Chapter 1
A Compiler-Blockable Algorithm for QR Decomposition[*]

Steve Carr[†] R. B. Lehoucq[‡]

Abstract

Because of an imbalance between computation and memory speed in modern processors, programmers are explicitly restructuring codes to perform well on particular memory systems, leading to machine-specific programs. This paper describes a block algorithm for QR decomposition that is derivable by the compiler and has good performance on small matrices — sizes that are typically run on nodes of a massively parallel system or workstation. The advantage of our algorithm over the one found in LAPACK is that it can be derived by the compiler and needs no hand optimization.

1 Introduction

The trend in high-performance microprocessor design is toward increasing the computational power on chip. Unfortunately, memory speed is not increasing at the same rate. The result is an increase in the number of cycles for a memory access – a latency of 10 to 20 machine cycles is quite common.

Although cache helps to ameliorate these problems, it performs poorly on scientific calculations with working sets larger than the cache size. This situation has led many programmers to restructure their codes by hand to improve performance in the memory hierarchy. This is a step in the wrong direction. The user should not be creating programs that are specific to a particular machine — an expensive and tedious process. Instead, the task of specializing a program to a target architecture should fall to the compiler.

There is a long history of the use of sophisticated compiler optimizations to achieve machine independence. The Fortran I compiler included enough optimizations to make it possible for scientists to abandon machine-language programming. More recently, advanced vectorization technology has made it possible to write machine-independent vector programs in a sub-language of Fortran 77. We contend that it will be possible to achieve the same success for memory-hierarchy management on scalar processors. More precisely, enhanced compiler technology will enable programmers to express an algorithm in a natural, machine-independent form and achieve memory-hierarchy performance good enough to obviate the need for hand optimization.

To investigate the viability of memory-hierarchy management by the compiler, experiments were undertaken to determine if a compiler could automatically generate the block algorithms in LAPACK from the corresponding point algorithms expressed in Fortran 77 [4, 5, 7, 11]. Previous results have shown that LU-decomposition and Cholesky decomposition are blockable by the compiler, but that QR-decomposition is not blockable. This paper

[*]Research supported by ARPA through ONR Grant N00014-91-J-1989 and by NSF Grant CCR-9120008 and DARPA contract TV-ORA4466.01

[†]Department of Computer Science, Michigan Technological University, Houghton MI 49931

[‡]Department of Computational and Applied Mathematics, Rice University, Houston TX 77251-1892

re-examines the QR result and presents a block algorithm that is derivable by the compiler. Experiments with this algorithm show that the compiler derivable version performs well on small to moderate matrix sizes — sizes that are likely to appear on a single node of a massively parallel machine or on a workstation.

The paper begins with an introduction to the transformations used by the compiler to optimize algorithms for memory performance. Next, the LAPACK and compiler-derivable block algorithms are presented. Finally, an experiment comparing the performance of the two algorithms on three different computers is detailed.

2 Background

2.1 Iteration-space blocking

To improve the memory behavior of loops that access more data than fit in cache, the iteration space of a loop can be grouped into blocks whose working sets are small enough for cache to capture the available temporal reuse [14, 12, 13]. Consider the following loop nest.

```
 do 10 j = 1,n
   do 10 i = 1,m
10   a(i) = a(i) + b(j)
```

Assume the value of `m` is much greater than the size of the cache. Reuse of values exists for `b`, but not for `a`. To exploit `a`'s reuse, the iteration space of `j` is blocked as follows:

```
 do 10 j = 1,n,js
   do 10 i = 1,m
     do 10 jj = j, min(j+js-1,n)
10       a(i) = a(i) + b(jj)
```

Temporal reuse of `a` and `b` now occurs if `js` is less than half the cache size and there is no significant cache interference [10].

2.2 Loop Distribution

Loop distribution separates individual statements within a loop into multiple loop nests with the same loop header.[1] For example, in the following loop,

```
 do 10 i = 1,n
   do 20 j = 1,n
20   a(i,j) = a(i,j) + b(i,j)
   do 10 j = 1,n
10   c(i,j) = c(i,j) + d(i,j)
```

distributing the `i`-loop would give:

```
 do 20 i = 1,n
   do 20 j = 1,n
20   a(i,j) = a(i,j) + b(i,j)
 do 10 i = 1,n
   do 10 j = 1,n
10   c(i,j) = c(i,j) + d(i,j)
```

In the context of this paper, loop distribution is used to enable iteration-space blocking.

