# Automatic Blocking Of QR and LU Factorizations for Locality

Qing Yi[†] Ken Kennedy[‡] Haihang You[§] Keith Seymour[§] Jack Dongarra[§]

[†] Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
[‡] Rice University, Houston, TX 77005, USA
[§] University of Tennessee, Knoxville, TN, 37996, USA

## Categories and Subject Descriptors

D.1 [**Programming techniques**]: Automatic Programming; G.4 [**Mathematical Software**]: Efficiency

## General Terms

Algorithms, Performance, Measurement, Experimentation

## Keywords

loop optimizations, blocking, locality, LAPACK, QR, LU

## ABSTRACT

QR and LU factorizations for dense matrices are important linear algebra computations that are widely used in scientific applications. To efficiently perform these computations on modern computers, the factorization algorithms need to be blocked when operating on large matrices to effectively exploit the deep cache hierarchy prevalent in today's computer memory systems. Because both QR (based on Householder transformations) and LU factorization algorithms contain complex loop structures, few compilers can fully automate the blocking of these algorithms. Though linear algebra libraries such as LAPACK provides manually blocked implementations of these algorithms, by automatically generating blocked versions of the computations, more benefit can be gained such as automatic adaptation of different blocking strategies. This paper demonstrates how to apply an aggressive loop transformation technique, dependence hoisting, to produce efficient blockings for both QR and LU with partial pivoting. We present different blocking strategies that can be generated by our optimizer and compare the performance of auto-blocked versions with manually tuned versions in LAPACK, both using reference BLAS, ATLAS BLAS and native BLAS specially tuned for the underlying machine architectures.


*MSP'04* June 8, 2004, Washington, DC, USA.


## 1. INTRODUCTION

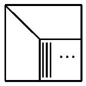
QR and LU factorizations for dense matrices are two important kernel computations in solving linear system equations and are included in many popular linear algebra libraries such as LINPACK [9] and LAPACK [3]. Because they are at the heart of many scientific applications, it is critical to provide efficient implementations of the factorization algorithms to achieve high performance on various advanced machine architectures. Specifically, to exploit the deep cache hierarchy prevalent in today's computer memory systems, these algorithms must be efficiently blocked when operating on large matrices so that data in caches are reused before being displaced from the caches.

Blocking (or tiling) is a highly effective strategy that enhances locality of applications by partitioning computations into smaller blocks. A set of unimodular loop transformation techniques [23, 6, 18] can efficiently block simple loop structures automatically. However, on more complicated loop structures, such as those in QR (based on Householder transformations) and LU (with partial pivoting) factorizations, these techniques often fail, even when an effective blocking is possible. As the result, few compilers can fully automate the blocking of these computations.

To illustrate the requirements of automatically blocking QR and LU, Figure 1 shows three equivalent versions of LU factorization without pivoting. These versions were initially introduced by Dongarra, Gustavson and Karp [10], with each version placing a different loop($k$, $i$ or $j$) at the outermost position. To fully block the non-pivoting LU code, a compiler needs to strip-mine each of the three loops ($k$, $i$ and $j$) and then shift the strip-enumerating loops inside. To achieve this blocking effect, the compiler must be able to freely interchange the nesting order of the three loops; that is, it must be able to freely translate between each pair of the three versions in Figure 1.

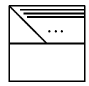
However, traditional unimodular transformation techniques cannot translate between Figure 1(a) (or (b)) and Figure 1(c) because these translations require the direct fusion and interchange of non-perfectly nested loops. For example, to translate (a) to (c), a compiler needs to fuse the $k(s_1)$ loop($k$ loop surrounding $s_1$) with the $j(s_2)$ loop in (a) and then place the fused loop outside the $k(s_2)$ loop. The fusion cannot be accomplished unless the original $k$ loop in (a) is first distributed. Since a dependence cycle connecting $s_1$ and $s_2$ is carried by this $k(s_1, s_2)$ loop, the distribution is not legal before fusion.

```
    do k = 1, n − 1
      do i = k + 1, n
s1:      a(i, k) = a(i, k)/a(k, k)
      enddo
      do j = k + 1, n
        do i = k + 1, n
s2:       a(i, j) = a(i, j)-a(i, k) ∗ a(k, j)
        enddo
      enddo
    enddo
```



