CONF-900992--2

DE90 011108

An Adaptive Blocking Strategy for Matrix Factorizations*

Christian H. Bischof

Philippe G. Lacroute

Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439–4801

Keywords: block algorithm, adaptive blocking, performance evaluation, performance portability, QR factorization.

Abstract. On most high-performance architectures, data movement is slow compared to floating-point (in particular, vector) performance. On these architectures block algorithms have been successful for matrix computations. By considering a matrix as a collection of submatrices (the so-called blocks) one naturally arrives at algorithms that require little data movement. The optimal blocking strategy, however, depends on the computing environment and on the problem parameters. Current approaches use fixed-width blocking strategies which are not optimal. This paper presents an "adaptive blocking" methodology for determining in a systematic manner an optimal blocking strategy for a uniprocessor machine. We demonstrate this technique on a block QR factorization routine on a uniprocessor. After generating timing models for the high-level kernels of the algorithm we can formulate the optimal blocking strategy in a recurrence relation that we can solve inexpensively with a dynamic programming technique. Experiments on one processor of a CRAY 2 show that in fact the resulting blocking strategy is as good as any fixed-width blocking strategy. So while we do not know the optimum fixed-width blocking strategy unless we re-run the same problem several times, adaptive blocking provides optimum performance in the very first run.

1 Introduction

On most high-performance architectures data movement is quite slow compared to floating-point (in particular, vector) performance. A memory hierarchy has been the traditional approach to overcome this problem [17, ch. 2.1]. For a single processor the memory hierarchy may be composed of vector registers, cache memory, main memory, and solid-state disks. On hierarchical and shared memory systems local memory, cluster memory, and global memory are further extensions, whereas on distributed-memory architectures we have the distinction between local and remote memory. Memory hierarchies can successfully overcome memory latency and bandwidth limitations if the algorithms exhibit a high degree of data reuse, that is, data at the low (and quick-to-access) end of the memory hierarchy is re-used often before new data is needed.

In this context block algorithms have been very successful for matrix computations [2,7,5,11,13, 14,21]. By considering the matrix at the highest level of the algorithm as a collection of submatrices

*This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

(the so-called "blocks") one can express the algorithm in terms of operations such as matrix-matrix multiplication that inherently allow for a great amount of data reuse.

The crucial decision with respect to the performance of block algorithms is to choose an optimal blocking strategy, i.e., partition the matrix in such a fashion that the running time of the program is minimized. This is of particular disadvantage for library routines that should be designed to perform close to optimal on *any* problem size. The issue of optimal blocking is not a simple one. The optimum strategy depends on the characteristics of the architecture: Generation of a block transformation usually becomes more expensive as the block size increases, whereas application of a block transformation performs faster with increasing block size. The cross-over point is machine-specific. The currently most common approach to block determination is to choose fixed-width blocks, based on some experimental results on a given machine. The optimal blocking strategy, however, depends also on the problem size. For a small problem the overhead associated with blocking might actually increase the running time, whereas for large problems big blocks will utilize available floating-point hardware more effectively.

Assuming a machine- and problem-dependent blocking strategy, one then has several possibilities to implement blocking in a program: the use of a fixed block size, the use of a varying block size determined by some a-priori strategy, and the use of a varying block size determined dynamically in the course of the execution of the factorization. In [4], Bischof suggested a methodology called *adaptive blocking* for finding 'good' block sizes for pipelined factorization algorithms on distributed memory multiprocessors. This technique tries to formalize the performance tradeoff resulting from different blocking strategies. System parameters are taken into account by generating timing models based on observed data, not simplistic measures such as floating point operations or memory access counts.

