

Scaling LAPACK Panel Operations Using Parallel Cache Assignment

Anthony M. Castaldo

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
castaldo@cs.utsa.edu

R. Clint Whaley

Department of Computer Science
University of Texas at San Antonio
San Antonio, TX 78249
whaley@cs.utsa.edu

Abstract

In LAPACK many matrix operations are cast as block algorithms which iteratively process a panel using an unblocked algorithm and then update a remainder matrix using the high performance Level 3 BLAS. The Level 3 BLAS have excellent weak scaling, but panel processing tends to be bus bound, and thus scales with bus speed rather than the number of processors (p). Amdahl's law therefore ensures that as p grows, the panel computation will become the dominant cost of these LAPACK routines. Our contribution is a novel parallel cache assignment approach which we show scales well with p . We apply this general approach to the QR and LU panel factorizations on two commodity 8-core platforms with very different cache structures, and demonstrate superlinear panel factorization speedups on both machines. Other approaches to this problem demand complicated reformulations of the computational approach, new kernels to be tuned, new mathematics, an inflation of the high-order flop count, and do not perform as well. By demonstrating a straight-forward alternative that avoids all of these contortions and scales with p , we address a critical stumbling block for dense linear algebra in the age of massive parallelism.

Categories and Subject Descriptors G.4 [Mathematical Software]: Parallel and vector implementations

General Terms Performance, Experimentation, Design, Algorithms.

Keywords ATLAS, LAPACK, QR, LU, factorization, parallel, multicore, multi-core, GPU

1. Overview

The parallelization technique we describe in this paper is quite general, even though our chosen demonstration cases involve a particular set of routines in dense linear algebra (where it solves a long-standing problem). The central lesson for parallel practitioners in general is that, given a bus-bound operation with the potential for

memory reuse, and a system with hardware-enforced cache coherence, it can be highly profitable to block for cache reuse *even if this introduces parallel overheads into the innermost loop*. In this paper, we demonstrate superlinear speedup for an operation that has resisted significant parallel speedup for years; we accomplish this feat by enabling cache reuse at the cost of increasing the number of parallel synchronization points by more than two orders of magnitude. This approach leads to superlinear speedups because it changes the computational speed from the speed of memory to the speed of the cache, and the additional parallel overhead only slightly mediates this performance bonanza, because the additional parallel synchronizations can be based on cache coherence, and thus run at the speed of hardware.

1.1 Basic Terminology

LAPACK [2, 15] (Linear Algebra PACKage) is one of the most widely-used computational APIs in the world. Since linear algebra is computationally intensive, it is important that these operations, which are inherently very optimizable, run as near to machine peak as possible. In order to allow for very high performance with a minimum of LAPACK-level tuning, LAPACK does the majority of its computation by calling a lower-level API, the BLAS (Basic Linear Algebra Subprograms). The BLAS are in turn split into three “levels” based on how much cache reuse they enjoy, and thus how computationally efficient they can be made to be. In order of efficiency, the BLAS levels are: Level 3 BLAS [11], which involve matrix-matrix operations that can run near machine peak, Level 2 BLAS [9, 10] which involve matrix-vector operations and Level 1 BLAS [18, 19], which involve vector-vector operations. The Level 1 and 2 BLAS have the same order of memory references as floating point operations (FLOPS), and so will run at roughly the speed of memory for out-of-cache operation.

2. Introduction

In LAPACK, many matrix operations are cast as block algorithms which iteratively process a panel using an unblocked Level 2 BLAS-based algorithm and then update a trailing matrix using the Level 3 BLAS. Level 2 BLAS perform $O(N^2)$ flops on $O(N^2)$ data, making them largely bus-bound. On the other hand, Level 3 BLAS routines perform $O(N^3)$ flops on $O(N^2)$ data, allowing for extensive cache and register reuse. This strongly reduces their dependence on bus speed, so that the Level 3 BLAS can often be tuned to run near the peak flop rate of the machine (i.e. the Level 3 BLAS tend to be compute bound rather than bus bound). LAPACK QR and LU factorizations exemplify this general BLAS/blocking strategy, and for large problems running in serial on our test systems, this tuned LAPACK approach results in QR and LU spending over

¹This work was supported in part by National Science Foundation CRI grants CNS-0551504 and CCF-0833203.

95% of their runtime in the Level 3 BLAS, resulting in extremely high performance.

However, as commodity platforms trend increasingly to multi-core systems that offer high degrees of parallelization, this approach is faltering. Level 3 BLAS routines typically scale well with p (the number of cores) but the bus bound Level 2 BLAS routines do not; once the bus is saturated they cannot get any faster, and bus speeds are not growing as fast as p . Therefore Amdahl’s law tells us that as p grows, the Level 2 BLAS-based panel factorizations will increasingly dominate the runtime. We can see this happening already on current architectures: On our 8-core test systems, QR panel factorization is only 4% of the runtime on a large problem when a single core is used, but over 20% of the runtime when the Level 3 BLAS routines are parallelized. This is going to be the case any time massive parallelization is employed; for example when the factorization is done on a GPU, the GPU typically relies upon the CPU to perform the panel factorizations[26, 27], which are typically done at bus speed, thus creating a serious bottleneck.

Clearly there is a need to further parallelize the panel factorization. Prior work[12, 17, 25] has shown that recursion can help do that by employing the Level 3 BLAS within the panel factorization. However as the recursion deepens and the panel thins, the Level 3 BLAS get less and less benefit from parallelism, cache and register blocking, and are ultimately bus bound again. This is the problem we address with our new approach, which we call Parallel Cache Assignment (PCA).

2.1 Our Approach: Parallel Cache Assignment (PCA)

The key to PCA is a data-ownership model of parallelism where data is assigned to the cores in order to enable cache reuse. One side effect of this assignment is that we introduce parallel synchronization *into* the innermost loop in order to maximize cache reuse: since without this transformation the performance is limited to the speed of memory, this counter-intuitive optimization turns out to more than pay for itself.

The central insight here is that a QR or LU panel factorization takes $O(N \cdot N_b^2)$ flops for $N \cdot N_b$ data, and thus, with $O(N_b)$ flops per data element, there is the potential of cache reuse. In other words we *should* be able to find some way to escape the bus bound limitation of the Level 2 BLAS routines being used. However, the reuse is tied to the *outer* loop that indexes the columns; by this we mean that if we split that loop into p equal parts we reduce the cache reuse by a factor of p . Instead, we break the *inner* loop that indexes the rows (in addition to enabling greater cache reuse, this allows for parallelism when operating on the very long columns seen in these panel operations). In short, we partition the panel horizontally, assigning an equal number of rows to each core, and have them *all* loop over the N_b columns of the panel. Then, each core can keep its share of the panel in its own cache (we target the L2 cache here), and loop over all N_b columns. However, this creates a new constraint: If we want to preserve the current factorization arithmetic, both QR and LU must proceed one full column at a time, from left to right. Thus all the cores cooperate and synchronize on every column, and on the subsequent trailing matrix updates within the panel.