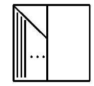
(a) KJI

```
    do i = 2, n
      do k = 1, i − 1
s1:     a(i, k) = a(i, k)/a(k, k)
        do j = k + 1, n
s2:       a(i, j) = a(i, j) − a(i, k) ∗ a(k, j)
        enddo
      enddo
    enddo
```



(b) IKJ

```
    do j = 1, n
      do k = 1, j − 1
        do i = k + 1, n
s2:       a(i, j) = a(i, j) − a(i, k) ∗ a(k, j)
        enddo
      enddo
      do i = j + 1, n
s1:     a(i, j) = a(i, j)/a(j, j)
      enddo
    enddo
```



(c) JKI

**Figure 1: Different versions of non-pivoting LU by Dongarra, Gustavson and Karp**

Both QR and LU with partial pivoting have similar loop structures as the code fragments shown in Figure 1. Similarly, to block these computations, compilers need the ability to directly fuse and interchange non-perfectly nested loops. As this requirement is beyond the capability of traditional compiler loop transformation techniques, few compiler implementations can effectively block these computations.

In this paper, We show how to automatically produce efficient blockings for both QR and partial-pivoting LU, using an earlier published novel loop transformation technique, dependence hoisting [25], that facilitates a combined fusion and interchange transformation of arbitrarily nested loops. In addition, we compare the performance of the auto-blocked QR and partial-pivoting LU with manually blocked ones from the popular linear algebra library, LAPACK [3], which has been carefully optimized by professional algorithm designers.

Although efficient blockings for QR and partial-pivoting LU already exists in LAPACK, by automatically blocking QR and LU, we potentially allow these kernels to be better optimized through automatic adaptation of different blocking strategies, while allowing libraries to maintain a single implementation for each kernel. We illustrate the wider optimization possibilities by automatically discovering blocking strategies other than the ones used by LAPACK. In addition, we show that the effect of blocking interacts with those of other lower-level optimizations. Our results indicate that better portable performance can be achieved by automatically constructing and exploiting the optimization search spaces.

In the following, Section 2 first introduces related work. Section 3 uses LU factorization without pivoting to illustrate how to apply dependence hoisting transformations to block arbitrarily nested loop structures. Section 4 and 5 then present the blocking of LU with partial pivoting and QR respectively. Section 6 presents experimental measurements for both QR and partial-pivoting LU. Conclusions are drawn in Section 7.

## 2. RELATED WORK

A set of unimodular loop transformations, including loop strip-mining, fusion, distribution, interchange, skewing and index-set splitting [23, 16, 18, 8, 11, 5], can be applied to successfully achieve blocking optimizations for scientific applications. These techniques are inexpensive and are widely used in production compilers to optimize simple loop structures, both for locality and for parallelization. However, these techniques are not effective enough when transforming complex, non-perfectly nested loop structures. Through the automated blocking of QR and LU factorizations, this paper illustrates how to use the earlier published dependence hoisting technique [25] to extend these traditional techniques in effectively transforming arbitrarily complex loop nests,

Carr, Kennedy and Lehoucq [4, 6] investigated the requirements for compilers to automatically optimize dense matrix QR, LU and Cholesky factorizations through unimodular loop transformation techniques. For partial-pivoting LU factorization, they were able to achieve comparable performance as that achieved by LAPACK [3]. However, their blocking strategy requires the insight that row interchange and column update matrix operations commute, an insight that can be achieved automatically by a compiler only through specialized pattern matching steps. Additionally, Carr and Lehoucq were able to produce only partial blocking for QR factorization, which were not as efficient as the LAPACK version of QR for large matrices. This paper further advances the results of their study through the application of *dependence hoisting*. Using *dependence hoisting*, we were able to fully automate the blocking of partial-pivoting LU without requiring any special commutativity insights. We were also able to automatically producing efficient blocking for QR and achieve competitive or even superior performance than that of the LAPACK QR for all matrix sizes.