This paper extends this work in several ways. By limiting ourselves to a uniprocessor, we can formulate a recursion formula for the truly optimal blocking strategy based on timing models. We show how this recursion formula can be solved inexpensively using a dynamic programming technique. We also describe our first attempts at generating timing models in a systematic fashion. As in [4], we choose the QR factorization of a dense matrix to demonstrate our methodology. The paper is organized as follows: §2 discusses block algorithms in more detail and develops a block QR factorization algorithm as an example. In §3 we develop the recurrence relation describing the choice of blocksizes minimizing run-time and devise a dynamic programming technique to that yields the optimum block sizes at little cost. In §4 we discuss how to create the required timing information for the dynamic programming algorithm. Our method consists of collecting a small sample of timing data and then using data-fitting techniques to produce the remaining data as necessary. We then present the results of some experiments with our method on one processor of a Cray 2 in §5. Finally we summarize our results and discuss some issues that have to be addressed in order to make adaptive blocking algorithms fully portable.

2 A Block Algorithm for Computing the QR Factorization

We present our adaptive blocking methodology in the context of an algorithm for computing the QR decomposition

$$A = QR \tag{1}$$

of a dense matrix on a uniprocessor. Here an $m \times n$ matrix A is decomposed into an orthogonal $m \times m$ matrix Q and an upper triangular $m \times n$ matrix R. This decomposition is one of the basic tools of numerical linear algebra; for applications, see [15].

The traditional algorithm for computing the QR factorization [15, p.148] employs a sequence of Householder reductions

$$H = H(u) = I - 2uu^T, \ ||u||_2 = 1. \tag{2}$$

Application of H to a given matrix A involves a matrix-vector multiplication $z \leftarrow A^T u$ and a rank-one update $A \leftarrow A - 2uz^T$. Each of these operations requires $O(n^2)$ floating-point operations and uses $O(n^2)$ data. In contrast, an operation like matrix-matrix multiplication uses $O(n^2)$ data for $O(n^3)$ floating-point operations, and so the ratio of data movement to arithmetic operations is $O(1/n)$. This *surface-to-volume ratio* [13] prevents degradation of performance from memory accessing delays.

To arrive at a block formulation of the Householder QR algorithm, we must be able to express a series of Householder reductions in a convenient closed form. Schreiber and Van Loan [22] expressed the product

$$Q = H_1 \cdots H_p$$

of a series of $m \times m$ Householder matrices (2) in the so-called *compact WY representation*

$$Q = I + YTY^T \tag{3}$$

where W and Y are $m \times p$ matrices and T is a $p \times p$ upper triangular matrix. The accumulation of Y and T is described in detail in [22], but here it suffices to say that compared to the traditional Householder algorithm, the accumulation of T requires $O(mp^2)$ extra flops and $\frac{p^2}{2}$ extra words for storage. Since typically $m \gg p$ this is a low-order term in the overall algorithmic complexity. The advantage of the compact WY representation is that the computation of $A \leftarrow Q^T A$ now involves two matrix-matrix multiplications $Z \leftarrow A^T YT$ and a rank-p update $A \leftarrow A + YZ^T$ instead of a series of p matrix-vector multiplications and rank-one updates.

We can now express the block Householder QR algorithm in terms of the primitives GenHH (compute QR factorization of a submatrix and generate a set of Householder vectors), AccWY (accumulate triangular factor of a block transformation), and AppCWY (apply compact WY factor):

$$[Y, R] \leftarrow \text{GenHH}(B)$$

applied to a m by p matrix computes p Householder vectors u_i such that $H(u_1) \cdots H(u_p) \cdot R = B$ and $Y(:, i) = u_i$.

$$T \leftarrow \text{GenCWY}(Y)$$

returns the compact WY factors T such that $H(u_1) \cdots H(u_p) = I + YTY^T$. as described in [22].

$$A \leftarrow \text{AppCWY}(Y, T, A)$$

performs the updates $A \leftarrow Q^T A$ and $Z \leftarrow A^T YT$.

