Scheduling of QR Factorization Algorithms on SMP and Multi-Core Architectures

FLAME Working Note #24

Gregorio Quintana-Ortí* Enrique S. Quintana-Ortí* Ernie Chan†
Robert A. van de Geijn† Field G. Van Zee†

Abstract

This paper examines the scalable parallel implementation of QR factorization of a general matrix, targeting SMP and multi-core architectures. Two implementations of algorithms-by-blocks are presented. Each implementation views a block of a matrix as the fundamental unit of data, and likewise, operations over these blocks as the primary unit of computation. The first is a conventional blocked algorithm similar to those included in libFLAME and LAPACK but expressed in a way that allows operations in the so-called critical path of execution to be computed as soon as their dependencies are satisfied. The second algorithm captures a higher degree of parallelism with an approach based on Givens rotations while preserving the performance benefits of algorithms based on blocked Householder transformations. We show that the implementation effort is greatly simplified by expressing the algorithms in code with the FLAME/FLASH API, which allows matrices stored by blocks to be viewed and managed as matrices of matrix blocks. The SuperMatrix run-time system utilizes FLASH to assemble and represent matrices but also provides out-of-order scheduling of operations that is transparent to the programmer. Scalability of the solution is demonstrated on a ccNUMA platform with 16 processors.

1 Introduction

The process of extracting parallelism from commonly used libraries must be reevaluated with the emergence of SMP architectures with many processors, multi-core systems that will soon have many cores, and hardware accelerators such as the Cell processor [3, 23]. These "many-threaded" architectures change many of the parameters that have driven the development of previous generations of such libraries. In particular, cores will be faster, latencies will be lower, the amount of memory per core will be smaller—causing data locality to be of greater concern—and parallelism will be demanded for smaller problems. In this paper, we explore techniques that may become part of the solution in the context of one dense linear algebra operation, the QR factorization [14, 39], which is among the most useful (and challenging) operations to parallelize.

Traditional algorithms for QR factorization based on Householder transformations, along with LU factorization with partial pivoting, exhibit a shared property: periodically, a column of the matrix must be updated as part of a sub-operation that resides along the critical path of execution. In order to compute the k-th Householder transformation, the k-th column must have been updated with respect to all previous transformations. This property significantly constrains the order in which the computations may be performed.

The problem is compounded by the fact that, for scalability and data locality, it is best to view and store the matrix as a two dimensional array of submatrices (blocks) and to compute sub-operations (tasks) with these blocks. This fact creates a situation where the column needed for computing the Householder

*Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071 - Castellón (Spain), {gquintan,quintana}@icc.uji.es.

†Department of Computer Sciences, The University of Texas at Austin, Austin, Texas 78712, {echan,rvdg,field}@cs.utexas.edu.

transformation most likely spans more than one block. Viewing and storing matrices by blocks was also observed for out-of-core dense linear algebra computations [34] and the implementation of dense linear algebra operations on distributed-memory architectures [18, 33, 37].

In this paper we explain how to continue to achieve high performance while keeping the underlying implementation simple. The contributions of this paper include the following:

- We demonstrate that high performance can be attained by programs written at a high level of abstraction, even those that implement algorithms for complex operations within sophisticated environments such as many-threaded architectures.
- We compare and contrast traditional blocked algorithms for the QR factorization via Householder transformations to a new algorithm-by-blocks that is similar to a blocked implementation of the QR factorization via Givens rotations.
- We show how the FLASH extension of FLAME elegantly supports storage by blocks for these algorithms and also how the SuperMatrix run-time system transparently supports out-of-order computation on blocks.

This paper is structured as follows. In Section 2 we review the description of the QR factorization via Householder transformations from [16]. Then, in Section 3, we present two algorithms-by-blocks. The first is an implementation of the conventional algorithm found in LAPACK [2] and libFLAME [4, 15] which is improved to advance operations that are in the critical path from "future" iterations. The second algorithm exploits parallelism through an approach based on Givens rotations while maintaining both level-3 BLAS performance and the reduced computational cost of the QR factorization via Householder transformations. The implementation of these algorithms using the FLAME/FLASH API is outlined in Section 4. In Section 5 we demonstrate the scalability of these solutions on an SGI Altix 350 composed of 16 Intel Itanium2 processors, and in Section 6 we provide a few concluding remarks.

In the paper, we adopt the following conventions. Matrices, vectors, and scalars are denoted by upper-case, lower-case, and lower-case Greek letters, respectively. The identity matrix is denoted by I and e_0 will denote the first column of this matrix. The dimensions and lengths of such matrices and vectors are generally obvious from the context.

Algorithms in this paper are given in a notation that we have developed as part of the FLAME project [4, 15]. If one keeps in mind that the thick lines in the partitioned matrices and vectors relate to how far the computation has proceeded, we believe the notation is mostly intuitive. Otherwise, we suggest that the reader consult some of these related papers for a broader review of the FLAME methodology.

2 Computing the QR factorization via Householder Transformations

Given an $m \times n$ real-valued matrix A, with $m \geq n$, its QR factorization is given by $A = QR$ where the $m \times m$ matrix Q is orthogonal (that is, $Q^T Q = QQ^T = I$), and the $m \times n$ matrix R is upper triangular.

There are many different methods for computing the QR factorization, including those based on Givens rotations, orthogonalization via Gram-Schmidt and Modified Gram-Schmidt, and Householder transformations [14, 39].

For dense matrices, the method of choice largely depends on how the factorization is subsequently used, the stability of the system, and the dimensions of the matrix. For problems where $m \gg n$, the method based on Householder transformations is typically the algorithm of choice, especially when Q does not need to be explicitly computed.

Algorithm: $[A,\ b] := \mathrm{QR}_{\mathrm{UNB}}(A)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$ and $b \rightarrow \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right)$

where A_{TL} is 0×0 and b_T has 0 elements

while $n(A_{BR}) \neq 0$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \rightarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right)$$

where α_{11} and β_1 are scalars

$$\left[\left(\begin{array}{c} 1 \\ \hline u_2 \end{array}\right), \eta, \beta_1\right] := h\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right)$$

$\alpha_{11} := \eta$

$a_{21} := u_2$

$w^T := a_{12}^T + u_2^T A_{22}$

$$\left(\begin{array}{c} a_{12}^T \\ \hline A_{22} \end{array}\right) := \left(\begin{array}{c} a_{12}^T - \beta_1 w^T \\ \hline A_{22} - \beta_1 u_2 w^T \end{array}\right)$$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} b_T \\ \hline b_B \end{array}\right) \leftarrow \left(\begin{array}{c} b_0 \\ \hline \beta_1 \\ \hline b_2 \end{array}\right)$$

endwhile

Figure 1: Unblocked Householder QR factorization.