Several compiler loop transformation frameworks [15, 20, 1, 17, 22] are theoretically more powerful but are also much more expensive than the dependence hoisting technique used in this paper. These general frameworks typically adopt a mathematical formulation of program dependences and transformations. They first compute a mapping from the iteration spaces of statements into some unified space. The unified space is then considered for transformation. Finally, a new program is constructed by mapping the selected transformations of the unified space onto the iteration spaces of statements. The computation of these mappings is expensive and generally requires special integer programming tools such as the Omega library [12]. Because of their high cost, these frameworks are rarely used in commercial compilers. In contrast, we seek simpler yet highly effective solutions with a much lower compile-time overhead. The compile-time overhead of applying dependence hoisting is comparable to that of applying unimodular loop transformation techniques, as presented in prior work by Yi, Kennedy and Adve [25].

Although dependence hoisting is only a combined loop fusion and interchange transformation, it can be integrated with other compiler techniques such as automatic selection of blocking factors [8, 16, 19], heuristics for loop fusion [14, 13], multi-level memory hierarchy management [7], and data layout transformations [21]. It thus extends traditional unimodular loop optimization systems with the ability to efficiently optimize arbitrarily complex structures. Specifically, for linear algebra kernels that manifest similar loop structures as those in QR and LU factorizations, by allowing multiple efficient blocked routines to be automatically generated, it enables these computations to be finer tuned automatically for high performance on different machine architectures.

# 3. DEPENDENCE HOISTING

We use the LU factorization code in Figure 1 to illustrate how to apply a novel transformation, *dependence hoisting*, to achieve efficient blocking for arbitrary loop structures. Dependence hoisting was described by Yi, Kennedy and Adve in detail in [25]. This section recapitulates the safety analysis and application steps of performing this transformation.

## 3.1 Safety Analysis

Dependence hoisting is a combined loop fusion and interchange transformation that fuses a collection of arbitrarily nested loops and then shifts the fused loop to the outermost position of an input loop structure. The collection of loops to be fused, together with their alignments during fusion, is denoted as a *computation slice* (or simply *slice*). The safety of the whole transformation is guaranteed by collecting only valid computation slices.

Given an arbitrarily nested loop structure $C$, a computation slice for $C$ contains the following information.

- *stmt-set*: the set of statements in $C$;
- *slice-loop*$(s)$ $\forall s \in$ *stmt-set*: for each statement $s$ in *stmt-set*, the *slicing loop* for $s$; that is, the selected loop (surrounding $s$) to be shifted to the outermost position;
- *slice-align*$(s)$ $\forall s \in$ *stmt-set*: for each statement $s$ in *stmt-set*, the alignment for *slice-loop*$(s)$.

Given the above slice, a dependence hoisting transformation fuses all the slicing loops into a single loop $\ell_f$ at the outermost position of $C$ s.t. $\forall\ \ell(s) = \textit{slice-loop}(s)$,

$$Ivar(\ell_f(s)) = Ivar(\ell(s)) + \textit{slice-align}(s). \qquad (1)$$

Here $Ivar(\ell(s))$ and $Ivar(\ell_f(s))$ are the induction variables for loops $\ell(s)$ and $\ell_f(s)$ respectively. Equation (1) specifies that each iteration instance $I$ of loop $\ell(s)$ $(= \textit{slice-loop}(s))$ is executed at iteration $I + \textit{slice-align}(s)$ of the fused loop $\ell_f$ after transformation.

To produce a correct dependence hoisting transformation, a valid computation slice must satisfy the following three conditions.