Figure 1 shows the block Householder algorithm using the compact WY representation. Here A is partitioned into block columns of width p and for simplicity we assume that all block columns are

for $k \leftarrow 1$ **to** $\lceil n/b \rceil$ **do**
  $i \leftarrow (k-1)b+1$
  $j \leftarrow \min(i+b-1, n)$
  $[Y, A(i:m, i:j)] \leftarrow \text{GenHH}(A(i:m, i:j))$
  $T \leftarrow \text{AccCWY}(Y)$
  $A(i:m, j+1:n) \leftarrow \text{AppCWY}(Y, T, A(i:m, i:n))$
end for

Figure 1: Fixed-Block Householder QR Algorithm

of the same size, so $n = pN$. We use the notation $A(i,j)$ to refer to entry (i,j) and $A(i:j, k:l)$ to refer to the submatrix of A consisting of row entries i to j and column entries k to l. Bischof and Van Loan [7], Harrod [16] and Mayes [20] used the WY representation to compute the QR factorization on the FPS-164/MAX, the Alliant FX/8 and the IBM 3090, respectively.

Finally, we generalize the fixed-block algorithm to allow variable-size blocks. The principle of a block algorithm is to divide the problem into subproblems which are just large enough to fit in the fastest memory level. However, in the fixed-block algorithm the size of the subproblems decreases as the computation progresses because the number of rows and columns left to be processed decreases at each step. As a result, we stand a better chance of achieving the minimum execution time if we allow for variable block sizes.

Assume we know the block size $B(k)$ for the k^{th} step of the algorithm, and *steps* is the total number of steps, where one 'step' refers to a single pass through the loop. Then Figure 2 shows the resulting algorithm which we call the variable-block QR factorization algorithm.

$i \leftarrow 1$
for $k \leftarrow 1$ **to** *steps* **do**
  $j \leftarrow i + B(k) - 1$
  $[Y, A(i:m, i:j)] \leftarrow \text{GenHH}(A(i:m, i:j))$
  $T \leftarrow \text{AccCWY}(Y)$
  $A(i:m, j+1:n) \leftarrow \text{AppCWY}(Y, T, A(i:m, i:n))$
  $i \leftarrow i + B(k)$
end for

Figure 2: Block QR Algorithm with Variable Blocking

3 Determining an Optimal Blocking Sequence

The performance of the variable-block QR algorithm depends strongly on how the matrix is partitioned into blocks. Our goal is to choose the partition which minimizes the execution time of the factorization. In order to make such a choice we must have a method to estimate how long the

QR factorization will take for a given sequence of block sizes. Suppose we have available a function Time(m, n, p) which returns the time required to perform one step of the QR algorithm with a block size of p on a matrix of size m by n. That is, Time(m, n, p) is the time

1. to compute the unblocked QR factorization of an m by p matrix
2. to accumulate the compact WY factor determined by the p Householder vectors
3. to apply the m by p block Householder factor to an $(m-p)$ by $(n-p)$ matrix

The execution time t_{total} of an entire factorization is found by adding the time required for each of the steps:

$$t_{total}(m, n, B) = \sum_{i=1}^{steps} \text{Time}(m_i, n_i, B(i)) \tag{4}$$

where $B(i)$ is the block size for the i^{th} step, *steps* is the total number of steps, and the size of the submatrix remaining at the i^{th} step is m_i by n_i. The submatrix dimensions are computed as follows:

$$m_i = m - \sum_{k=1}^{i-1} B(k) \text{ and } n_i = n - \sum_{k=1}^{i-1} B(k).$$

The obvious way to find the optimal sequence of block sizes for a matrix is to enumerate all possible sequences. However, the number of block size sequences is exponential in the number of columns in the matrix so it is impractical to enumerate them all. Instead, we can formulate the problem in terms of a recurrence relation which can be solved in polynomial time. Equation (4) can be rewritten as follows:

$$\begin{aligned} t_{total}(m, 0, B) &= 0 \\ t_{total}(m, n, B) &= \text{Time}(m, n, \text{H}(B)) + t_{total}(m - \text{H}(B), n - \text{H}(B), \text{T}(B)). \end{aligned} \tag{5}$$

H(B) returns the first block size in B (the "head" of the list), and T(B) returns the remaining portion of B after the first block size has been removed (the "tail" of the list). The second equation states that the time to factor an $m \times n$ matrix using a given sequence of block sizes is equal to the time to process the first block size followed by the time to factor the remaining portion of the matrix with the remaining block sizes.