2.1 Householder transformations (reflectors)

Given the real-valued vector x of length m, partition $x = \left(\begin{array}{c} \chi_0 \\ \hline x_1 \end{array}\right)$, where χ_0 equals the first element of x. The Householder vector associated with x is defined as the vector $u = \left(\begin{array}{c} 1 \\ \hline x_1/\nu_0 \end{array}\right)$, where $\nu_0 = \chi_0 + \mathrm{sign}(\chi_0)\|x\|_2$. If $\beta = \frac{2}{u^T u}$ then $(I - \beta u u^T)x = \eta e_0$, annihilating all but the first element of x, which becomes $\eta = -\mathrm{sign}(\chi_0)\|x\|_2$. The transformation $I - \beta u u^T$ is referred to as a Householder transformation or *reflector*.

Let us introduce the notation $[u, \eta, \beta] := h(x)$ as the computation of the above mentioned η, u, and β from vector x and the notation $H(x)$ for the corresponding transformation $(I - \beta u u^T)$. An important feature of $H(x)$ is that it is orthogonal and symmetric $(H(x)^T = H(x))$.

2.2 A simple (unblocked) algorithm

The computation of the QR factorization proceeds as described in Figure 1. The idea is that Householder transformations are computed to successively annihilate all the subdiagonal elements of matrix A. The Householder vectors are stored by overwriting those elements that have been previously annihilated. Upon completion, matrix R will have overwritten the upper triangular part of the matrix while the Householder vectors are stored in the strictly lower trapezoidal part of the matrix. The scalars β discussed above are stored in the vector b of length n. The cost of this algorithm is $2n^2(m - n/3)$ floating-point arithmetic operations (FLOPs).

If the matrix Q is explicitly desired, it can be formed by carefully accumulating the product $H_0 \cdot H_1 \cdots H_{n-1} = Q$, where H_k equals the $(k+1)$-th Householder transformation computed as part of the factorization described above. Frequently, Q does not need to be formed explicitly, and thus we will not discuss the issue further.

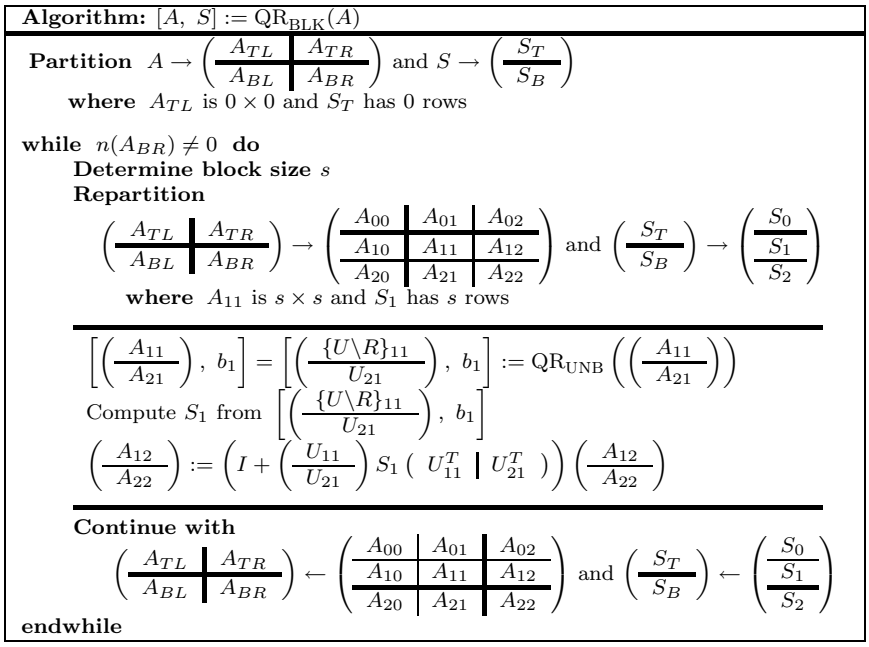

Figure 2: Blocked Householder QR factorization.

2.3 A high-performance (blocked) algorithm

It is well-known that high performance can be achieved in a portable fashion by casting algorithms in terms of matrix-matrix multiplication [10, 12]. We now review this technique as applied to the QR factorization [6].

Two observations play a key role:

- Let $u_0, \ldots, u_{s-1}$ equal the first s Householder vectors computed as part of the factorization and let $\beta_0, \ldots, \beta_{s-1}$ be the corresponding scalars so that $H_j = (I - \beta_j u_j u_j^T)$, $j = 0, \ldots, s-1$. Then $H_0 \cdot H_1 \cdots H_{s-1} = I + U_s S_s U_s^T$, where U_s is an $n \times s$ unit lower-trapezoidal matrix, S_s is an $s \times s$ upper-triangular matrix, and the subdiagonal of the $(j+1)$-th column of U_s equals u_j, $j = 0, \ldots, s-1$.
- The QR factorization of the first k columns of A yields the same k vectors u_k and the same values in the upper triangular part of those k columns as would a full QR factorization.

These two observations underlie the blocked algorithm given in Figure 2. Provided $s \ll m, n$ the cost of this algorithm equals that of the unblocked algorithm. The algorithm returns the $s \times s$ upper triangular "S_1" matrices that are part of the block Householder transformation stored into an $n \times s$ matrix S. This algorithm represents a departure from traditional implementations such as those found in LAPACK [2] and will allow us to avoid recomputation of those matrices. In [21, 30, 38], a slightly more efficient way of accumulating Householder transformations is given, which we call the UT-transform. Since the compact WY-transform $I + USU^T$ [32] is better known, we focus our discussion on it instead.

2.4 Parallelism within the BLAS

Implementations of the QR factorization, whether unblocked or blocked, typically are written so that the bulk of the computation is performed by the *Basic Linear Algebra Subprograms* (BLAS), which export a standardized interface to common operations such as matrix-vector (level-2 BLAS [11]) and matrix-matrix multiplication (level-3 BLAS [10]).