- it includes all the statements in $C$;
- all of its slicing loops can be legally shifted to the outermost loop level;
- each pair of slicing loops $\ell_x(s_x)$ and $\ell_y(s_y)$ can be legally fused s.t. $Ivar(\ell_x) + \textit{slice-align}(s_x) = Ivar(\ell_y) + \textit{slice-align}(s_y)$.

The above constraints are determined by examining the dependence constraints [2, 24, 25] of the original code fragment $C$. Figure 3(a) shows the original $KJI$ form of non-pivoting LU along with the dependence constraints between statements $s_1$ and $s_2$, where each dependence is marked with relations between iterations of the surrounding loops. Note that these relations involve not only iterations of common loops surrounding both $s_1$ and $s_2$ (for example, $k(s_1, s_2)$), but also non-common loops surrounding only one of the statements (for example, $j(s_2)$). The extra information is necessary to precisely model dependence constraints independent of the original loop structure.

Based on the dependence constraints in Figure 3(a), Figure 2 shows the three valid computation slices for this code (similar slices can be collected for Figure 3(b) and (c)). These slices can be used to freely translate between any two of the three loop orderings for non-pivoting LU in Figure 1. Section 3.2 illustrates how to translate (a) to (c) using $slice_j$. Similarly, using $slice_i$ can translate (a) to (b), and using $slice_k$ can translate (c) to (a).

## 3.2 Transformation Steps

We illustrate the application steps of dependence hoisting by translating Figure 3(a) to (c), using $slice_j$ from Figure 2. Specifically, we show how to facilitate the fusion of the $k(s_1)$ and $j(s_2)$ loops in (a) by successfully distributing the original $k(s_1, s_2)$ loop. As discussed in Section 1, this distribution cannot be achieved through unimodular loop transformation techniques because of the dependence cycle that connects statements $s_1$ and $s_2$ and is carried by the $k(s_1, s_2)$ loop.

The translation is in three steps. First, we create a new dummy loop surrounding the original code in Figure 3(a). This dummy loop has an index variable $x$ that iterates over the union of the iteration ranges of loops $k(s_1)$ and $j(s_2)$. In the same step, we insert conditionals in (a) so that statement $s_1$ is executed only when $x = j$ and statement $s_2$ is executed only when $x = k$. Figure 3(b) shows the result of this step, along with the modified dependences which include relations involving iterations of the new outermost $x$ loop.

Now, because the conditionals $x = k$ and $x = j$ in Figure 3(b) synchronize the $k(s_1)$ and $j(s_2)$ loops with the new $x(s_1, s_2)$ loop in a lock-step fashion, loop $x(s_1)$ always has the same dependence conditions as those of loop $k(s_1)$, and loop $x(s_2)$ always has the same dependence conditions as those of loop $j(s_2)$. As shown in the dependence graph of (b), the new outermost $x$ loop now carries the dependence edge from $s_1$ to $s_2$ and thus carries the dependence cycle connecting $s_1$ and $s_2$. This shifting of dependence level makes it possible for the second transformation step to distribute the $k(s_1, s_2)$ loop in (b), which no longer carries a dependence cycle. The transformed code after distribution is shown in Figure 4(a). Note that this step requires interchanging the order of statements $s_1$ and $s_2$.

Finally, we can now remove the redundant loops $k(s_1)$ and $j(s_2)$ in Figure 4(a) along with the conditionals that synchronize them with the outermost $x$ loop. To legally remove these loops and conditionals, we substitute the index variable $x$ for the index variables of the removed loops $k(s_1)$ and $j(s_2)$. In addition, we adjust the upper bound of the $k(s_2)$ loop to $x - 1$, in effect because the $j(s_2)$ loop is exchanged outward before being removed. The transformed code after this cleanup step is shown in Figure 4(b).