Our goal is to minimize t_{total} by choosing the elements of B appropriately. This can be done as follows:

$$\begin{aligned} t_{best}(m, n) &= \min_B \{ t_{total}(m, n, B) \} \\ &= \min_B \{ \text{Time}(m, n, \text{H}(B)) + t_{total}(m - \text{H}(B), n - \text{H}(B), \text{T}(B)) \} \\ &= \min_B \{ \text{Time}(m, n, \text{H}(B)) + t_{best}(m - \text{H}(B), n - \text{H}(B)) \} \\ &= \min_p \{ \text{Time}(m, n, p) + t_{best}(m - p, n - p) \} \end{aligned} \tag{6}$$

where $p = H(B)$. This recurrence shows that if we know the minimum execution times for submatrices of size $m - p$ by $n - p$ with $1 \leq p < n$, then we can find the best execution time for the

entire matrix by solving a minimization problem in the one integer variable p. Since block sizes are bounded by the size of the matrix, the minimization problem can be solved in linear time simply by trying all possibilities for p.

```
BestCost(0) ← 0
for n' ← 1 to n do
   m' ← m − n + n'
   BestCost(n') ← ∞
   for p ← 1 to min(maxbsize, n) do
      cost = Time(m', n', p) + BestCost(n' − p)
      if cost < BestCost(n')
         BestCost(n') ← cost
         BestSize(n') ← p
      end if
   end for
end for

BlockSequence ← []
n' ← n
while n' > 0 do
    BlockSequence ← [BlockSequence, BestSize(n')]
    n' ← n' − BestSize(n')
end while
```

Figure 3: Determining the Sequence of Optimal Block Sizes

This situation is ideal for an algorithm based on dynamic programming [1]. The idea is to start by finding the optimal partitions for small matrices and to use the results computed so far to find the solutions to successively larger problems. The partial results are stored in a table so they can be re-used as needed. When the i^{th} step of the algorithm begins, we have available $t_{best}(m_j, n_j)$ for each of the $i-1$ submatrices A_1 to A_{i-1} defined by $A_j = A(n-j+1, m : n-j+1, n)$. To find the best block sequence for A_i we examine each possible size for the first block of the sequence and determine which one satisfies (6). Once the first block has been chosen, the best partition of the remaining columns of the matrix then can be found directly from the previously computed portion of the table. Thus at each step i there are only i possibilities to be considered, and there are a total of n steps, so the algorithm requires $O(n^2)$ time.

The complete algorithm is given in Figure 3. BestSize and BestCost are tables which contain the optimal block sizes and the corresponding execution times, respectively. BestSize(i) contains the first block size in the optimal sequence for the submatrix A_i. The second element in the sequence is therefore BestSize(n − BestSize(i)). The entire sequence can be found by extracting entries from the table in an analogous manner. The nested for-loops fill the two tables, and then the while-loop forms the list of optimal block sizes and assigns it to the variable BlockSequence. *Maxbsize* is the

largest block size the algorithm will consider.

In the for-loops, the size of the submatrix under consideration is specified by the variables m' and n' which each increase by one with each iteration of the outer loop. The possibilities for the block size p are enumerated in the inner loop. The cost associated with each block size is determined in the inner loop exactly as specified by (6).

If *maxbsize* is greater than n then the algorithm considers all possible sequences of block sizes and the first loop requires $O(n^2)$ execution time. However, extremely large block sizes are unlikely to be beneficial because very large subproblems cannot possibly fit into the fastest memory level. As a result, *maxbsize* can be set to a relatively small constant value with the result that the algorithm requires only $O(n)$ time. In both cases the tables occupy only $O(n)$ storage.