In practice every core processes their share of the column, and at the necessary sync points we perform a $\log_2(p)$ combine of their results. Once that is complete the cores use the global result to proceed independently to the next sync point. LU requires only one synchronization point; QR requires several.

Note that we must introduce parallel synchronizations into the innermost loop. Driving the parallel synchronization into the innermost loop is counter-intuitive as it will obviously increase parallel overhead, and if it required expensive communications it would represent an intolerable drag on performance. However, on multi-

core systems with shared memory, we can produce a lightweight and efficient memory-based sync scheme (see 4.1), and in this multi-core environment we found the cost of frequent synchronization is far outweighed by the savings we gain from cache reuse and finer-grained parallelism. The cores continue to use the serial Level 2 BLAS routines, but instead of being bus bound from memory, they can run at L2 cache speeds.

The panel can be too large to fit into even the aggregate caches of all cores, in which case we employ column-based recursion[12, 17, 25] until the problem fits in cache. For extremely large problems, even one column could overflow the cache; in such cases we end the recursion when the number of columns grows too small for the recursion’s Level 3 BLAS usage to extract meaningful parallelism. Normally, each core performs a data copy of its share of the panel into local storage at the beginning of the panel factorization, and copies the computed result back out at the end. This allows us to optimize memory alignment, minimize TLB usage, false sharing, and cache line conflict misses. Due to its more effective use of the memory hierarchy, this copy operation usually improves performance significantly. However, if the number of rows causes recursion to severely restrict the panel width, and/or the panel overflows the aggregate cache, we get better performance by omitting the copy steps. The crossover point between copy and no copy can be empirically tuned for improved performance.

We will show superlinear speedup in both QR and LU panel factorizations on 8-core systems (e.g. 16-fold speedups over the corresponding single-core performance). We also show speedups of 2–3 times the fastest known parallel panel factorizations, and we show our approach scales well with p . As p continues to grow, aggregate cache size is likely to increase as well, so ever larger problems can be made to run at cache speed rather than memory speed.

Other work done in this area includes[5, 7, 16]. However, some of this work significantly increase the number of flops beyond the LAPACK blocked algorithm, while our approach requires no extra flops at all. Our approach also has the advantage of making no changes to the arithmetic of the LAPACK blocked algorithms, and is thus guaranteed to preserve both its stability and the error bound¹.

We believe PCA is a general approach for all or most LAPACK panel operations, and we have implemented it as described for both QR and LU factorizations. For reasons of space we will concentrate primarily on the QR factorization, which is the more difficult case, but we will show results for LU as well.

2.2 Surveyed Libraries and Important Routines

Both the LAPACK and BLAS APIs have a reference implementation available on netlib, but many vendors and academic groups provide optimized alternatives. In general, the reference BLAS (particularly the Level 3 BLAS) are orders of magnitude slower than optimized BLAS. Optimized LAPACK implementations can show notable improvements, but this is not nearly as marked as for the Level 3 BLAS.

In this paper we discuss two optimized LAPACK and BLAS frameworks. ATLAS [30, 31, 33] (Automatically Tuned Linear Algebra Software) is our package which uses empirical tuning to auto-adapt the entire BLAS and a subset of LAPACK to arbitrary cache-based systems. ATLAS uses the netlib reference LAPACK to provide those LAPACK routines that are not currently supported. The GotoBLAS [13, 14] provides meticulously hand-tuned implementations (written mostly in carefully crafted assembly) for many current machines, with performance that is almost always near or at the best known for the machine. The GotoBLAS provide a complete BLAS, and a subset of LAPACK.

¹ Due to parallel blocking, the error bound constant is reduced.

We also compare against a new linear algebra framework, which is planned as a complete rewrite of LAPACK in order to exploit massive parallelism. This new research project is called **PLASMA**[1] (Parallel Linear Algebra for Scalable Multi-core Architectures), and currently implements new, explicitly parallel, approaches to a subset of LAPACK.

In this paper we will often refer to an algorithm by its LAPACK or BLAS name. For both APIs, the first character provides the type or precision the routine operates on; for this paper all routines will start with **D**, indicating they operate on double precision real data. New names will be introduced as needed, but there are a few of enough importance to introduce here. DGEQRF is the LAPACK routine which performs the QR factorization (see §3) on an entire matrix. In LAPACK this algorithm is parallelized implicitly by the Level 3 BLAS, which also provide the foundation for its serial performance. The performance of DGEQRF rests primarily on three routines. In order of typical importance these are DGEMM, DTRMM and DGEQR2. DGEMM and DTRMM are Level 3 BLAS routines handling general rectangular and triangular matrix multiplication, respectively. The third such routine, DGEQR2, is an LAPACK sub-program called at each step of DGEQRF's iteration to factor a column panel (i.e. a block of columns). DGEQR2 in turn primarily relies on the performance of two Level-2 BLAS routines for its performance; DGER performs a rank-1 update (update a matrix using the outer product of two vectors), and DGEMV performs a matrix-vector product. In DGEQR2, DGEMV is always called with a transposed matrix, which we will denote as DGEMV^T .

2.3 Outline

In Section 3 we review the routines involved in the QR panel factorization, and §4 provides a detailed description of our technique and its application to QR. §5 discusses our timing methodology. In §6 we provide a quantitative comparison of this technique on two representative commodity architectures; show they have excellent scaling with p and produce significant measured speedup versus ATLAS, the GotoBLAS [13, 14], and the PLASMA library (which implements QR as described in [5]). Finally, in §8 we discuss future work, and offer our summary and conclusions in §9.

3. QR Matrix Factorization

A QR factorization of $A \in \mathbb{R}^{M \times N}$ with $M \geq N$ yields

$$A = Q \cdot R = [Q_1 \quad Q_2] \cdot \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 \cdot R_1 \quad (1)$$

Where $Q \in \mathbb{R}^{M \times M}$ and $R_1 \in \mathbb{R}^{N \times N}$ is upper triangular. There are multiple ways to accomplish this, for examples see [23]. In LAPACK, Q is found as a product of Householder transforms, one per column. For our example we need N . Each transform is of the form $H_i = I - \tau_i \cdot v_i \cdot v_i^T$, where v_i has a unit-2 norm. Note that Householder transforms are *involuntary*, i.e. $H_i = H_i^T$ and $H_i \cdot H_i = I$ (H_i is its own inverse). Each H_i is computed in order to zero the elements beneath the diagonal of A on column i , thus

$$(H_n \cdots H_1) \cdot A = R, \Rightarrow$$

$$A = (H_1^{-1} \cdot H_2^{-1} \cdots H_N^{-1}) \cdot R, \Rightarrow$$

$$Q = H_1 \cdot H_2 \cdots H_N.$$

Q is orthogonal, so $Q^{-1} = Q^T$, i.e. inversion has no loss of precision. This is useful in solving systems of linear equations $A \cdot X = B$, QR produces the result $R \cdot X = Q^T \cdot B$, which is then solved to high precision using back substitution.