The final transformed code in Figure 4(b) is the same as the $JKI$ form of non-pivoting LU in Figure 1(c) except that the name of the outermost loop induction variable is $x$ instead of $j$. In reality, the induction variables of the new loops can often reuse those of the removed loops so that a compiler does not have to create a new loop induction

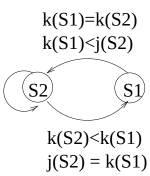
```
slice_j:
  stmt-set = {s1, s2}
  slice-loop(s1) = k(s1)
  slice-align(s1) = 0
  slice-loop(s2) = j(s2)
  slice-align(s2) = 0
```

```
slice_k:
  stmt-set = {s1, s2}
  slice-loop(s1) = k(s1)
  slice-align(s1) = 0
  slice-loop(s2) = k(s2)
  slice-align(s2) = 0
```

```
slice_i:
  stmt-set = {s1, s2}
  slice-loop(s1) = i(s1)
  slice-align(s1) = 0
  slice-loop(s2) = i(s2)
  slice-align(s2) = 0
```

**Figure 2: Computation slices for non-pivoting LU**

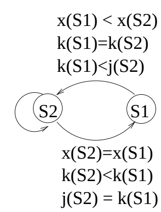
```
    do k = 1, n-1
      do i = k+1, n
s1:     a(i,k) = a(i,k)/a(k,k)
      enddo
      do j = k+1, n
        do i = k+1, n
s2:       a(i,j) = a(i,j)-a(i,k) * a(k,j)
        enddo
      enddo
    enddo
```




(a) original code

⇒

```
    do x = 1, n
      do k = 1, n-1
        do i = k+1, n
          if (k = x) then
s1:         a(i,k) = a(i,k)/a(k,k)
          endif
        enddo
        do j = k+1, n
          do i = k+1, n
            if (j = x) then
s2:           a(i,j) = a(i,j) - a(i,k) * a(k,j)
            endif
          enddo
        enddo
      enddo
    enddo
```




(b) after shifting dependence level

**Figure 3: Transforming $KJI$ version of non-pivoting LU. Step(1): shift dependence levels**

```
slice_j:
  stmt-set = {s1, s2, s3, s4, s5}
  slice-loop(s1) = j(s1)
  slice-align(s1) = 0
  slice-loop(s2) = j(s2)
  slice-align(s2) = 0
  slice-loop(s3) = j(s3)
  slice-align(s3) = 0
  slice-loop(s4) = j(s4)
  slice-align(s4) = 0
  slice-loop(s5) = j(s5)
  slice-align(s5) = 0
```

```
slice_k:
  stmt-set = {s1, s2, s3, s4, s5}
  slice-loop(s1) = j(s1)
  slice-align(s1) = 0
  slice-loop(s2) = j(s2)
  slice-align(s2) = 0
  slice-loop(s3) = k(s3)
  slice-align(s3) = 0
  slice-loop(s4) = j(s4)
  slice-align(s4) = 0
  slice-loop(s5) = k(s5)
  slice-align(s5) = 0
```

**Figure 5: Computation slices for partial-pivoting LU**

variable at each dependence hoisting transformation.

## 3.3 Achieving Blocking

Because dependence hoisting can be seen as a loop interchange transformation on arbitrarily nested loop structures, by combining dependence hoisting with loop strip-mining, we can achieve blocking for arbitrary loop structures. Specifically, given a collection of valid computation slices for a single loop structure $C$, we order these slices in the reverse of the desired nesting order of the corresponding loops. After using each slice to drive a dependence hoisting transformation, we strip-mine the new fused loop $\ell_f$ into a strip-counting loop $\ell_c$ and a strip-enumerating loop $\ell_t$. We then use loop $\ell_t$ as the input loop nest for further dependence hoisting transformations, which in turn will shift a new set of loops outside loop $\ell_t$ but inside loop $\ell_c$, thus blocking loop $\ell_f$. This process is further illustrated in Section 4 in blocking partial-pivoting LU factorization.