Note that the algorithm we have presented can be used to find the optimal block sequence for any blocked algorithm which has the same top-level control structure as the variable-block QR algorithm. For instance, we need only substitute a different set of timing estimates to determine the best partition for a variable-block LU or Cholesky factorization.

4 Timing Models

The algorithm presented in the previous section requires estimated execution times of the inner loop of the factorization algorithm. In the case of our implementation of the QR factorization, the inner loop consists primarily of a small number of calls to numerical kernels from LAPACK [3, 6, 9], a portable library of linear algebra routines for high-performance computing environments. This package relies on the Basic Linear Algebra Subprograms (BLAS) [12,19] implementing matrix-matrix, matrix-vector and vector-vector operations.

More specifically, the block QR code relies on the routines SGEQR2 (unblocked QR factorization), SLARFT (accumulation of the triangular factor in the compact WY representation), and SLARFB (application of a an orthogonal matrix in compact WY representation to another matrix). Instead of generating timing estimates for one block step of the QR factorization, we produce a timing model for each of the numerical kernels SGEQR2, SLARFT, and SLARFB. Then the required timings of the inner loop of the algorithm in Figure 2 can be found by summing values from the models for the three kernels. This method is advantageous because timings can be estimated without any additional work for any other subroutine written using these kernels.

In the previous section we assumed that the execution time of each step of the factorization depends only on the block size and the size of the submatrix left at that step. Since we are considering factorizations of dense matrices, the values in the matrix is not important. However, there are many variables introduced by the hardware architecture, the operating system and the compiler which affect the performance of a given code. These variables are extremely difficult, if not impossible, to quantify. As a result, we chose to predict execution times by making experimental measurements and fitting a model to those measurements. Note that the model will capture the variations in execution speed for different problem sizes, but has to be changed to account for differences in architecture and systems software.

It would be impractical to measure the execution time of a kernel for every matrix and block size, so we generate a timing model by fitting a function to a limited set of measurements. We implemented a three-step process: First, a benchmark program timed the numerical kernels for a fixed set of matrix and block sizes and stored all of the results. Second, we used a multivariate

surface-fitting program to compute a function which "fits" the observed data. This function is stored and the observed timings are discarded. Finally, during execution of the algorithm in Figure 3 the necessary points on the surface are evaluated. The first two steps need to be performed only once, perhaps at the time the library of kernels is installed on a new machine. Only the final step must be computationally efficient since the algorithm to find the best partition must not require more time than the performance improvement it achieves.

There are many available methods for fitting surfaces to a set of observations (see for example [18]). For our purposes we need a method which allows at least three independent variables, and preferably more so that we can model subroutines with an arbitrary number of integer parameters. Next, we must choose between a method which interpolates the observed data and one which smoothes the observations. Timing measurements are subject to small random errors due to the granularity of the system clock and random caching and paging delays. In our view an appropriate smoothing algorithm is advantageous to average out these random errors, while interpolation algorithms would be more appropriate for applications with perfectly-accurate data. A third distinction which affects our choice is that between global and local methods. Loosely speaking, the value of a point on a surface computed by a local method depends only on observations "near" that point, while for a global method each computed point depends on every observation. Local methods seem to be more attractive, however, both because they are generally more computationally efficient and because we expect the observed timings for matrices of very different sizes to be independent.

For our initial experiments we used the LOESS moving least squares smoothing package described in [8] and available from *netlib* [10]. LOESS is a local method which works on data with an arbitrary number of independent variables. The basic idea behind the method is to determine the value of the surface at a given point by computing a weighted least squares fit to the observations near that point. The closest observations receive the greatest weight while distant observations are completely ignored. We also scaled our raw data before fitting a surface to it in order to reduce the variation in the data. Surface-fitting algorithms work best on data which deviate only slightly from a constant value. We used the following scaling transformation:

$$t_i' = \log(\frac{t_i}{m_i n_i p_i}).$$

The product $m_i n_i p_i$ is an easily-computable approximation to the number of floating-point operations required for the routines being timed. We expect the time to be very roughly proportional to the number of floating-point operations required, so this scaling function tends to produce a rather 'flat' set of data. The scaling function can be inverted trivially when data values are retrieved from the surface.