Parallelism can be attained within each invocation of a BLAS routine with the following benefits:

- The approach allows legacy libraries, such as LAPACK, to be used without modifying the library source code.

- Parallelism within sub-operations, e.g., the update of $\left(\frac{A_{12}}{A_{22}}\right)$, can be achieved through multithreaded implementations of the BLAS.

Disadvantages of this approach, on the other hand, include:

- The degree of parallelism achieved is potentially limited by the efficiency of the underlying multithreaded BLAS implementation.
- The end of each BLAS routine executed becomes an implicit synchronization point (or barrier) for the threads.
- For many operations the choice of algorithmic variant can significantly impact performance [5].

In the next section we propose two algorithms to overcome these difficulties.

3 Algorithms-by-blocks

Since the early 1990s, various researchers [9, 13, 20, 29] have proposed that matrices should be stored by blocks rather than the more customary column-major storage used in Fortran and row-major storage used in C. Storing matrices by blocks recursively reflects a generalization of that idea. It was originally thought that storing submatrices contiguously would yield a performance improvement due to improved data locality. More recently, we have further advocated that matrices should not only be stored by blocks but also *viewed* as blocks through appropriate abstraction mechanisms [7, 8, 26]. These abstractions simplify the task of expressing basic units of computations in terms of submatrices.

For our discussion below, it suffices to think of matrices as being partitioned into square blocks. In other words, there is one level of recursion in the data structure. When considering the algorithm descriptions, the reader need not concern himself with how the blocks are physically stored.

Consider for simplicity that $m = m_s s$ and $n = n_s s$ with $m_s,\ n_s$ being two integers and s the block size, then we envision the partitioned matrix in a typical state at the top of the loop in the algorithm in Figure 2 as

$$A = \left(\begin{array}{ccc|c|ccc} \{U\backslash R\}^{0,0} & \cdots & R^{0,k-1} & R^{0,k} & \cdots & R^{0,n_s-1} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ U^{k-1,0} & \cdots & \{U\backslash R\}^{k-1,k-1} & R^{k-1,k} & \cdots & R^{k-1,n_s-1} \\ \hline U^{k,0} & \cdots & U^{k,k-1} & A^{k,k} & \cdots & A^{k,n_s-1} \\ \hline \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ U^{m_s-1,0} & \cdots & U^{m_s-1,k-1} & A^{m_s-1,k} & \cdots & A^{m_s-1,n_s-1} \end{array}\right), \tag{1}$$

where all blocks in $A,\ R,\ U$ are $s \times s$. Here the thick and thin lines are visual hints to indicate that, in the repartitioning at the top of the algorithm loop,

$$A_{11} = A^{k,k}, \quad A_{21} = \left(\begin{array}{c} A^{k+1,k} \\ \vdots \\ A^{m_s-1,k} \end{array}\right), \quad \text{etc.}$$

3.1 Algorithm-by-blocks I

Our first algorithm that exploits viewing the matrix as a collection of blocks implements the algorithm in Figure 2. We assume that *no* parallelism will be extracted from the factorization of the current panel (that is, within the call to the unblocked algorithm).

We discuss each operation in the body of the loop separately:

- The factorization of the current panel,

$$\left[\left(\frac{A_{11}}{A_{21}}\right),\ b_1\right] := \left[\left(\frac{\{U\backslash R\}_{11}}{U_{21}}\right),\ b_1\right] = \mathrm{QR}_{\mathrm{UNB}}\left(\left(\frac{A_{11}}{A_{21}}\right)\right),$$

poses an inherent problem due to our basic assumption that no parallelism occurs within this operation. Our algorithm copies the individual blocks in $\left(\frac{A_{11}}{A_{21}}\right)$ to a temporary matrix so that it can be factored using a single-threaded conventional unblocked algorithm, after which the result is copied back into the original blocks.

Optimization: Since this operation is inherently sequential, it is desirable to use a sequential blocked algorithm with a block smaller than s to cast the computation in terms of level-3 BLAS operations, similar to the recursive algorithm in [13, 35].

- Compute S_1. In [21, 30, 38], it is shown how this operation can be cast so that essentially all computation occurs in the operation $U_{11}^T U_{11} + U_{21}^T U_{21}$ which, in terms of the partitioned matrix in (1), can be obtained as

$$\begin{array}{rcl} U_{11}^T U_{11} + U_{21}^T U_{21} & = & \left(\frac{U^{k,k}}{\vdots \atop U^{m_s-1,k}}\right)^T \left(\frac{U^{k,k}}{\vdots \atop U^{m_s-1,k}}\right) \\ & = & (U^{k,k})^T U^{k,k} + (U^{k+1,k})^T U^{k+1,k} + \cdots + (U^{m_s-1,k})^T U^{m_s-1,k}. \end{array}$$

Therefore, a certain degree of parallelism can be obtained at the expense of a careful summation of contributions from different threads. For simplicity, in our implementation this operation is combined with the factorization of the current panel and performed by only one thread. The cost of computing S_1 in this manner is similar to that of the factorization of the current panel.

- The bulk of independent computations comes from the update

$$\left(\frac{A_{12}}{A_{22}}\right) := \left(I + \left(\frac{U_{11}}{U_{21}}\right) S_1 \left(\begin{array}{c|c} U_{11}^T & U_{21}^T \end{array}\right)\right) \left(\frac{A_{12}}{A_{22}}\right),$$

which can be decomposed as follows:

  - First, use a temporary matrix W to compute

$$\begin{array}{rcl} W & := & \left(\frac{U_{11}}{U_{21}}\right)^T \left(\frac{A_{12}}{A_{22}}\right) \\ & = & \left(\frac{U^{k,k}}{\vdots \atop U^{m_s-1,k}}\right)^T \left(\begin{array}{ccc} A^{k,k} & \cdots & A^{k,n_s-1} \\ \hline \vdots & \ddots & \vdots \\ A^{m_s-1,k} & \cdots & A^{m_s-1,n_s-1} \end{array}\right), \end{array} \tag{2}$$

    which exposes ample parallelism in form of the computations $(U^{i,k})^T A_{ij}$, $i = k, \ldots, m_s - 1$, $j = k, \ldots, n_s - 1$.
  - Next, perform $W := S_1 W$ by multiplying each $s \times s$ block in W as

$$W := S_1 W = \left(\begin{array}{ccc} S_1 W^{k+1} & \cdots & S_1 W^{n_s-1} \end{array}\right). \tag{3}$$