# 4. BLOCKING PARTIAL-PIVOTING LU

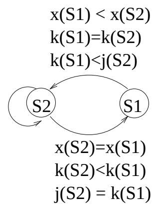
This section shows the effect of applying dependence hoisting (together with loop strip-mining) to block LU factorization with partial pivoting. Figure 6(a) presents the original non-blocked version generated from *dgetf2*, the right-looking Level2 LAPACK routine that computes the LU factorization of a $m \times n$ matrix. We obtained Figure 6(a) by first inlining all the subroutines invoked by *dgetf2* and then removing error-checking conditionals inside loops. In addition, we manually performed a preliminary loop index-set splitting transformation: as shown in Figure 6(a), the two $k$ loops ($k = j, n$ and $k = 1, j-1$) surrounding $s_3$ and $s_3'$ were split from a single original loop $k = 1, n$. This change is required to exclude statement $s_3'$ from participating the dependence hoisting transformations, as the dependence constraints connecting $s_3'$ would invalid the computation slice $slice_k$ in Figure 5. Although this step can be automated, it has not yet been implemented in our translator.

Before performing safety analysis of dependence hoisting, our translator applies a preliminary loop distribution step to separate out the statements that can disable certain dependence hoisting transformations. As the result, the original $j(s_3')$ and $k(s_3')$ loops in Figure 6(a) are separated into another loop nest, as shown in Figure 6(b) and (c). Our translator can then apply dependence hoisting transformations only to loops surrounding statements other than $s_3'$.

Figure 5 presents the two valid computation slices for the partial-pivoting LU code in Figure 6(a). The first slice, $slice_j$, selects the outermost $j$ loop as slicing loops for all the statements ($s_1$,$s_2$,$s_3$,$s_4$ and $s_5$); the second, $slice_k$, selects the $j$ loop for statements $s_1$,$s_2$ and $s_4$, but selects the $k$ loops as slicing loops for $s_3$ and $s_5$. Figure 6(b) shows the transformed code after using $slice_k$ to perform a dependence hoisting transformation, which effectively interchanges the nesting orders of the original $j$ and $k$ loops surrounding $s_3$ and $s_5$, and is similar to the translation of non-pivoting LU in Figure 4. Note that the original $k = j, n$ loop surrounding $s_3$ in Figure 6(a) has been split into $k = j$ and $k = j+1, n$ in (b), which accounts for $s_3$ appearing in two places. The details of the transformation algorithm can be found in [25].

In contrast to the original code in Figure 6(a), which sweeps the matrix from left to right, the transformed code

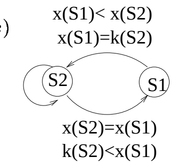
```
do x = 1, n
 do k = 1, n − 1
  do j = k + 1, n
   do i = k + 1, n
    if (j = x) then
s2:  a(i, j) = a(i, j) − a(i, k) ∗ a(k, j)
    endif
   enddo
  enddo
 enddo
 do k = 1, n − 1
  do i = k + 1, n
   if (k = x) then
s1: a(i, k) = a(i, k)/a(k, k)
   endif
  enddo
 enddo
enddo
```




(a) after distributing $k(s1, s2)$

⇒

```
do x = 1, n
  do k = 1, x − 1
    do i = k + 1, n
s2:     a(i, x) = a(i, x) − a(i, k) ∗ a(k, x)
    enddo
  enddo
  do i = x + 1, n
s1:   a(i, x) = a(i, x)/a(x, x)
  enddo
enddo
```




(b) after cleanup

**Figure 4: Transforming $KJI$ version of non-pivoting LU. Steps (2) and (3): distribute loops and cleanup**

in (b) defers all the row-interchange and column update operations (statements $s_3$ and $s_5$) to each column until the current column needs to be pivoted and scaled. Since these accumulated operations update the current column by reading other columns on its left, we have effectively translated the right-looking computation in (a) into a left-looking implementation in (b).