3.1 The LAPACK Implementation

We address the LAPACK formulation of a blocked QR factorization, as coded in DGEQRF, which relies heavily on [23] in forming the compact WY representation which saves storage space (versus the WY representation of [3]). The compact WY representation of Q is $Q = (I - Y \cdot T \cdot Y^T)$, where the columns of Y represent the vectors v computed for each Householder transform, and T is upper triangular.

Consider factoring a double precision matrix $A \in \mathbb{R}^{M \times N}$ using a blocking factor of N_b . This is done iteratively; we first factor the column panel $A[1..M, 1..N_b]$ using the routine DGEQR2. Thus at the end of the panel factorization, Y is $M \times N_b$, and because the vector length decreases by one for each column, it is lower trapezoidal and stored beneath the diagonal of the original A matrix².

Figure 1 illustrates the basic operation. The panel is factored by the routine DGEQR2 into the upper triangular R_p and lower trapezoidal Y_p . Then the routine DLARFT uses Y_p to compute T in a work area that is $N_b \times N_b$ elements, and finally, DLARFB is used to compute Q times the trailing matrix portion of A , using the identity $Q = (I - Y_p \cdot T \cdot Y_p^T)$, producing R_t (the upper portion of the final R) and A_t (gray area, and the new matrix to factor). The dotted lines indicate the next iteration of the algorithm (building T is not necessary on the final iteration).

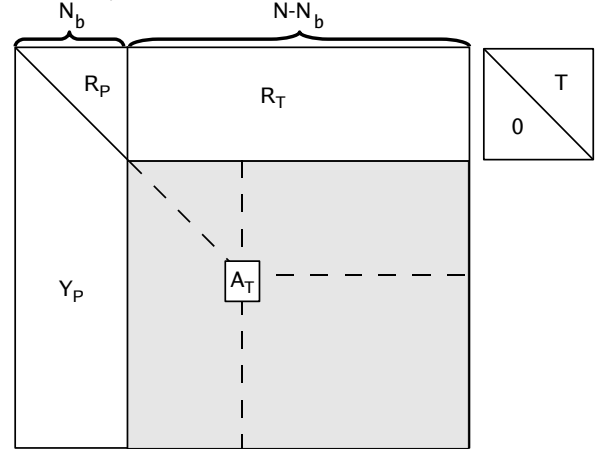


Figure 1. LAPACK Blocked QR Factorization

4. Applying PCA to QR Panel Factorization

To describe our work we need to further detail the implementation. In the default ATLAS multi-core implementation of this algorithm, the DGEQR2 and DLARFT operations are performed by a single core (these routines use the Level 1 and 2 BLAS, which are not parallelized in ATLAS). DGEQR2 consists of Level 1 and Level 2 BLAS routines, and as discussed these are bus-bound and thus often scale poorly with p even when they are threaded (threading these routines on some platforms can actually cause slowdowns due to bus contention). The DLARFB operation uses two Level 3 BLAS routines, DTRMM and DGEMM, which have parallel implementations, which we did not change. Figure 2 provides an outline of the DGEQR2 process, which is described more fully in the following paragraphs.

PCA uses a data ownership model of parallelism along the rows; so in Figure 1 we divide the column panel (the first N_b columns of

²Unlike the textbook QR factorization, LAPACK forces v to have 1.0 as its first element; thus this element is not stored. R is stored on and above the diagonal of A , thus the factored A replaces the original A .

```

DGEQR2
  Loop over Nb columns of A
  {
    DLARFG/P    (See Figure Note below)
    DNRM2*     (compute norm of column)
    DSCAL      (scale column)
    DLARF      (update trailing matrix)
    DGEMVT*    (build update vector W, uses Y)
    DGER       (Rank-1 update remaining cols)
  }

DLARFT
  Loop i over Nb columns of Y
  {
    DGEMVT*    (compute W=Y[1..i-1] x Y[i])
    DTRMV      (Core 0, T[i]=T[1..i-1] x W)
  }

```

Note: We parallelized DLARFG as found in LAPACK 3.1.1; DLARFG was replaced by DLARFP in LAPACK 3.2, but with no impact on either the parallelization or performance. See [8].

Figure 2. DGEQR2, DLARFT Operational Outline

A, shown as R_p and Y_p) horizontally, into a stack of p equal-height blocks, one block per core. $p = 8$ in our case, and we actually assign cores multiples of 8 rows at a time (because 8 doubles fills one cache line). Any “leftover” rows are assigned to the top block, on the theory it has less work to do by virtue of owning the triangular portion of Y_p . There are three operations that require the full panel height to produce a global result, and two additional operations on the full panel height that are parallelized. We will take each of these five operations in turn, referring to Figure 2. All the cores loop over all N_b columns. The asterisks in Figure 2 indicate synchronization points within the loop.

1. **DNRM2***: To compute the Householder vector we need the 2–norm of the full column. The 2–norm requires the square root of the sum of the squares. This is actually complicated in LAPACK, because it is coded to preserve precision and avoid unnecessary overflow. The idea is simple: as it traverses the vector it separately keeps track of the largest magnitude number found thus far and scales all numbers to it; i.e. before squaring, it multiplies new numbers by the inverse of the scaling number, and *then* adds the result to the sum of squares. However, occasionally a *new* largest magnitude must become the new scale, so it must rescale the sum-so-far to the new scale. When complete, DNRM2 can return $(\text{scale} \cdot \sqrt{\text{sum_of_squares}})$ as the final answer, thus avoiding the possible overflow of having ever squared the largest magnitude number, and also the potential loss of precision created by summing smaller squares to the largest squared number.

To parallelize this and preserve the behavior, each core must return for its portion of the column both the scale it found and the sum of squares. We combine these two at a time in a $\log_2(p)$ operation, using the obvious arithmetic³. Core 0 then produces the final answer and signals the other cores that it is available.

2. **DSCAL**: All cores do the math of producing the Householder scalar values for TAU, Beta, etc. “Problem” vectors must be rescaled and the 2–norm recomputed; we do this as well, looping

³ Given $scale_1$, sum_1 , $scale_2$ and sum_2 , and presuming $scale_1 > scale_2$, the proper combined sum of squares would be $scale_1^2 \cdot sum_1 + scale_2^2 \cdot sum_2$, but we want $scale_1^2$ factored out, and we want to avoid computing the squares of either $scale_1$ or $scale_2$. Thus we compute the pair $\left[scale_1, \left(sum_1 + \left(\frac{scale_2}{scale_1} \right)^2 \cdot sum_2 \right) \right]$.

back to the DNRM2* step as necessary. Note the decision to rescale and recompute is based on the final global value, so if one core rescales, they will all make the same decision. Problem vectors are extremely rare, normally each core proceeds to scaling its portion of the column and then immediately to the next step. This scaling is actually producing the Householder vector v_i (for column i); the Householder matrix is $H_i = I - \tau_i \cdot v_i \cdot v_i^T$.