5 Experimental Results

We tested our method on a single processor of a Cray 2. The Cray 2 has a memory hierarchy including a vector register file, a fast local memory and a slower main memory, so blocked algorithms are ideal for the machine.

We collected approximately 6000 timings each for the LAPACK routines SGEQR2, SLARFT and SLARFB used by the variable-block QR factorization subroutine. For each timing we used a different combination of matrix and block sizes. The matrices were all square and contained from 1

to 600 columns, and the block sizes ranged from 2 to 50. Next, we used the LOESS surface-fitting routines to compute a timing estimate function. We chose the size of the neighborhood of points used in the local least squares fits by trial and error, but we intend to automate this process in the future. We obtained the most accurate estimates when the number of points used equaled one to two percent of the total number of measurements. Finally, using the timing estimate function and the algorithm in Figure 3 we determined optimal block size sequences for a number of matrix sizes.

For comparison, we measured execution times for the fixed-block QR algorithm with block sizes from 1 to 32. Figure 4 shows the results for a 500 × 500 matrix. Note that the time we show for

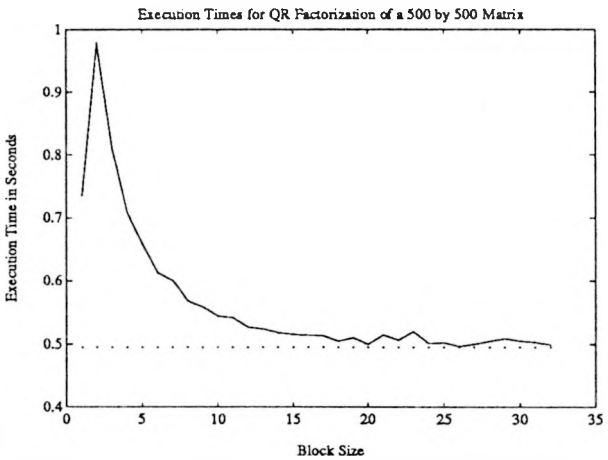

Figure 4: Execution Times for Blocked QR Factorization on a Cray 2

block size 1 actually corresponds to the unblocked algorithm and not the blocked algorithm. The initial jump in time from the unblocked algorithm to the blocked algorithm with block size 2 is due to the additional overhead of the blocked algorithm which is not made up for when the block size is so small. However, for larger block sizes the blocked code takes significantly less time than the unblocked code. The optimal fixed block size is 26, and the corresponding computation takes 0.496 seconds.

Our algorithm returned the following block sizes for a 500 × 500 matrix:

28, 36, 33, 34, 34, 24, 27, 24, 20, 28, 22, 22, 22, 22, 21, 20, 20, 17, 19, 26, 1

The measured execution time for this sequence is 0.495 seconds which is 0.2% lower than the execution time for the best fixed block size. This value is shown by the horizontal dotted line in Figure 4. While the performance gain of the variable-block algorithm over the fixed-block algorithm is negligible, we have achieved this performance without having to compute the QR factorization multiple times with different block sizes. Similar behavior was observed for other block sizes.

6 Conclusions

Block algorithms are a well-known technique for arriving at matrix algorithms that exhibit low data movement overhead. As a result these algorithms achieve high floating-point performance on systems employing any kind of memory hierarchy. An optimal blocking strategy, however, depends both on the computing environment and the parameters of the problem at hand.

We explored these issues in the context of a block QR factorization algorithm for vector uniprocessor. By capturing the system behavior in timing models for the high-level kernels of the algorithm, we were able to devise a strategy that described how block sizes should be chosen to achieve optimum performance. Using a dynamic programming technique, we could implement this strategy inexpensively. The resulting adaptive blocking strategy performed as well as any fixed-width blocking strategy, independent of the problem dimensions. Adaptive blocking solved a given problem in an optimal fashion *in the very first run*, which is exactly what most users expect. This makes adaptive blocking strategies very suitable for library routines which in the first run are supposed to perform optimally over a wide range of problems.