  - Finally, compute

$$\begin{array}{rcl} \left(\frac{A_{12}}{A_{22}}\right) & := & \left(\frac{A_{12}}{A_{22}}\right) + \left(\frac{U_{11}}{U_{21}}\right) W \\ & = & \left(\begin{array}{ccc} A^{k,k+1} & \cdots & A^{k,n_s-1} \\ \hline \vdots & \ddots & \vdots \\ A^{m_s-1,k} & \cdots & A^{m_s-1,n_s-1} \end{array}\right) + \left(\frac{U^{k,k}}{\vdots \atop U^{m_s-1,k}}\right) \left(\begin{array}{ccc} W^{k+1} & \cdots & W^{n_s-1} \end{array}\right), \end{array} \tag{4}$$

which again exposes ample parallelism in the form of individual computations $A^{i,j} := A^{i,j} + U^{i,k}W^j$, $i = k, \ldots, m_s - 1$, $j = k+1, \ldots, n_s - 1$.

The fundamental question now becomes how to create as much parallelism as possible without the factorization of the current panel and the formation of S_1 becoming a bottleneck because of dependencies. This problem is addressed by applying updates to future panels as early as possible so that the factorization of those panels can commence by one thread while other updates of later panels are being performed by other threads. While this idea is conceptually simple, it is often painful to implement. In Section 4 we will discuss how to manage the complexity that arises when employing this technique.

The algorithm-by-blocks described above is reminiscent of the algorithm by Toledo [35] for out-of-core QR factorization when blocks are stored contiguously out-of-core. In Toledo's algorithm, the QR factorization of $\left(\frac{A_{11}}{A_{21}}\right)$ requires a block to be brought into memory a few columns at a time, which requires copying from storage-by-blocks to a "flat" matrix so that an in-core factorization of those columns can proceed.

3.2 Algorithm-by-blocks II

An alternative approach to computing the QR factorization uses Givens rotations. Given a vector $\begin{pmatrix} \chi_0 \\ \chi_1 \end{pmatrix}$, γ and σ can be determined so that $\gamma^2 + \sigma^2 = 1$ (γ and σ are the cosine and sine of an angle, thus the name "rotation") and $\begin{pmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{pmatrix} \begin{pmatrix} \chi_0 \\ \chi_1 \end{pmatrix} = \begin{pmatrix} \check{\chi}_0 \\ 0 \end{pmatrix}$ with $\check{\chi}_0 = \pm\sqrt{\chi_0^2 + \chi_1^2}$.

Givens rotations can be used to compute the QR factorization of an $m \times n$ matrix $A = (\alpha_{i,j})$ observing that, by choosing $\chi_0 = \alpha_{0,0}$ and $\chi_1 = \alpha_{1,0}$ in the above discussion, then

$$\begin{pmatrix} \gamma & \sigma & 0 & \ldots & 0 \\ -\sigma & \gamma & 0 & \ldots & 0 \\ 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \ldots & 1 \end{pmatrix} \begin{pmatrix} \alpha_{0,0} & \alpha_{0,1} & \alpha_{0,2} & \ldots & \alpha_{0,n-1} \\ \alpha_{1,0} & \alpha_{1,1} & \alpha_{1,2} & \ldots & \alpha_{1,n-1} \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \ldots & \alpha_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \alpha_{m-1,2} & \ldots & \alpha_{m,n-1} \end{pmatrix}$$

$$= \begin{pmatrix} \check{\alpha}_{0,0} & \check{\alpha}_{0,1} & \check{\alpha}_{0,2} & \ldots & \check{\alpha}_{0,n-1} \\ 0 & \check{\alpha}_{1,1} & \check{\alpha}_{1,2} & \ldots & \check{\alpha}_{1,n-1} \\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \ldots & \alpha_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{m-1,0} & \alpha_{m-1,1} & \alpha_{m-1,2} & \ldots & \alpha_{m-1,n-1} \end{pmatrix}.$$

In practice, $\check{\alpha}_{0,0} = \check{\chi}_{0,0}$ overwrites $\alpha_{0,0}$ while $\begin{pmatrix} \check{\alpha}_{0,j} \\ \check{\alpha}_{1,j} \end{pmatrix} = \begin{pmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{pmatrix} \begin{pmatrix} \alpha_{0,j} \\ \alpha_{1,j} \end{pmatrix}$, $j = 1, \ldots, n-1$, overwrite the corresponding elements of A. By similarly computing and applying rotations with $\alpha_{0,0}$ and $\alpha_{i,0}$, $i = 2, \ldots, m-1$, all elements of the first column below the diagonal are annihilated, after which the process is repeated with the submatrix that starts at $\alpha_{1,1}$.

The order in which elements are annihilated is not unique. On distributed-memory parallel architectures, it was often suggested to use the pair $\alpha_{i-1,0}$ and $\alpha_{i,0}$ to annihilate $\alpha_{i,0}$, $i = m-1, \ldots, 1$ before moving on to the next column. This procedure allows for nearest-neighbor communication on a mesh of processors and can create a "wavefront" of computation [27].

The problem with using Givens rotations is that the bulk of the computations in these transformations is cast in terms of vector operations (level-1 BLAS [24]), which do not attain high performance on current architectures with deep cache hierarchies. For instance, the application of the first rotation,

$$\begin{pmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{pmatrix} \begin{pmatrix} \alpha_{0,1} & \alpha_{0,2} & \ldots & \alpha_{0,n-1} \\ \alpha_{1,1} & \alpha_{1,2} & \ldots & \alpha_{1,n-1} \end{pmatrix} = \begin{pmatrix} \check{\alpha}_{0,1} & \check{\alpha}_{0,2} & \ldots & \check{\alpha}_{0,n-1} \\ \check{\alpha}_{1,1} & \check{\alpha}_{1,2} & \ldots & \check{\alpha}_{1,n-1} \end{pmatrix},$$

requires $O(n)$ operations on $O(n)$ data, limiting the speed of the processor to that of the memory subsystem. In addition, computing the QR factorization using Givens rotations is more expensive, as it requires about $3n^2(m-n/3)$ FLOPs—that is, $n^2(m-n/3)$ FLOPs more than the corresponding operation via Householder transformations.

