf{1}_n$ the all zeros and all ones vectors of length n. Notice immediately that U_n can be constructed recursively:

$$U_1 = \begin{pmatrix} 0 & 1 \end{pmatrix}, \quad U_n = \left(\begin{array}{c|c} \mathbf{0}_{n/2}^T & \mathbf{1}_{n/2}^T \\ \hline U_{n/2} & U_{n/2} \end{array}\right).$$

Applying U_n to any vector z requires less than $4n$ operations, which can be shown by dividing z into its first and second halves, z_1 and z_2, and computing the product $U_n z$ recursively.

$$U_n z = \left(\begin{array}{c|c} \mathbf{0}_{n/2}^T & \mathbf{1}_{n/2}^T \\ \hline U_{n/2} & U_{n/2} \end{array}\right) \left(\begin{array}{c} z_1 \\ \hline z_2 \end{array}\right) = \left(\begin{array}{c} \mathbf{0}_{n/2}^T z_1 + \mathbf{1}_{n/2}^T z_2 \\ \hline U_{n/2}(z_1 + z_2) \end{array}\right)$$

Computing the vector $z_1 + z_2$, and the sums $\mathbf{0}_{n/2}^T z_1$ and $\mathbf{1}_{n/2}^T z_2$ requires (at most) $2n$ operations. Denoting by $T(n)$ the number of operations required for applying U_n to a vector $x \in \mathbb{R}^n$, we get the recurrence relation:

$$T(2) = 2, \qquad T(n) = T(n/2) + 2n \qquad \Rightarrow \qquad T(n) \leq 4n.$$

Computing $\mathbf{0}_{n/2}^T z_1$ can of course be removed from the runtime analysis, but we keep it in anticipation of alphabets other than $\{0,1\}$.

Constant sized alphabets

A matrix A is said to be over an alphabet Σ if $\forall i,j \;\; A(i,j) \in \Sigma$. Such matrices can be decomposed similarly and applied to any vector in $O(mn \log(|\Sigma|)/\log(n))$ time. The definition of U_n is changed such that it encodes all possible strings over $\Sigma = \{\sigma_1, \ldots, \sigma_S\}$, $|\Sigma| = S$.

$$U_1 = (\sigma_1, \ldots, \sigma_S) \qquad U_n = \left(\begin{array}{c|c|c} \sigma_1 \mathbf{1}_{n/S}^T & \cdots & \sigma_S \mathbf{1}_{n/S}^T \\ \hline U_{n/S} & \cdots & U_{n/S} \end{array}\right)$$

The matrix U_n is of size $\log_S(n) \times n$ and it can be applied to $x \in \mathbb{R}^n$ in $O(n)$ time. The construction of P also changes. $P(i,j)$ is set to 1 iff $A^{(j)}$ represents the number $i-1$ in base S under the transformation $\sigma_1 \to 0, \ldots, \sigma_S \to S-1$. If the number of rows, m in A is larger than $\log_S(n)$, we divide A it into $\lceil m/\log_S(n) \rceil$ sections of size at most $\log_S(n) \times n$. The total running time therefore becomes $O(mn \log(S)/\log(n))$.

Saving a $\log(m)$ factor

If the number of rows, m, in A is larger than the number of columns, n, we can achieve an application running time of $O(mn/\log(m))$. Assume that A is of size $m \times n$ where $n = \log(m)$. We have that $A = P^T U_m^T$, where P is the correspondence matrix for A^T as explained above. It is left to show that computing $U^T x$ for $x \in \mathbb{R}^{\log(m)}$ requires $O(m)$ operations. Denote by x_1 the first element of x and by $x_{2:n}$ the remaining $n-1$ elements.

$$U_m^T x = \left(\begin{array}{c|c} \mathbf{0}_{m/2} & U_{m/2}^T \\ \hline \mathbf{1}_{m/2} & U_{m/2}^T \end{array}\right) \left(\begin{array}{c} x_1 \\ \hline x_{2:n} \end{array}\right) = \left(\begin{array}{c} \mathbf{0}_{m/2} x_1 + U_{m/2}^T x_{2:n} \\ \mathbf{1}_{m/2} x_1 + U_{m/2}^T x_{2:n} \end{array}\right)$$

Clearly, computing $U_m^T x$ requires computing $U_{m/2}^T x_{2:n}$ and an additional $O(m)$ operations which gives a total running time of $O(m)$. If the number of columns n of A is more than $\log(m)$, we partition A vertically to at most $\lceil n/\log m \rceil$ submatrices of size $m \times \log m$. Since each submatrix is applicable in $O(m)$ operations, the total running time of applying A to a vector $x \in \mathbb{R}^n$ is $O(mn/\log(m))$.