We readily admit, however, that much work remains to be done until performance portability can be achieved in an automatic fashion. Generating timing models is not an easy problem, and there is not very much software available for smoothing or interpolating multivariate surfaces. And as we found out with the LOESS smoothing package, even with the right software there usually are parameters controlling its behavior that have to be chosen carefully. Once timing models have been generated, however, they convey a great deal of information about the behavior of a given machine on a particular kernel routine. We certainly gained a lot of insight by looking at renderings of these surfaces on a graphics workstation. We expect this technique to have applications in performance prediction and benchmarking as well.

References

[1] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974.

[2] Michael Berry, Kyle Gallivan, William Harrod, William Jalby, Sy-Shin Lo, Ulrike Meier, Bernard Philippe, and Ahmed Sameh. Parallel algorithms on the Cedar system. In W. Händler, editor, *Proceedings of CONPAR 86*, pages 25–39. Springer Verlag, New York, 1986.

[3] Christian Bischof, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. LAPACK Working Note #5: Provisional contents. Technical Report ANL–88–38, Argonne National Laboratory, Mathematics and Computer Sciences Division, September 1988.

[4] Christian H. Bischof. Adaptive blocking in the QR factorization. *The Journal of Supercomputing*, 3(3):193–208, 1989.

[5] Christian H. Bischof. Computing the singular value decomposition on a distributed system of vector processors. *Parallel Computing*, 11:171–186, 1989.

[6] Christian H. Bischof and Jack J. Dongarra. *Parallel and Vector Supercomputing: Methods and Algorithms*, chapter A Project for Developing a Linear Algebra Library for High-Performance Computers, pages 45–56. John Wiley & Sons, Somerset, NJ, 1989.

[7] Christian H. Bischof and Charles F. Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8:s2–s13, 1987.

[8] William S. Cleveland, Susan J. Devlin, and Eric Grosse. Regression by local fitting: Methods, properties and computational algorithms. *Journal of Econometrics*, 37:87–114, 1988.

[9] Jim Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, and Danny Sorensen. Prospectus for the development of a linear algebra library for high-performance computers. Technical Report ANL–MCS–TM97, Argonne National Laboratory, Mathematics and Computer Sciences Division, September 1987.

[10] Jack Dongarra and Eric Grosse. Distribution of mathematical software by electronic mail. *Communications of the ACM*, 30(5):403–407, 1987.

[11] Jack Dongarra, Ahmed Sameh, and Danny Sorensen. Implementation of some concurrent algorithms for matrix factorization. *Parallel Computing*, 3(1):25–34, 1986.

[12] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.

[13] Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen. Block reduction of matrices to condensed form for eigenvalue computations. Technical Report ANL–MCS–TM–99, Argonne National Laboratory, Mathematics and Computer Sciences Division, September 1987.

[14] Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. *SIAM Journal on Scientific and Statistical Computing*, 8(6):1079–1084, November 1987.

[15] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.

[16] William Harrod. Solving linear least squares problems on an Alliant FX/8. Technical report, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1986.

[17] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, New York, 1984.

[18] Peter Lancaster and Kęstutis Šalkauskas. *Curve and Surface Fitting: An Introduction*. Academic Press, San Diego, 1986.

[19] C. L. Lawson, R. J. Hanson, R. J. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, September 1979.

[20] Peter Mayes and Guiseppe Radicati di Brozolo. Block factorization algorithms on the IBM 3090/VF. In *Proceedings of the International Meeting on Supercomputing*, 1989.

[21] Robert Schreiber. *Block Algorithms for Parallel Machines*, pages 197–207. Number 13 in IMA Volumes in Mathematics and its Applications. Springer Verlag, Berlin, 1988.

[22] Robert Schreiber and Charles Van Loan. A storage efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.