Thus, the natural question becomes whether a blocked version of this algorithm can be attained that 1) performs roughly the same number of operations as the classic Householder QR factorization, 2) employs level-3 BLAS operations, and 3) exhibits the same level of parallelism as the algorithm that employs Givens rotations. We show next that such an implementation is indeed possible.

Assume for simplicity that A can be partitioned into $m_t \times n_t$ blocks of size $t \times t$, with $m = m_t t$ and $n = n_t t$,

$$A = \begin{pmatrix} M^{0,0} & M^{0,1} & M^{0,2} & \cdots & M^{0,n_t-1} \\ M^{1,0} & M^{1,1} & M^{1,2} & \cdots & M^{1,n_t-1} \\ M^{2,0} & M^{2,1} & M^{2,2} & \cdots & M^{2,n_t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M^{m_t-1,0} & M^{m_t-1,1} & M^{m_t-1,2} & \cdots & M^{m_t-1,n_t-1} \end{pmatrix}.$$

The first step is to compute the QR factorization $M^{0,0} = Q^{0,0}R^{0,0}$ using an algorithm based on Householder transformations and to update the remaining blocks in the first block row of A accordingly:

$$\begin{aligned} A &:= \begin{pmatrix} (Q^{0,0})^T & 0 & 0 & \cdots & 0 \\ 0 & I & 0 & \cdots & 0 \\ 0 & 0 & I & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & I \end{pmatrix} \begin{pmatrix} M^{0,0} & M^{0,1} & M^{0,2} & \cdots & M^{0,n_t-1} \\ M^{1,0} & M^{1,1} & M^{1,2} & \cdots & M^{1,n_t-1} \\ M^{2,0} & M^{2,1} & M^{2,2} & \cdots & M^{2,n_t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M^{m_t-1,0} & M^{m_t-1,1} & M^{m_t-1,2} & \cdots & M^{m_t-1,n_t-1} \end{pmatrix} \\ &= \begin{pmatrix} R^{0,0} & \tilde{M}^{0,1} & \tilde{M}^{0,2} & \cdots & \tilde{M}^{0,n_t-1} \\ M^{1,0} & M^{1,1} & M^{1,2} & \cdots & M^{1,n_t-1} \\ M^{2,0} & M^{2,1} & M^{2,2} & \cdots & M^{2,n_t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M^{m_t-1,0} & M^{m_t-1,1} & M^{m_t-1,2} & \cdots & M^{m_t-1,n_t-1} \end{pmatrix}. \end{aligned}$$

(Notice that in this entire discussion, we refer to "applying" a matrix Q, which is never formed explicitly. Instead, acculmulated Householder transformations are applied.) Provided t is large enough, the computation of this $t \times t$ factorization can be performed using the blocked algorithm in Figure 2 choosing $s \ll t$. This factorization results in a level-3 BLAS algorithm that performs approximately $4t^3/3$ FLOPs. The application of the Householder transformations to the remaining portion of the first block row may be similarly expressed in terms of level-3 BLAS operations, incurring a cost of $2t^3(n_t - 1)$ FLOPs.

Next, the QR factorization

$$\begin{pmatrix} R^{0,0} \\ M^{1,0} \end{pmatrix} = Q^{1,0}R^{1,0} = \begin{pmatrix} \Gamma^{1,0} & \Sigma^{0,1} \\ \Sigma^{1,0} & \Gamma^{0,1} \end{pmatrix} \begin{pmatrix} \check{R}^{0,0} \\ 0 \end{pmatrix} \tag{5}$$

is computed using the algorithm based on Householder transformations, and the corresponding transforma-

tions are applied to the first two block rows:

$$
\begin{aligned}
A \;:=\; & \begin{pmatrix} (\Gamma^{1,0})^T & (\Sigma^{1,0})^T & 0 & \dots & 0 \\ (\Sigma^{0,1})^T & (\Gamma^{0,1})^T & 0 & \dots & 0 \\ 0 & 0 & I & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & I \end{pmatrix} \begin{pmatrix} R^{0,0} & \tilde{M}^{0,1} & \tilde{M}^{0,2} & \dots & \tilde{M}^{0,n_t-1} \\ M^{1,0} & M^{1,1} & M^{1,2} & \dots & M^{1,n_t-1} \\ M^{2,0} & M^{2,1} & M^{2,2} & \dots & M^{2,n_t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M^{m_t-1,0} & M^{m_t-1,1} & M^{m_t-1,2} & \dots & M^{m_t-1,n_t-1} \end{pmatrix} \\
= & \begin{pmatrix} \check{R}^{0,0} & \check{M}^{0,1} & \check{M}^{0,2} & \dots & \check{M}^{0,n_t-1} \\ 0 & \check{M}^{1,1} & \check{M}^{1,2} & \dots & \check{M}^{1,n_t-1} \\ M^{2,0} & M^{2,1} & M^{2,2} & \dots & M^{2,n_t-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ M^{m_t-1,0} & M^{m_t-1,1} & M^{m_t-1,2} & \dots & M^{m_t-1,n_t-1} \end{pmatrix}.
\end{aligned}
$$

Here, the pre-multiplication by $\begin{pmatrix} \Gamma^{1,0} & \Sigma^{0,1} \\ \Sigma^{1,0} & \Gamma^{1,1} \end{pmatrix}^T$ merely represents the action of the Householder transformations that resulted from the factorization of $\begin{pmatrix} R^{0,0} \\ M^{1,0} \end{pmatrix}$ with the Householder transformations overwriting $M^{1,0}$. The application of these transformations to $\begin{pmatrix} M^{0,j} \\ M^{1,j} \end{pmatrix}$, $j = 1, \dots, n_t - 1$, only requires the transformations stored in $M^{1,0}$ in addition to the blocks being updated since zeroes in $R^{0,0}$ can be "preserved" [16, 28]. Moreover, by exploiting the upper triangular structure of $R^{0,0}$, the cost of the QR factorization in (5) is reduced to $t^2(s + 2t)$ FLOPs and the update of the remaining part of the first two block rows becomes $t^2(s + 4t)(n_t - 1)$ FLOPs. Level-3 BLAS performance can still be attained from these two classes of operations while exploiting the block upper triangular structure of $R^{0,0}$. For that purpose, we employ a blocked algorithm that operates on s columns at a time, which has been used previously in the context of the computation of a "tiled" out-of-core QR factorization in [16] and the computation of the matrix disk function of a matrix pencil in [28].