[1]Note that statements involved in a recurrence must remain in the same loop to ensure safety.

3 QR Decomposition from LAPACK

The LAPACK point algorithm for computing the QR decomposition consists of forming the sequence $A_{k+1} = V_k^T A_k$ for $k = 1, \ldots, n-1$. The initial matrix $A_1 = A$ has m rows and n columns, where for this study we assume $m \geq n$. The elementary reflectors $V_k = I - \tau_k v_k v_k^T$ update A_k in order that the first k columns of A_{k+1} form an upper triangular matrix. The update is accomplished by performing the matrix vector multiplication $w_k = A^T v_k$ followed by the rank one update $A_{k+1} = A_k - \tau_k v_k w_k^T$. Efficiency of the implementation of the level 2 BLAS subroutines determines the rate at which the decomposition is computed. For a more detailed discussion of the QR decomposition see Golub and Van Loan [8].

The LAPACK block QR decomposition is an attempt to recast the algorithm in terms of calls to level 3 BLAS using iteration-space blocking [7]. If the level 3 BLAS are hand-tuned for a particular architecture, the block QR algorithm may perform significantly better than the point version on large matrix sizes (those that cause the working set to be much larger than the cache size). However, the optimized BLAS codes are often not portable and the application programmer must rely upon the machine vendor to make the software investment to optimize the kernels for the architecture. The ability to express an algorithm in a machine-independent form with the compiler handling the machine-dependent optimizations is the desired goal.

Unfortunately, the block QR algorithm in LAPACK is not automatically derivable by a compiler [5]. The block application of a number of elementary reflectors involves both computation and storage that does not exist in the original point algorithm [7]. As an illustration, consider a block of two elementary reflectors:

$$\begin{aligned} Q = &\quad (I - \tau_1 v_1 v_1^T)(I - \tau_2 v_2 v_2^T), \\ = &\quad I - (v_1 v_2) \begin{pmatrix} \tau_1 & \tau_1\tau_2(v_1^T v_2) \\ 0 & \tau_2 \end{pmatrix} \begin{pmatrix} v_1^T \\ v_2^T \end{pmatrix}. \end{aligned}$$

The computation of the matrix

$$T = \begin{pmatrix} \tau_1 & \tau_1\tau_2(v_1^T v_2) \\ 0 & \tau_2 \end{pmatrix}$$

is not part of the original algorithm. Hence, the LAPACK version of block QR-decomposition is a different algorithm from the point version, rather than just a reshaping of the point algorithm for better performance.

In the next section, a compiler-derivable block algorithm for QR-decomposition is presented. This algorithm gives comparable performance to the LAPACK version on small matrices while retaining machine independence.

4 Compiler Derivable QR Decomposition

Consider the application of j elementary reflectors, V_k, to A_k :

$$A_{k+j} = \quad (I - \tau_{k+j-1} v_{k+j-1} v_{k+j-1}^T) \cdots (I - \tau_{k+1} v_{k+1} v_{k+1}^T)(I - \tau_k v_k v_k^T) A_k.$$

The compiler derivable algorithm, henceforth called *cd*-QR, only forms the first $k+j-1$ columns of A_{k+j} and then updates the remainder of matrix with the j elementary reflectors. The final update of the trailing $n-k-j$ columns is "rich" in floating point operations that the compiler organizes to best suit the underlying hardware. Code optimization techniques such as outer-loop unrolling are left to the compiler [6]. The derived algorithm depends upon the *compiler* for efficiency in contrast to the LAPACK algorithm that depends on hand optimization of the BLAS.

```
  do ii = 1, n, ib
    do i = ii, min0(ii+ib-1,n)
       do j = i+1, m
          a(j,i) = a(j,i)/(a(i,i)-beta)
       end do
       do j = i+1, n
          t1 = zero
          do k = i, m
             t1 = t1 + a(k,j)*a(k,i)
          enddo
          do k = i, m
             a(k,j) = a(k,j) - tau(i)*t1*a(k,i)
          enddo
      enddo
   enddo
enddo
```