3. **DGEMV^T***: Having computed the Householder vector v_i , we must now update the trailing matrix of the panel by multiplying it by H_i . In Figure 1 imagine Y_p is divided vertically on some column i ; and we will call Y_L the columns $1 \dots i$ and Y_R the columns $(i + 1) \dots N_b$. We need a new Y_R , which we will call Y'_R . We compute that as $Y'_R = H_i \cdot Y_R$.

Replacing H_i with $(I - \tau_i \cdot v_i \cdot v_i^T)$, we must compute $Y'_R = (I - \tau_i \cdot v_i \cdot v_i^T)(Y_R)$, and this expands to $Y'_R = Y_R - \tau_i \cdot v_i \cdot (v_i^T \cdot Y_R)$. Working from the right we can first compute the vector $W = v_i^T \cdot Y_R$. It is easier to compute W^T , and since W is a vector this requires no actual transpose operation. So this step will compute $W^T = (Y_R)^T \cdot v_i$.

Because we divide Y_{PR} on rows, our cores each compute a vector that is $(N_b - i)$ elements long; and we add them in a $\log_2(p)$ combine. This is a sync point because in the next step, all cores will need the combined global W vector to complete this operation.

4. **DGER**: This is the second step of applying the Householder matrix. In the previous step we computed $W = v_i^T \cdot Y_R$, now we complete the application of H_i to Y_R . By substitution we have $Y'_R = Y_R - \tau_i \cdot v_i \cdot W$, which we recognize as a simple rank-1 update of Y_R . DGER accomplishes that. No synchronization is necessary after DGER, so once each core completes its DGER it proceeds to the next column and begins work on its DNRM2*. After the last column, they can proceed to the DLARFT computation (or the threads can exit if DLARFT is not needed).
5. **DLARFT, DGEMV^T***: After completing DGEQR2, we must build the T matrix with a call to DLARFT, where we primarily parallelize the work in DGEMV^T*. The DGEMV^T* portion of this is quite similar to the previous DGEMV^T* of DLARF. We are using the Schreiber/van Loan[23] method of computing T , so we only update one column at a time. But unlike the previous DGEMV^T*, this time we are using the *left* portion of the Y_p matrix at each column, and computing the vector $Y_{pL}^T \cdot v$.

All cores compute their share of this vector and we sum them ($\log_2(p)$) to produce a final vector. Core 0 alone completes the update of T with a DTRMV. We require synchronization because the computation of the new T must be completed before the core can begin the next DGEMV^T* (so technically Core 0 does not signal completion until it completes the DTRMV). Currently this DTRMV is too small ($N_b \times N_b$) to gain any performance from parallelization, and is done by Core 0 alone, but if future N_b grow large enough then parallelizing this step should be straightforward.

To minimize synchronization overhead we use a memory-based synchronization that relies on hardware-enforced cache coherence instead of OS calls. We also tweaked the ATLAS tuning framework to tune DGEMV and DGER for the L2-cache, and provided vectorized kernels to the framework (when these kernels were better than the existing kernels, both the parallel and serial timings reported in this paper used the improved kernels). Cache-specific kernel tuning is an area we hope to explore in future work. We will now outline our approach for lightweight shared-memory synchronization, and

then present timings to demonstrate the benefits of PCA for both LU and QR.

4.1 Implementation Note: Lightweight Memory Synchronization

Our PCA algorithm depends on low-overhead synchronization, which we can implement cheaply on cache-coherent systems through memory (therefore running at the speed of hardware, rather than the OS, as in the case of condition variables). Since all participating cores are dedicated to the panel factorization, we can use spinlocks without loss of performance. Because these memory synchronizations are prone to subtle programming bugs, we outline the approach we use here (this approach is well known in several areas and we are not claiming any originality; we discuss it here for implementors who may be unaware of the details of this long-standing technique).

In our code we assign one thread per core and the thread ranks start at zero and ascend. Each thread knows its rank, and the rank is used to index two critical arrays (declared as `volatile` to avoid register assignment). Each array is P_m long, where P_m is the maximum number of threads that can be used. The first such array (`active`) is a simple boolean array; before the master program launches any threads, it sets `active[rank]` to true if the thread of rank `rank` will be participating in the panel factorization; otherwise it is set to false. This allows active threads to check which partners are participating in the operation; this information will be needed when local information is combined to produce global results.

The `sync[]` array has one integer per thread, and the master program initializes all of these to -1 . In our panel operations two types of synchronization are necessary; one for the \log_2 combine and the other to signal that a global result is finished and the threads can use it to proceed. All our synchronizations have the effect of a barrier, so the first part of the idea here is that when a thread finishes its work up to a sync point, it signals interested partner threads by incrementing its sync array entry (`sync[i_am]++`).

If a thread T_i is responsible for combining a result, say T_4 must wait on T_5 to finish, then T_4 must wait until `sync[5] > sync[4]`. We do this by spinning on memory; e.g. if `i_am=4` and `him=5`:

```
if (active[him]) while (sync[i_am] >= sync[him]);
```

When this loop exits, the fact that T_5 incremented `sync[5]` implies it is done with its work. After T_4 has done its combine (more generally, all of the combines for which it is responsible) then it increments `sync[4]`.

To wait on a global result, after each thread completes its work and has incremented its own sync array entry, it can wait for `sync[0]` to match its own entry; e.g. `while (sync[0] < sync[i_am]);`, assuming rank 0 is the destination of the combine.

An easy example is the LU pivot operation. Each thread must compute a local maximum, and these must be combined to produce a global maximum, the row that holds that value is exchanged with the active row. T_0 does the final combine to discover the global maximum, and then performs the exchange of the rows (which isn't worth parallelizing). Once that exchange is complete, T_0 increments `sync[0]`, which allows all the other threads to proceed.

This general technique is easily used in any \log_2 (or linear) combine operation.

5. Experimental Methodology

When performing serial optimization on a kernel with no system calls, researchers often report the best achieved performance over several trials [29]. This will not work for parallel times: parallel times are strongly affected by system states during the call and vary widely based on unknown starting conditions. If we select the best

result out of a large pool of results, we cannot distinguish between an algorithm that achieves the optimal startup time by chance once in a thousand trials from one that achieves it every time by design. An average of many samples will make that distinction.

In our timings, the sample count varies to keep experiment runtimes reasonable: for all panel timings, we take the average of 100 trials; for full square problems, we use 50 trials to factor matrices under 4000×4000 elements, and at least 20 trials for larger matrices. We flush all processors' caches between timing invocations, as described in [29].