The factorization proceeds like the algorithm based on Givens rotations, annihilating the remaining blocks below the diagonal block of the first column block, from $i = 2, \dots, m_t - 1$, and exploiting the upper triangular structure of the diagonal block. After that, the process is repeated with the submatrix that starts at $M^{1,1}$. Thus, one can conceptually think of this procedure as an algorithm based on Givens rotations that computes with blocks as the unit of data.

Provided $t \ll m, n$ and neglecting lower order terms, the overall cost of the algorithm-by-blocks II is

$$
\begin{aligned}
& \textstyle\sum_{k=0}^{n_t-1} \left(4t^3/3 + \sum_{j=k+1}^{n_t-1} 2t^3 + \sum_{i=k+1}^{m_t-1} \left(t^2(s+2t) + \sum_{j=k+1}^{n_t-1} t^2(s+4t) \right) \right) \\
& \quad \approx \textstyle\sum_{k=0}^{n_t-1} \sum_{i=k+1}^{m_t-1} \sum_{j=k+1}^{n_t-1} t^2(s+4t) \\
& \quad \approx 2n^2(m - n/3)\left(1 + \tfrac{s}{2t}\right) \text{ FLOPs.}
\end{aligned}
$$

This cost equals that of the QR factorization of an $m \times n$ matrix using Householder transformations when $s \ll t$, in which case $s/t \to 0$ and the overhead becomes negligible. In practice, the ratio s/t is small and the overhead is minor. For instance if $s = 16$ and $t = 256$, the algorithm only incurs $\frac{s/(2t)}{1+s/(2t)} \approx 3\%$ overhead. These values of s and t are close to the optimal values determined empirically during our study.

The level of parallelism realized from the algorithm-by-blocks II depends on the size of t relative to the problem dimension where smaller values of t provide a higher level of concurrency. Unfortunately, decreasing t may negatively affect the performance of the algorithmic subproblems since s must be decreased correspondingly if overhead is to be kept low. In these small blocksize situations, level-3 BLAS performance is less likely to be attained due to the smaller matrix dimensions.

4 Tools

In this section we briefly review some of the tools that the FLAME project puts at our disposal.

```
FLA_Error FLA_QR_blk( FLA_Obj A, FLA_Obj S, int nb_alg )
{
  FLA_Obj ATL, ATR,     A00, A01, A02,
          ABL, ABR,     A10, A11, A12,
                        A20, A21, A22;
  FLA_Obj ST,           S0,
          SB,           S1,
                        S2;
  int     s;

  FLA_Part_2x2( A,     &ATL, &ATR,
                       &ABL, &ABR,     0, 0, FLA_TL );
  FLA_Part_2x1( S,     &ST,
                       &SB,            0, FLA_TOP );

  while ( FLA_Obj_width ( ABR ) != 0 )
  {
    s = min( nb_alg, FLA_Obj_width( ABR ) );

    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                           /* ************* */   /* ******************** */
                                                  &A10, /**/ &A11, &A12,
                              ABL, /**/ ABR,      &A20, /**/ &A21, &A22,
                              s, s, FLA_BR );
    FLA_Repart_2x1_to_3x1( ST,       &S0,
                           /* ** */   /* ** */
                                      &S1,
                              SB,      &S2,     s, FLA_BOTTOM );

    /*--------------------------------------------------------------*/

    /* [ ( A11,       := QR_accum_S_unb( ( A11,
           A21 ), S1 ]                     A21 ) ); */

    FLA_QR_accum_S_unb( A11,
                        A21, S1 );

    /* ( A11,   := I + ( U11   * S1 * ( U11^T U21^T ) * ( A12,
         A22  )          U21 )                           A22  ) */

    FLA_QR_update_blk( A11, A12,
                       A21, A22, S1 );

    /*--------------------------------------------------------------*/

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                                                      A10, A11, /**/ A12,
                              /* ************** */   /* ****************** */
                                &ABL, /**/ &ABR,        A20, A21, /**/ A22,
                                FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &ST,        S0,
                                          S1,
                              /* ** */   /* ** */
                                &SB,        S2,     FLA_TOP );

  }

  return FLA_SUCCESS;
}
```

```
FLA_Error FLASH_QR_blk( FLA_Obj A, FLA_Obj S )
{
  FLA_Obj ATL, ATR,     A00, A01, A02,
          ABL, ABR,     A10, A11, A12,
                        A20, A21, A22;
  FLA_Obj ST,           S0,
          SB,           S1,
                        S2;

  FLA_Part_2x2( A,     &ATL, &ATR,
                       &ABL, &ABR,     0, 0, FLA_TL );
  FLA_Part_2x1( S,     &ST,
                       &SB,            0, FLA_TOP );

  while ( FLA_Obj_width ( ABR ) != 0 )
  {
    FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                           /* ************* */   /* ******************** */
                                                  &A10, /**/ &A11, &A12,
                              ABL, /**/ ABR,      &A20, /**/ &A21, &A22,
                              1, 1, FLA_BR );
    FLA_Repart_2x1_to_3x1( ST,       &S0,
                           /* ** */   /* ** */
                                      &S1,
                              SB,      &S2,     1, FLA_BOTTOM );

    /*--------------------------------------------------------------*/

    /* [ ( A11,       := QR_accum_S_unb( ( A11,
           A21 ), S1 ]                     A21 ) ); */

    ENQUEUE_FLA_QR_accum_S_unb( A11,
                                A21, S1 );

    /* ( A11,   := I + ( U11   * S1 * ( U11^T U21^T ) * ( A12,
         A22  )          U21 )                           A22  ) */

    FLASH_QR_update_blk( A11, A12,
                         A21, A22, S1 );

    /*--------------------------------------------------------------*/

    FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                                                      A10, A11, /**/ A12,
                              /* ************** */   /* ****************** */
                                &ABL, /**/ &ABR,        A20, A21, /**/ A22,
                                FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &ST,        S0,
                                          S1,
                              /* ** */   /* ** */
                                &SB,        S2,     FLA_TOP );

  }

  FLASH_Queue_exec( );

  return FLA_SUCCESS;
}
```

Figure 3: FLAME/C (left) and SuperMatrix (right) implementation of the blocked algorithm in Figure 2.