Remark 1 (Matrix operations over semirings) *Suppose we equip Σ with an associative and commutative addition operation $(+)$, and a multiplication operation $(\cdot)$ that distributes over $(+)$. If both A and x are chosen from Σ the matrix vector multiplication over the semiring $\{\Sigma, +, \cdot\}$ can be performed using the exact same algorithm. This is, of course, not a new result. However, it includes all other* log*-factor-saving results.*

Dimensionality reduction

Assume we are given p points $\{x_1, \ldots, x_p\}$ in $\mathbb{R}^n$ and are asked to embed them into $\mathbb{R}^m$ such that all distances between points are preserved up to distortion ε and $m \ll n$. A classic result by Johnson and Lindenstrauss [7] shows that this is possible for $m = \Theta(\log(p)/\varepsilon^2)$. The algorithm first chooses a random $m \times n$ matrix, A, from a special distribution. Then, each of the vectors (points) $\{x_1, \ldots, x_p\}$ is multiplied by A. The embedding $x_i \to Ax_i$ can be shown to have the desired property with at least constant probability.

Clearly a naive application of A to each vector requires $O(mn)$ operations. A recent result by Ailon and Liberty [3], improving on Ailon and Chazelle [2], gives a distribution over matrices A which can be applied in $O(n\log(m))$ operations. We now claim that, in some situations, an older result by Achlioptas can be (trivially) modified to outperform this last result. In particular, Achlioptas [1] showed that the entries of A can be chosen *i.i.d.* from $\{-1, 1\}$ with probability $1/2$ each. Using our method and Achlioptas's distribution, one can apply the chosen A to each vector in $O(n\log(p)/\log(n)\varepsilon^2)$ operations (recall that $m = O(\log(p)/\varepsilon^2)$). Therefore, if the number of points, p, is at most a constant polynomial in their dimension, n, $\log(p) = O(\log(n))$ and applying A to x_i requires only $O(n/\varepsilon^2)$ operations. For a constant ε this yields an $O(n)$ algorithm.[3] It is important to mention that the above construction is the first to match the running time lower bound for general dimensionality reduction.

Experiments

Here we compare the running time of applying a $log(n) \times n$, $\{0, 1\}$ matrix A to a vector of floating point variables $x \in \mathbb{R}^n$ using three methods. The first is a *naïve* implementation in C: this simply consists of two nested loops ordered with respect to memory allocation to minimize cache faults. The second is an *optimized* matrix vector code: we test ourselves against LAPACK which uses BLAS subroutines [8, 6]. The third is, of course, the *mailman* algorithm following the preprocessing stage.

Although the complexity of the first two methods is $O(n\log(n))$ and that of the mailman algorithm is $O(n)$, we do not seem to get a $\log(n)$ speedup. This is because the memory access pattern in applying P is problematic at best. The reader should bare in mind that these results might depend on memory speed and size vs. CPU speed of the specific machine. The experiment below denotes the time required for the actual multiplication not including memory allocation. (Experiments were conducted on an Intel Pentium M 1.4Ghz processor, 598Mhz Bus speed and 512Mb of RAM.)

As seen in figure 1, the mailman algorithm operates faster than the naïve algorithm even for 4×16 matrices. It also outperforms the LAPACK procedure for most matrix sizes. The machine specific optimized code (LAPACK) is superior when the matrix row allocation size approaches the memory page size. A machine specific optimized mailman algorithm might take advantage of the same phenomenon and outperform the LAPACK on those values as well.

[3] Notice that the dependance on ε is $1/\varepsilon^2$ instead of $\log(1/\varepsilon)$ which makes this result actually much slower for most practical purposes.