The libraries used were ATLAS (v-3.9.11), GotoBLAS (v-r1.26), PLASMA (v-1.0.0), and LAPACK (v-3.1.1). All timings used the ATLAS timers and ATLAS's LAPACK autotuner[28] was used to empirically tune the LAPACK blocking parameter to the GotoBLAS, and to autotune both blocking factors used in PLASMA. Goto uses the LAPACK outer routines (DGEQRF, DGEQR2, etc); PLASMA uses its own versions of the outer routines and the ATLAS *serial* BLAS. ATLAS uses C equivalents of the LAPACK routines or a newly written DGEQRR and DGEQR2 that are drop-in replacements for DGEQR2. Both ATLAS and Goto libs are tuned separately for each test platform to provide high performance.

Unlike ATLAS, the GotoBLAS parallelize DGEV and DGER and thus provide us with a ready-made implementation of the traditional route to panel factorization parallelization. Goto, ATLAS and PLASMA, properly tuned, are the state-of-the-art for comparison purposes.

The ATLAS BLAS create and destroy threads for each BLAS call, using the master last algorithm [6]. Our PCA factorizations creates and joins p threads for each call to the panel factorization (in addition to the threads that the Level 3 BLAS call will create). Both PLASMA and the GotoBLAS use pthreads and affinity, and both employ "persistent" threads⁴.

We timed on two commodity platforms, both of which have 8 cores in two physical packages. We use Linux, but even different minor releases of Linux can have scheduler differences that change timing significantly, so we provide kernel version information here. Our two platforms were:

(1) **Opt8, O8:** 2.1Ghz AMD Opteron 2352 running Fedora 8 Linux 2.6.25.14-69 and gcc 4.2.1, GotoBLAS r1.26, PLASMA 1.0.0, LAPACK 3.1.1 and ATLAS 3.9.11 (on this platform, we did not use ATLAS's architectural defaults, since the full ATLAS search yielded noticeably faster BLAS),

(2) **Core2, C2:** 2.5Ghz Intel E5420 Core2 Xeon running Fedora 9 Linux 2.6.25.11-97 and gcc 4.3.0, ATLAS 3.9.11, Goto r1.26, PLASMA 1.0.0, LAPACK 3.1.1 and ATLAS 3.9.11.

Each physical package on the Opt8 consists of one chip, which has a built-in memory controller. The physical packages of the Core2 contain two chips, and all cores share an off-chip memory controller.

ATLAS tunes GEMM (the Level 3 BLAS routine providing **GE**neral **M**atrix **M**ultiply) differently on these two platforms. On the Core2 platform the ATLAS blocking factor is 56, and on the Opt8 we use 72. Our best performing QR blocking factors will turn out to be multiples of these when ATLAS is performing the GEMM.

To compare our panel factorization performance to the Goto and PLASMA libraries, we choose a panel width of 128. This is not ideal for ATLAS or PLASMA. The GotoBLAS does well with multiples of 64. The reason for choosing 128 is to make a fair head-to-head comparison of the methods with a realistic panel width that

⁴ Technically, the GotoBLAS delay thread exits for a fraction of a millisecond, so it can reuse threads if the BLAS is called again quickly enough. In the context of our factorizations this is always true so this is the equivalent of creating persistent worker threads.

doesn't explicitly favor our PCA or the ATLAS-tuned BLAS we are using. PLASMA does not possess a routine specialized for panel factorization, but since panel factorization is merely a factorization of a heavily non-square matrix, we report its best performance on that task.

6. Impact of these techniques on runtime

The measurements that interest us most are

- **Panel Speedup:** How much faster our panel factorization is than the alternatives, such as serial panel factorization or parallelizing using the parallel Level 2 BLAS routines (we show superlinear speedups versus serial asymptotically);
- **Overall Speedup:** The impact of faster panel factorization on the overall QR performance (we show up to 35% speedup over ATLAS/LAPACK's previous method).
- **Scaling:** Whether this panel factorization speedup can be expected to scale with p (we show excellent weak scaling);

The blocked version of QR requires more flops than the unblocked version. On square problems, the extra flops required range from about 9% for 1000, declining quickly to about 1.5% for 8000. Note, however, that all MFLOP rates reported in this paper always use the unblocked flop count, as discussed below.

In measuring panel performance, we note that in the LAPACK blocked algorithm DGEQRF, DGEQR2 is followed by DLARFT, the two routines we have parallelized. When we report Panel MFLOPS, we are reporting the MFLOP rate achieved on both of these operations, for just the first panel of the factorization. We report the rate based on the *effective* flops, meaning we count the *time* of both DGEQR2 and DLARFT, but do not count any flops consumed by DLARFT. The PLASMA approach does not provide separate routines for a DGEQR2 and DLARFT; thus we time how long it takes to factor a panel. Although technically this is only the DGEQR2 equivalent time, we believe PLASMA will require extra flops roughly equal to those consumed by the canonical DLARFT.

For panel timings all of the panels are 128 columns wide (i.e. $N_b = 128$) and we use the average of 100 trials per problem size. We use DGEQR2's flop count for all panel operations, which is given in equation 2 for an $M \times N_b$ factorization, where $M \geq N_b$ (this equation comes from LAPACK's `dopla.f`).

$$fc = \frac{1}{6} \cdot (12 \cdot M \cdot N_b^2 + 12 \cdot M \cdot N_b + 6 \cdot N_b^2 + 28 \cdot N_b - 4 \cdot N_b^3) \quad (2)$$

The most critical result is the speedup of our approach over the best known algorithm (Figures 3(a) and (c)), and over the original LAPACK serial algorithm (Figures 3(b) and (d)).

In Figure 3 the X-axis is always the panel height (the panel width is 128 elements); and the Y-axis is the speedup we obtained. Thus in Figure 3(d), our QR PCA factorization of a 13000×128 element panel is 15 times faster than the serial version on the Core-2 platform. Both platforms have 8 cores, so notice we achieve superlinear speedup over serial for both QR and LU on both test platforms. The superlinear speedups are due to cache reuse: when the problem exceeds one core's cache, the operations run at memory speed, but when the panel is cache-contained, it can run roughly at cache speed. Our approach uses all caches in parallel, and when even this roughly p -fold increase in cache space is not enough to contain the panel, we recur until the problem can be cache partitioned. For best performance, it is necessary to discover empirically how much cache space can be effectively used by a core (remember that caches are typically combined instruction/data, sometimes shared amongst processors, and often use non-LRU replacement, all of which tend to lower the effective cache size). For the Core2