4.1 The FLAME/C API

FLAME is a methodology for deriving and implementing dense linear algebra operations [4, 15]. For this discussion it suffices to focus on the FLAME API for the C programming language, which allows the algorithm in Figure 2 to be represented in code as shown in Figure 3 (left).

4.2 FLASH

While several studies have been performed in the past on the benefits of storing matrices by blocks [9, 13, 20, 29], the fundamental problem with this approach is that the coding effort becomes very complex. Some researchers have tried to solve this via programming language solutions [36, 40] while others view matrices as higher dimensional arrays to capture the levels of blocking [1, 17].

We have observed that, conceptually, one naturally thinks of matrices stored by blocks as matrices of matrices. As a result, if the API encapsulates information that describes a matrix in an object, as FLAME does, and allows an element in a matrix to itself be a matrix object, then algorithms over matrices stored by blocks can be represented in code at the same high level of abstraction as the algorithm in Figure 3 (right). This layering may be instantiated recursively if multiple levels of hierarchy in the matrix are to be exposed. We call this extension to the FLAME API the FLASH API [26]. Examples of how simpler operations can be transformed from FLAME to FLASH implementations can be found in [7, 8].

4.3 SuperMatrix

Finally, we observe that given an API that views matrices as composed of unit blocks and an algorithm implemented using this API, the inner workings of the library can be changed so that instead of executing operations over blocks, a directed acyclic graph (DAG) is built that represents all tasks that need to be performed together with their dependencies. The DAG is then combined with a run-time system that dynamically schedules tasks for execution as dependencies are fulfilled. These two phases—constructing the DAG (*analyzer*) and scheduling the tasks (*scheduler/dispatcher*)—can take place transparently regardless of the algorithm used in the library routine.

In Figure 3 (right) we present the SuperMatrix implementation of the algorithm-by-blocks I. The call to `ENQUEUE_FLA_QR_accum_S_unb` enqueues a single task which will be responsible for both the factorization of the current column block and the construction of S_1. The call to `FLASH_QR_update_blk` decomposes itself into component tasks, which are then placed on a *task queue* corresponding to the operations identified in (2), (3), and (4). Once all tasks are enqueued, the DAG is complete, and a call to `FLASH_Queue_exec` initiates parallel execution. When a task completes execution, all dependent tasks that use blocks updated by the recently completed task are "notified". Once a notified task has all of its dependencies fulfilled, it is marked as ready and available and then enqueued at the tail of the *waiting queue*. Idle threads dequeue tasks from the head of this second queue until all tasks have been executed. We call this extension to FLASH the *SuperMatrix* run-time system since it allows out-of-order computation similar to machine instructions within superscalar architectures [19]. For further details on SuperMatrix, see [7, 8].

We used OpenMP to provide multithreading facilities where each thread executes asynchronously. We have also implemented SuperMatrix using the POSIX threads API to reach a broader range of platforms.

Approaches that hide complexity similar to SuperMatrix have been described for more irregular problems in the frame of the Cilk project [25] (for problems that can be easily formulated as divide-and-conquer, unlike the QR factorization) and also for general problems but with the specific target of the Cell processor in the CellSs project [3].

5 Experiments

In this section, we examine various multithreaded implementations for the QR factorization in order to assess the potential performance benefits offered by SuperMatrix out-of-order scheduling.

All experiments were performed on an SGI Altix 350 server using double-precision floating-point arithmetic. This ccNUMA architecture consists of eight nodes, each with two 1.5GHz Intel Itanium2 processors, providing a total of 16 CPUs and a peak performance of 96 GFLOPs/sec. (96×10^9 FLOPs per second). The nodes are connected via an SGI NUMAlink connection ring and collectively provide 32 GB (32×2^{30} bytes) of general-purpose physical RAM. The OpenMP implementation provided by the Intel C Compiler served as the underlying threading mechanism used by SuperMatrix. Performance was measured by linking to the BLAS in Intel's Math Kernel Library (MKL) version 8.1.

We report the performance (in GFLOPs/sec.) of the following four parallelizations of the QR factorization:

- **LAPACK `dgeqrf` + multithreaded MKL**. LAPACK 3.0 routine `DGEQRF` (QR factorization) linked to multithreaded BLAS in MKL 8.1.
- **Multithreaded MKL `dgeqrf`**. Multithreaded implementation of routine `DGEQRF` (QR factorization) in MKL 8.1.
- **AB-I + serial MKL + FLASH**. Our implementation of the algorithm-by-blocks I (AB-I), with matrices stored hierarchically using the FLASH API, scheduled with the SuperMatrix run-time system and linked to serial BLAS in MKL 8.1.
- **AB-II + serial MKL + FLASH**. Our implementation of the algorithm-by-blocks II (AB-II), with matrices stored hierarchically using the FLASH API, scheduled with the SuperMatrix run-time system and linked to serial BLAS in MKL 8.1.

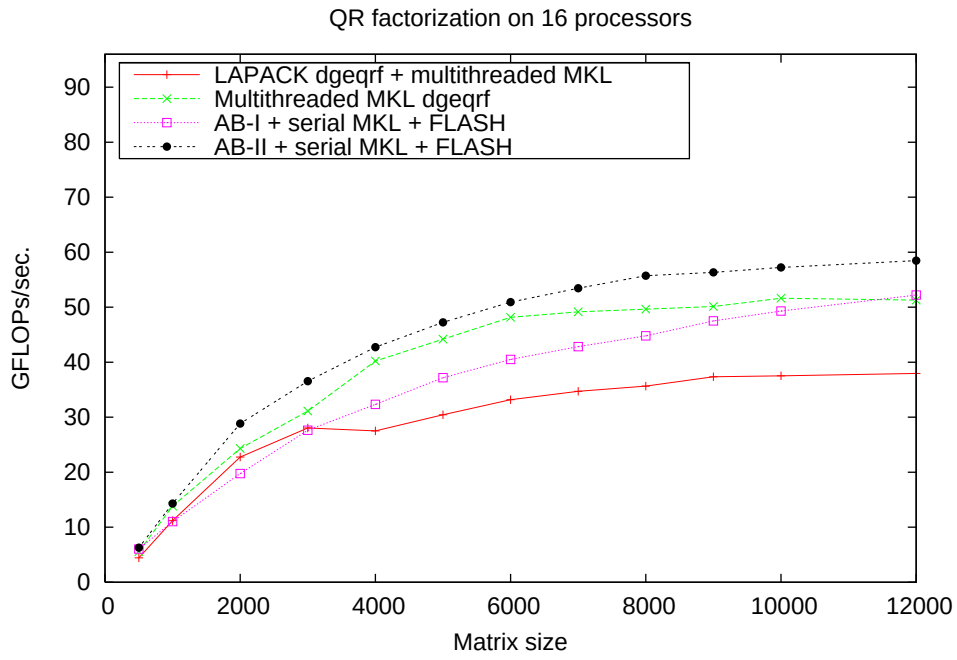

Figure 4: Performance of the QR factorization algorithms on 16 CPUs.