FIG. 1. *Strip-Mined Point QR Decomposition*

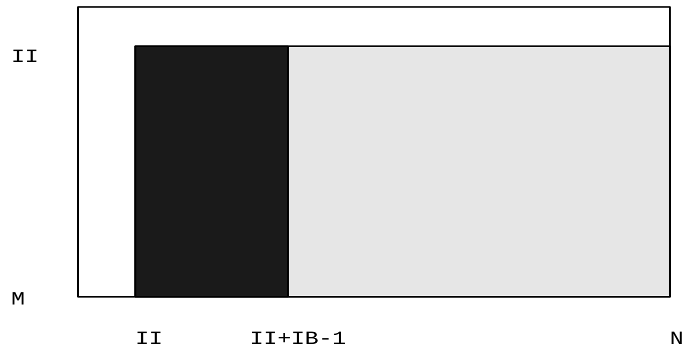

FIG. 2. *Regions of* a *Accessed by QR Decomposition*

4.1 Compiler Blockability of CD-QR

Cd-QR can be obtained from the point algorithm for QR decomposition using array section analysis, loop distribution and iteration-space blocking [5, 2, 9]. For reference, segments of the code for the point algorithm are shown in Figure 1. To complete iteration-space blocking of the code in Figure 1 to obtain *cd*-QR, the i-loop must be distributed around the loop that surrounds the computation of V_i and around the update before being interchanged with the j-loop. However, there is a recurrence between the definition and use of a(k,j) within the update section and the definition and use of a(j,i) in computation of V_i that appears to prevent distribution.

Figure 2 shows the sections of the array a(:,:) accessed for the entire execution of the i-loop. If the sections accessed by a(j,i) and a(k,j) are examined, a legal partial distribution of the i-loop is revealed. The section accessed by a(j,i) (the black region) is a subset of the section accessed by a(k,j) (both the black and gray regions). Since the recurrence exists for only a portion of the iteration space of the loop surrounding the update of a, the index-set of j can be split at the point j = i+ib-1 to create two loops: one that iterates over the iteration space where a(k,j) accesses the black region and one that iterates over the iteration space where a(k,j) accesses the gray region. The second loop can be reshaped using iteration-space blocking to obtain *cd*-QR because the accesses by a(k,j) are disjoint from the accesses by a(j,i).

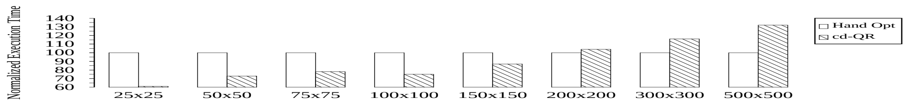

Fig. 3. *Performance on IBM RS/6000*

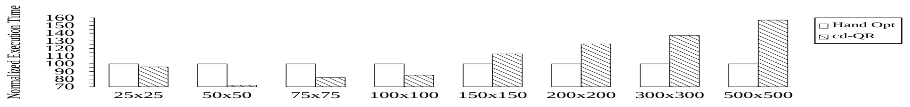

Fig. 4. *Performance on IBM ES/9000*

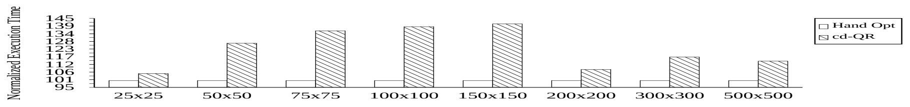

Fig. 5. *Performance on Convex*

5 Experiment

The performance of each block algorithm was compared on three different architectures: IBM RS/6000, IBM ES/9000 and Convex C-3240. A block size of 16 was used. *Cd*-QR was obtained by hand since Memoria, a compiler based on the ParaScope programming environment that performs memory optimizing transformations, does not support section analysis[3, 1]. Memoria was used to further optimize *cd*-QR by applying outer-loop unrolling. All software was compiled with full optimization on all the machines and additionally vectorization on the ES/9000 and Convex. On the RS/6000, version 2.2 of the *xlf* compiler was used. On the ES/9000, version 2.5.1 of the FORTVS compiler was used, and on the Convex, version 8.0 of the Fortran compiler was used. For the RS/6000 a hand-optimized version of DGEMV and DGEMM was used. On the Convex and ES/9000, the VECLIB and ESSL subroutine libraries were used, respectively.