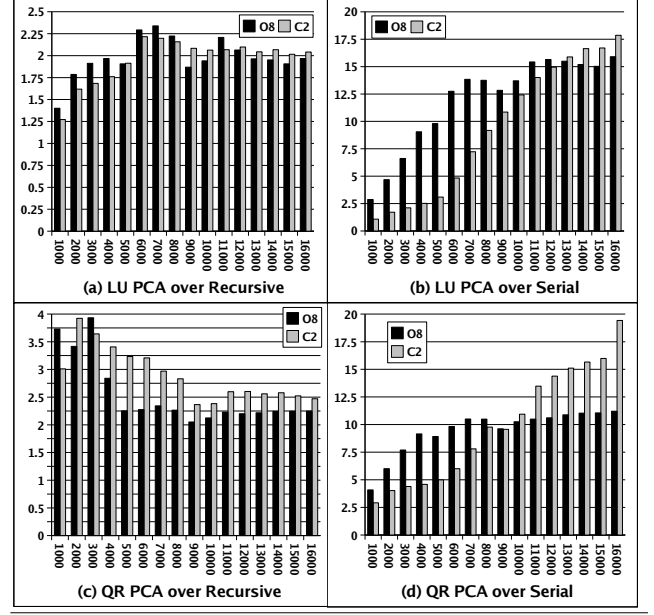


Figure 3. ATLAS QR and LU Panel Speedups

(Opt8) we found a cache threshold of 1.2MB (512KB). The asymptotic speedup over serial for QR panel factorization on the Core2 (Opt8) speedup was 19.4 (11.2). The asymptotic speedup over serial for LU panel factorization on the Core2 (Opt8) was 17.9 (15.9).

Recursive formulations represent the current state-of-the-art in panel factorization; these formulations improve performance through increased Level 3 BLAS use, which provides decent parallel speedups for reasonable sized problems. The LU recursion stops at two columns; the QR recursion stops at 16 columns (the highest-performance stopping point on both machines). Both QR and LU recurse only on the column dimension.

Comparing against the recursive formulations, our PCA approach asymptotically gets 2.5 (2.0) speedups on the Core2, for QR (LU). We get similar improvement on the Opt8, where we see a speedup of 2.25 (2.0) asymptotically for QR (LU). If we look at the aggregate L2 cache sizes, we can fit panels of 4096×128 or 9800×128 in the Opt8 and Core2 respectively. As core count increases, the size of the problem that fits in the aggregate caches should grow as well, but even when it is exceeded recursion can ameliorate the performance loss. Initial experiments on the Core2 show a roughly 20% decline in absolute performance for panels with 48,000 rows, yet this remains almost 2.5 times faster than the original recursive QR.

We next compare PCA for the QR panel factorization to the most credible alternatives in HPC libraries that are freely available today. By combining LAPACK with the GotoBLAS (the Goto-BLAS do not provide a native QR implementation) we can achieve a state-of-the-art traditional panel factorization, where all of the parallelism comes from the BLAS (we cannot do this with the ATLAS BLAS, because ATLAS does not parallelize the Level 2 BLAS). This method of parallelism is labeled **PGOTO** in Figure 4. We also time the **PLASMA** library [1] which aims to completely redesign LAPACK for large-scale parallelization. It parallelizes the panel using a tiling approach and dynamic assignment of operations to cores through the use of a DAG (Directed Acyclic Graph) of computational dependencies. PLASMA increases parallelism at the cost of performing significant extra flops.

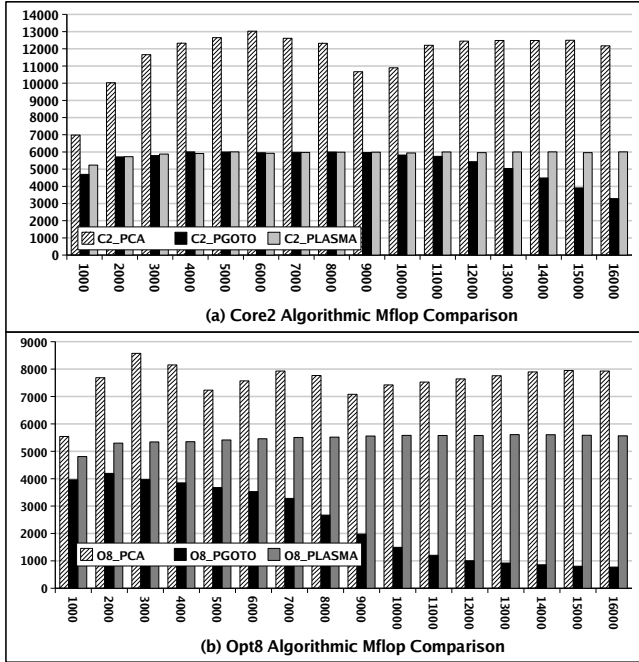


Figure 4. QR Panel Factorization MFLOPS on Core2, Opt8 PCA, Parallel GotoBLAS, PLASMA Parallel Algorithm

In Figure 4 all libraries are factoring the same size panel ($M \times 128$). Each library has been tuned separately to find its best performance on each problem size (i.e. we allowed different blocking factors for every M). We did an exhaustive search for each library, and took the blocking factors that produced the best average speed.

PCA and PGOTO are both building the $T \in \mathbb{R}^{128 \times 128}$ matrix required for the trailing matrix update (by DLARFB). PCA is the first bar (clear with diagonal lines) in each set; PGOTO is the second bar (black) and PLASMA is the third bar (gray). Note the scales are different; so PLASMA is getting roughly the same flop rate on both platforms. PGOTO begins to decline in performance for large problems, probably due to cache effects.

PCA achieves by far the fastest performance on the panel for both platforms. The speedups for the Core2 (Figure 4(a)) are easy to characterize, because the Parallel GotoBLAS and PLASMA plateau at nearly identical speeds of 6000 MFLOPS. PCA ramps up from a speedup (over both) of 1.5–1.6 at 1000 to a speedup of 2.3–2.4 at 3000. The dip in PCA performance at $M = 9000/10,000$ is where the benefit of copying the data has faded; for $M = 11,000$ and above, PCA is processing the data in-place. The asymptotic speedup is about 2.0 against PLASMA, and 2.42 against PGOTO.

On the Opt8 (Figure 4(b)) PLASMA and PGOTO differ markedly. PLASMA displays admirable consistency very quickly, settling in at 5600 MFLOPS, while PGOTO is declining throughout, finally reaching a speed of 775 MFLOPS. We suspect it is struggling with cache overflow problems. PCA shows a clear dip at 4500, the last panel size it can factor without recursion, and another dip at 9000, the last panel for which copying provides a speedup. It recovers to an asymptotic speed of 8000 MFLOPS. Our speedup over PGOTO climbs fairly steadily from 1.3 to about 2.9 at 8000, and from there to 10.2 at 16000. Against PLASMA, PCA starts out with a speedup of 1.15 at 1000, but then varies between 1.3 and 1.5 for the rest of the graph.