When hand-tuning block sizes, an effort was made to determine the best values of s and t for all combinations of parallel implementations and BLAS.

Performance results are reported in Figure 4. The matrix size ($m = n$) is reflected along the x-axis. The y-axis is scaled such that the top of the graph represents the theoretical peak performance of the system.

A few comments are due:

- AB-I outperforms the LAPACK implementation linked to MKL starting with problems of size 4000. The implementation of the QR factorization in MKL, however, attains higher performance than AB-I for all except the largest problem size.
- AB-II is a clear winner. The performance is close to the LAPACK implementation linked to MKL for small problems and much higher for problems of dimension larger than 2000. As the problem size approaches 12000, the performance of our two new algorithms appears to converge.

6 Conclusions

We have presented an experimental study that compares the traditional blocked algorithm for the QR factorization via Householder transformations to two new algorithms-by-blocks. The best algorithm, according to our preliminary study, combines the parallelism of the Givens rotation scheme for the QR factorization with a reduced cost and the level-3 BLAS performance of algorithms based on Householder transformations. The same ideas can be applied to yield two new algorithms-by-blocks for the LU factorization with pivoting of a matrix using the incremental pivoting scheme and the algorithms in [22, 31].

Clearly more parallelism could have been extracted from algorithm-by-blocks I by parallelizing the factorization that was left as an inherent sequential component. However, the success of algorithm-by-blocks II appears to make further study into this optimization moot.

The programming effort was greatly reduced by coding the algorithms with the FLASH extension of the FLAME/C API while the SuperMatrix run-time system—with its support for out-of-order scheduling via dynamic dependency analysis—facilitated easy and transparent algorithm-level parallelization of the QR

factorization. The results demonstrate that high performance can be attained by programs coded at a high level of abstraction on SMP and, likely, future many-threaded architectures.

The SuperMatrix methodology, for algorithm-by-blocks II in particular, should be of interest to those working to program the IBM Cell processor and similar accelerators. On such architectures, the run-time system only needs to be changed to dispatch computations along with blocks to the compute cores of such processors.

Acknowledgments

We thank the other members of the FLAME team for their support.

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] R. C. Agarwal and F. G. Gustavson. Vector and parallel algorithms for Cholesky factorization on IBM 3090. In *SC '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 225–233, New York, NY, USA, 1989.

[2] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 1992.

[3] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a programming model for the Cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 5–15, Tampa, FL, USA, November 2006.

[4] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[5] Paolo Bientinesi, Brian Gunter, and Robert van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software*, 2007. Conditionally accepted.

[6] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM Journal on Scientific and Statistical Computing*, 8(1):2–13, January 1987.

[7] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 2007.

[8] Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Satisfying your dependencies with SuperMatrix. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, Austin, TX, USA, September 2007.

[9] S. Chatterjee, A.R. Lebeck, P.K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 13(11):1105–1123, 2002.

[10] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[11] 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, March 1988.

[12] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadelphia, PA, USA, 1991.

[13] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[14] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, USA, 3rd edition, 1996.

[15] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[16] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, March 2005.

[17] F. G. Gustavson, L. Karlsson, and B. Kagstrom. Three algorithms on distributed memory using packed storage. B. Kagstrom, E. Elmroth, editors, *Computational Science – PARA '06, Lecture Notes in Computer Science*. Springer-Verlag, 2007. To appear.

[18] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. *SIAM Journal on Scientific and Statistical Computing*, 15(5):1201–1226, 1994.

[19] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 3rd edition, 2003.

[20] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, February 1992.

[21] Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field G. Van Zee. Accumulating Householder transformations, revisited. *ACM Transactions on Mathematical Software*, 32(2):169–179, 2006.

[22] Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. *PARA '04, Lecture Notes in Computer Science*, 3732:413–422. Springer-Verlag, 2005.

[23] James Kahle, Michael Day, Peter Hofstee, Charles Johns, Theodore Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, September 2005.

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

[25] Charles Leiserson and Aske Plaat. Programming parallel applications in Cilk. *SINEWS: SIAM News*, 31, 1998.

[26] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. Technical Report TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.

[27] F.T. Luk. A rotation method for computing the QR decomposition. *SIAM Journal on Scientific and Statistical Computing*, 7:452–459, 1986.

[28] Mercedes Marqués, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Specialized spectral division algorithms for generalized eigenproblems via the matrix disk function. *PARA '06, Lecture Notes in Computer Science*. Springer-Verlag, 2007. To appear.

[29] N. Park, B. Hong, and V.K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[30] Chiara Puglisi. Modification of the Householder method based on the compact WY representation. *SIAM Journal on Scientific and Statistical Computing*, 13(3):723–726, 1992.

[31] Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 2007. Conditionally accepted.

[32] 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, January 1989.

[33] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[34] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. James Abello and Jeffrey Scott Vitter, editors, *External Memory Algorithms and Visualization*, pages 161–180. American Mathematical Society Press, Providence, RI, USA, 1999.

[35] Sivan Toledo and Eran Rabani. Very large electronic structure calculations using an out-of-core filter diagonalization method. *Journal of Computational Physics*, 180:259–269, 2002.

[36] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–840, 2002.

[37] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[38] Homer F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 9(1):152–163, 1988.

[39] David Watkins. *Fundamentals of Matrix Computations*. John Wiley and Sons, 2nd edition, 2002.

[40] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for morton-order matrices. In *PPoPP '01: Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 24–33, New York, NY, USA, 2001.