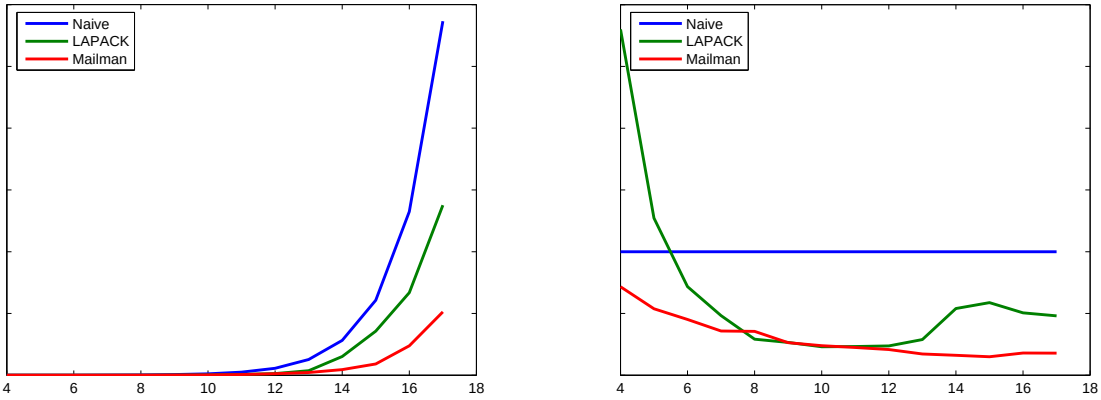

Figure 1: Running time for multiplying an $m \times 2^m$ ± 1 matrix to a double precision vector. m is given on the x-axis. The y-axis gives the running time for achieving the multiplication using three different algorithms. (1) Naive: matrix multiplication implemented as two nested for-loops written in C and ordered correctly with respect to memory access. (2) LAPACK: a general purpose matrix multiplication implementation which is machine optimized. (3) Mailman: the mailman algorithm as described above. The left figure is a plot of the 3 running times on an absolute scale, whereas the right one plots them relatively to the naive algorithm.

Concluding remark

It has been known for a long time that a log factor can be saved in matrix-vector multiplication when the matrix and the vector are over constant size alphabets. In this paper we described an algorithm that achieves this, while also dealing with real-valued vectors. As such, the idea by itself is neither revolutionary nor complicated, but it is useful in current contexts. We showed, as a simple application of it, that random projections can be achieved asymptotically faster than the best currently known algorithm, provided the number of projected points is polynomial in their original dimension. Moreover, we saw that our algorithm is advantageous in practice even for small matrices.

References

[1] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.

[2] Nir Ailon and Bernard Chazelle. Approximate nearest neighbors and the fast Johnson-Lindenstrauss transform. In *Proceedings of the 38st Annual Symposium on the Theory of Compututing (STOC)*, pages 557–563, Seattle, WA, 2006.

[3] Nir Ailon and Edo Liberty. Fast dimension reduction using rademacher series on dual bch codes. In *Symposium on Discrete Algorithms (SODA), accepted*, 2008.

[4] Roger W. Brockett and David P. Dobkin. On the number of multiplications required for matrix multiplication. *SIAM J. Comput.*, 5(4):624–628, 1976.

[5] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM.

[6] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17, 1988.

[7] William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemp. Math.*, 26:189–206, 1984.

[8] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.

[9] Nicola Santoro. Extending the four russians' bound to general matrix multiplication. *Inf. Process. Lett.*, 10(2):87–88, 1980.

[10] Nicola Santoro and Jorge Urrutia. An improved algorithm for boolean matrix multiplication. *Computing*, 36(4):375–382, 1986.

[11] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, (4):354–356, 08 1969.

[12] M.A. Kronrod V.L. Arlazarov, E.A. Dinic and I.A. Faradzev. On economic construction of the transitive closure of a direct graph. *Soviet Mathematics, Doklady*, (11):1209–1210, 1970.

[13] Ryan Williams. Matrix-vector multiplication in sub-quadratic time: (some preprocessing required). In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 995–1001, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[14] Shmuel Winograd. On the number of multiplications necessary to compute certain functions. *Communications on Pure and Applied Mathematics*, (2):165–179, 1970.