In summary, these experiments demonstrate that our new panel factorization produces superlinear speedups over the original LAPACK serial algorithm. We also achieve double or triple the per-

formance of the recursive parallel approaches. We are 1.5 to 2.5 times faster asymptotically than the traditional parallel BLAS-based panel factorization using the GotoBLAS, more than double the performance of tuned PLASMA on the Core2 and 30–50% faster on the Opt8.

6.1 Full Problem Performance

We have shown our approach to be the fastest panel factorization by a wide margin. Obviously, this means huge performance improvements if you are solving strongly overdetermined systems, but will it have a noticeable effect on the full algorithm running on large square problems? Figure 5 answers that question with a decided yes. This chart shows results for full QR on large square problems; the Y-axis charts the speedup of the ATLAS-tuned full QR (DGE-QRF) using our PCA panel factorization over the ATLAS-tuned DGEQRF which uses the default LAPACK panel factorization. The X-axis runs to 8000 by 8000 problems, in steps of 500.

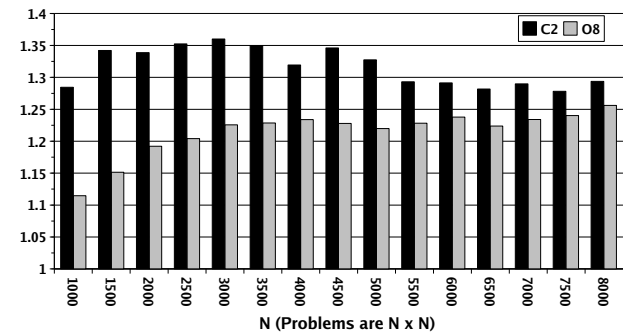


Figure 5. QR Full Problem Speedups, Core2 and Opt8

Figure 5 shows that our approach improves the asymptotic performance of the full problem by almost 30% (25%) on the Core2 (Opt8) on even the 8-node systems that are readily available today. There are two reasons for this boost; the first is simply that the panel factorizations have been reduced from 20% of the original runtime to less than 4% of the improved runtime. The second reason is more subtle; when the panel factorization is made to run faster, we can use it to factor a wider panel without a large performance penalty, which allows us to use a larger N_b when doing our update of the trailing matrix. These new PCA-tuned N_b 's are 2–4 times the N_b 's selected when tuning for the serial panel factorization. Using larger N_b allows the Level 3 BLAS to get greater cache reuse, which improves both serial and parallel performance. Unlike in LU, however, the optimal QR N_b is still strongly constrained, since the number of extra flops rises with N_b .

7. Scaling

We now would like to address *scaling*, more specifically *weak scaling*. Perfect weak scaling is experiencing the same performance/core as p is increased *when the work per core is kept constant*. Strong scaling is not what we aim for: we have shown superlinear speedups for fixed problem sizes, so for large problem sizes we have better than perfect strong scaling. Weak scaling, where each core has the same amount of work to do, and thus roughly the same amount of potential cache reuse to employ, is a much more honest measure of scalability for dense linear algebra. Figure 6 plots the weak scaling of our PCA algorithm. We keep the panel width constant (128), and the panel height for each core is 1000 (so when $p = 8$, we are factoring an 8000×128 QR).

In Figure 6(a), the X-axis is the number of cores used, the Y-axis is the MFLOP rate achieved. We can see that our scaling is quite good, as the performance goes up pretty much linearly

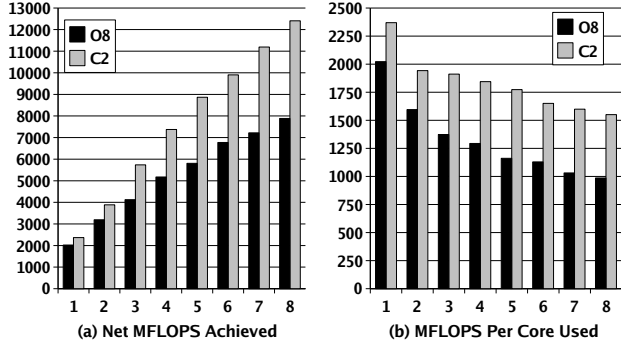


Figure 6. QR Scaling, Core2 and Opt8

with the number of cores. However, we are not achieving perfect weak scaling, as Figure 6(b) shows. In this figure, we have divided achieved MFLOPS from (a) by the number of cores to compute the MFLOPS per core. Here we can see the MFLOP rate per core is decreasing modestly. Almost all of this loss comes from the need to bring the panel into the cache, and the fact that the caches do not use true LRU line replacement. To understand how non-LRU replacement can cause problems, consider random replacement, where a read of enough data to fill a cache will, on some reads, randomly replace some of the data we have already read and will soon be using. For 8- or 12-way caches (like those on our test systems) only about 2/3 of the data read will actually be retained in the cache. We performed an experiment (not shown) in which we traversed the data several times; until statistically 99% of it could be expected to reside in the L2 cache (again, ignoring interrupts and context switches); and we compared the execution times (excluding copy times). The algorithm then achieved an almost flat performance per core curve; in fact this experiment eliminated roughly 90% of the decline shown in Figure 6(b). The remaining 10% is probably mainly threading overheads and cache misses due to interrupts and context switches.

8. Future work

This initial investigation paves the way for a host of future research. We should be able to use the same parallel cache assignment approach for Level 3 BLAS-based operations, including the original recursive panel factorizations. This should provide us with the same scalability, but with a higher peak performance (in our initial work, we chose the Level 2 BLAS-based approach because we were primarily interested in QR, where blocked algorithms require extra flops). More obviously, we can apply PCA to a host of additional LAPACK routines; the two-sided factorizations should be particularly interesting.

When the higher-level algorithm is rewritten explicitly to facilitate cache reuse as we do here, it makes sense to have an autotuning framework such as ATLAS tune the kernels being used for in-cache usage, since prior work [29, 32] has indicated that tuning for cache state can drastically change the best optimization set. The ATLAS framework has the basic support for this differential tuning, but it presently requires hand intervention during the install to get the framework to tune for varying levels of cache (by default, ATLAS only tunes for out-of-cache performance). This cache-sensitive tuning needs to be fully automated, and its impact on performance more fully explored.

If an operation cannot be transformed to enable much greater cache reuse, then it must be optimized specifically for the parallel context. For instance, essentially the only part of our PCA algorithm which does not scale perfectly with p is the initial/final data

copy, which is completely bus bound. Research questions include: is it possible to maximize the performance of even bus-bound operations by restricting the number of active memory consumers? What are the most efficient bus gating systems, does this vary by architecture, and how many consumer threads can be active before bus throughput begins to decrease⁵? Can we automate memory/compute staging for kernels that are only intermittently bus bound?