Figures 3, 4 and 5 plot the performance results. Performance is reported in normalized execution time where the LAPACK version of the algorithm is the base performance and the performance of *cd*-QR is presented as a percentage of the LAPACK algorithm. Unless specified otherwise all software was written in FORTRAN and the results are for floating point words of 64 bits. We do not present the results of the point version of QR because it was never significantly better than either of the block algorithms.

On both the RS/6000 and ES/9000 *cd*-QR performed better than the LAPACK version for matrix sizes less than 150x150 (on the RS/6000 less than 200x200). The compiler came within a reasonable percentage (approximately 25%) of the LAPACK algorithm for matrices up to size 300x300 on the RS/6000 and 200x200 on the ES/9000. On the Convex, *cd*-QR was close to the performance of the hand-optimized LAPACK version (usually within 25%).

6 Summary

The goal of this paper is to show that compiler techniques allow the user to express QR decomposition naturally with the expectation of good memory-hierarchy performance. The results presented have extended previous work by demonstrating that a compiler can derive a block algorithm for QR decomposition that is effective on matrix sizes typically run on workstations or nodes of a massively parallel machine. Given that future machine designs are certain to have increasingly complex memory hierarchies, compilers will need to adopt increasingly sophisticated memory-management strategies so that programmers can remain free to concentrate on algorithm design and program logic.

Acknowledgments

Ken Kennedy and Richard Hanson provided the original motivation for this work. Bob Metzger ran the experiments on the Convex. Ken Kennedy, Keith Cooper and Danny Sorensen provided financial support for this research. Robert Reynolds and Phil Sweany gave helpful suggestions during the preparation of this document.

References

[1] D. Callahan, K. Cooper, R. Hood, K. Kennedy, L. Torczon, and S. Warren, *Parallel programming support in ParaScope*, in Proceedings of the 1987 DFVLR Conference on Parallel Processing in Science and Engineering, Koln, West Germany, June 1987.

[2] D. Callahan and K. Kennedy, *Analysis of interprocedural side effects in a parallel programming environment*, in Proceedings of the First International Conference on Supercomputing, Springer-Verlag, Athens, Greece, 1987.

[3] S. Carr, *Memory-Hierarchy Management*, PhD thesis, Rice University, Department of Computer Science, September 1992.

[4] S. Carr and K. Kennedy, *Blocking linear algebra codes for memory hierarchies*, in Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing, Chicago, IL, December 1989.

[5] ——, *Compiler blockability of numerical algorithms*, in Proceedings of Supercomputing '92, Minneapolis, MN, November 1992, pp. 114–124.

[6] ——, *Improving the ratio of floating-point operations to memory operations in loops*, Tech. Rep. TR93-08, Michigan Technological University, Dec. 1993. To appear in *ACM Transactions on Programming Languages and Systems*.

[7] J. Dongarra, I. Duff, D. Sorensen, and H. van der Vorst, *Solving Linear Systems on Vector and Shared-Memory Computers*, SIAM, Philadelphia, 1991.

[8] G. Golub and C. V. Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, 1989.

[9] P. Havlak and K. Kennedy, *An implementation of interprocedural bounded regular section analysis*, IEEE Transactions on Parallel and Distributed Systems, 2 (1991), pp. 350–360.

[10] M. S. Lam, E. E. Rothberg, and M. E. Wolf, *The cache performance and optimizations of blocked algorithms*, in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Santa Clara, California, 1991, pp. 63–74.

[11] R. Lehoucq, *Implementing efficient and portable dense matrix factorizations*, in Proceedings of the Fifth SIAM Conference on Parallel Processing for Scientific Computing, 1992.

[12] A. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications*, PhD thesis, Rice University, May 1989.

[13] M. Wolf and M. Lam, *A data locality optimizing algorithm*, in Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991.

[14] M. Wolfe, *Iteration space tiling for memory hierarchies*, in Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing, Dec. 1987.