The approach presented here can be made even more powerful by combining it with pipelining and/or dynamic scheduling. As p continues to increase, many panel operations will be small enough to fit into only a subset of available parallel caches. In such a case, one could create two pipe stages with associated cores: the first stage runs our PCA panel operation, and produces data that drives the update stages. With large enough p , we can move cores amongst stages so that the speed of the stages are roughly equal, enabling all processors to be efficiently employed for all but the smallest of problems. This approach should fit nicely with current DAG-based approaches [4, 5, 20, 21, 24, 34], and is easily adapted for use in GPU-based factorizations.

Current GPU (Graphics Processing Units) factorizations [22, 27] often utilize the CPU to provide the panel factorization, in which case our algorithm can be used directly. More interesting is the idea of extending this approach directly to the GPU, where one would block for the registers rather than the cache, using the synchronization and register blocking methods outlined in [26].

9. Summary and Conclusions

We have presented a new approach to parallelizing panel operations in LAPACK. We believe this approach is truly general, and can be applied to pretty much any LAPACK-like panel operation; we have shown impressive results for both LU and QR factorizations, which are some of the most widely-used operations in dense linear algebra. For these operations, we achieved superlinear speedups over the original LAPACK serial algorithm (almost 20-fold asymptotic speedup for the best case), and impressive speedups over the prior state-of-the-art (asymptotically 2–2.5 times faster). Our approach can be used directly in present GPU-based factorizations, which today mostly rely on the CPU to handle the bus-bound panel factorizations (our algorithm should also be implementable on the actual GPU, where we can block for registers in the same way we currently do for the cache). We have shown that speeding up the panel factorization provides impressive speedups (as high as 1.35) for the full factorization on today’s machines; this improvement will grow with core count.

We have demonstrated excellent scaling for this algorithm and it is the most scalable panel factorization that we are aware of. This addresses a serious concern in HPC today, and a critical one for tomorrow. We conclude that with efficient parallel synchronization, it is worth moving parallel overheads into the innermost loop in order to increase both cache reuse and fine-grained parallelism, and that this approach should be extended to the Level 3-BLAS based algorithms as well.

References

- [1] Emmanuel Agullo, Bilel Hadri, Hatem Ltaief, and Jack Dongarra. Comparative study of one-sided factorizations with multiple software packages on multi-core hardware. In *SC’09: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2009. Accepted.
- [2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and

⁵This is architecture-specific, and therefore an obvious target for empirical tuning.

- D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 3rd edition, 1999.
- [3] C. Bischof and C. van Loan. The WY representation for products of Householder Transformations. *SIAM J. Sci. Statist. Comput.*, 8(1):s2–s13, Jan 1987.
 - [4] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, University of Tennessee, September 2007.
 - [5] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20:1573–1590, June 2008.
 - [6] Anthony M. Castaldo and R. Clint Whaley. Minimizing Startup Costs for Performance-Critical Threading. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.
 - [7] Ernie Chan, Enrique S. Quintana-Orti, Gregorio Quintana-Orti, and Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 116–125, 2007. 978-1-59593-667-7.
 - [8] James W. Demmel, Mark Hoemmen, Yozo Hida, and E. Jason Riedy. Non-Negative Diagonals and High Performance on Low-Profile Matrices from Householder QR. 2008. (LAPACK Working Note No. 203).
 - [9] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. Algorithm 656: An extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Transactions on Mathematical Software*, 14(1):18–32, 1988.
 - [10] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, 1988.
 - [11] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990.
 - [12] E. Elmroth and F. Gustavson. Applying recursion to serial and parallel qr factorization leads to better performance. *IBM Journal of Research and Development*, 44(4):605–624, 2000.
 - [13] Kazushige Goto. Gotoblas homepage. <http://www.tacc.utexas.edu/resources/software/>.
 - [14] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. Accepted for publication in *Transactions on Mathematical Software*, 2008.
 - [15] LAPACK group. Lapack homepage. <http://www.netlib.org/lapack/>.
 - [16] Brian C. Gunter and Robert A. van de Geijn. Parallel Out-Of-Core Computation and Updating of the QR Factorization. *ACM Transactions on Mathematical Software*, 31(3):60–78, 2005.
 - [17] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, 1997.
 - [18] R. Hanson, F. Krogh, and C. Lawson. A Proposal for Standard Linear Algebra Subprograms. *ACM SIGNUM Newsl.*, 8(16), 1973.
 - [19] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, 1979.
 - [20] Hatam Ltaief, Jakub Kurzak, and Jack Dongarra. Scheduling Two-sided Transformations using Algorithms-by-Tiles on Multicore Architectures. Technical Report UT-CS-09-638, University of Tennessee, April 2009.
 - [21] Mercedes Marques, Gregorio Quintana-Orti, Enrique S. Quintana-Orti, and Robert A. van de Geijn. Out-of-Core Computation of the QR Factorization on Multi-Core Processors. In accepted for publication in *Euro-Par 2009*, August 2009.
 - [22] Gregorio Quintana-Orti, Francisco D. Igual, Enrique S. Quintana-Orti, and Robert van de Geijn. Solving Dense Linear Algebra Problems on Platforms with Multiple and GPUs. Technical Report TR-08-22, University of Texas at Austin, May 2008.
 - [23] Robert Schreiber and Charles van Loan. A Storage Efficient WY Representation For Products of Householder Transformations. *SIAM J. Sci. Stat. Comput.*, 10(1):53–57, Jan 1989.
 - [24] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. Technical Report UT-CS-09-638, University of Tennessee, April 2009.
 - [25] S. Toledo. Locality of reference in lu decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications*, 18(4), 1997.
 - [26] Vasily Volkov and James Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Supercomputing*, November 2008.
 - [27] Vasily Volkov and James Demmel. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. Technical report, University of California, Berkeley, Berkeley, CA, USA, May 2008.
 - [28] R. Clint Whaley. Empirically tuning lapack's blocking factor for increased performance. In *Proceedings of the International Multi-conference on Computer Science and Information Technology*, Wisla, Poland, October 2008.
 - [29] R. Clint Whaley and Anthony M Castaldo. Achieving accurate and context-sensitive timing for code optimization. *Software: Practice and Experience*, 38(15):1621–1642, 2008.
 - [30] R. Clint Whaley and Antoine Petit. Atlas homepage. <http://math-atlas.sourceforge.net/>.
 - [31] R. Clint Whaley and Antoine Petit. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. <http://www.cs.utsa.edu/~whaley/papers/spercw04.ps>.
 - [32] R. Clint Whaley and David B. Whalley. Tuning high performance kernels through empirical compilation. In *The 2005 International Conference on Parallel Processing*, pages 89–98, Oslo, Norway, June 2005.
 - [33] R. Clint Whaley, Antoine Petit, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
 - [34] Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable Parallelization of FLAME Code via the Workqueuing Model. *ACM Transactions on Mathematical Software*, 34(2), March 2008.