By strip-mining the outermost $k$ loop in Figure 6(b) and then using $slice_j$ in Figure 5 to perform another dependence hoisting transformation, our translator can automatically obtain the blocked partial-pivoting LU code in Figure 6(c). This code operates on a single block of columns at a time. As the update of each column block requires reading the left-hand side of the matrix, it is a left-looking computation. This blocking strategy is based on the observation that, although the code dealing with selecting pivots in Figure 6(a) imposes bi-directional dependence constraints among rows of the input matrix, the dependence constraints among columns of the matrix have only one direction—from columns on the left to columns on the right. Therefore the factorization can be blocked in the column direction of the matrix.

By reversing the application order of $slice_j$ and $slice_k$ in Figure 5, our translator can produce a different blocking for LU. That is, by first strip-mining the original outermost $j$ loop in Figure 6(a) and then using $slice_k$ to shift the $k$ loops surrounding $s_3$ and $s_5$ outside, we can obtain a blocked right-looking version that is very similar to the manually blocked routine of LU by LAPACK [3]. Section 6 presents the performance measurements of both blocked versions.

Note that the manually blocked partial-pivoting algorithm in LAPACK takes advantage of the knowledge that row interchange and column updates of a single matrix commute, irrespective of the dependence constraints among them. Our translator is based on dependence analysis and does not have this commutativity knowledge. The lack of such insight can degrade the overall performance of our auto-blocked codes, as shown in the performance measurements in Section 6.

## 5. BLOCKING QR

This section shows the effect of applying our translator to automatically block QR factorization. Figure 8(a) presents the original version generated from the level2 (non-blocked) LAPACK routine, *dgeqr2*, that computes the QR factorization of a real m by n matrix. Similarly to partial-pivoting LU, we inlined the invoked subroutines in *dgeqr2* and then removed error-checking conditionals inside loops.

$stmt\text{-}set = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$

| $slice_i$: | $slice_j$: |
|---|---|
| $slice\text{-}loop(s_1) = i(s_1)$ | $slice\text{-}loop(s_1) = i(s_1)$ |
| $slice\text{-}align(s_1) = 0$ | $slice\text{-}align(s_1) = 0$ |
| $slice\text{-}loop(s_2) = i(s_2)$ | $slice\text{-}loop(s_2) = i(s_2)$ |
| $slice\text{-}align(s_2) = 0$ | $slice\text{-}align(s_2) = 0$ |
| $slice\text{-}loop(s_3) = i(s_3)$ | $slice\text{-}loop(s_3) = i(s_3)$ |
| $slice\text{-}align(s_3) = 0$ | $slice\text{-}align(s_3) = 0$ |
| $slice\text{-}loop(s_4) = i(s_4)$ | $slice\text{-}loop(s_4) = i(s_4)$ |
| $slice\text{-}align(s_4) = 0$ | $slice\text{-}align(s_4) = 0$ |
| $slice\text{-}loop(s_5) = i(s_5)$ | $slice\text{-}loop(s_5) = i(s_5)$ |
| $slice\text{-}align(s_5) = 0$ | $slice\text{-}align(s_5) = 0$ |
| $slice\text{-}loop(s_6) = i(s_6)$ | $slice\text{-}loop(s_6) = j2(s_6)$ |
| $slice\text{-}align(s_6) = 0$ | $slice\text{-}align(s_6) = 0$ |
| $slice\text{-}loop(s_7) = i(s_7)$ | $slice\text{-}loop(s_7) = j2(s_7)$ |
| $slice\text{-}align(s_7) = 0$ | $slice\text{-}align(s_7) = 0$ |

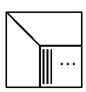
**Figure 7: Computation slices for QR**

Figure 7 shows the two valid computation slices for QR. Note here that statement $s'_5$ is excluded from dependence hoisting transformations and is thus separated into a single loop nest in the blocked code, shown in Figure 8(b). The separated $i(s'_5)$ loop restores the diagonal elements of the matrix with correct pivot values, which were saved in array $aii$ by statement $s_5$ during the factorization process.

The computation slices for QR are very similar to the ones for partial-pivoting LU. This is due to the almost identical dependence patterns of these two computations. As shown in the identical pictorial illustrations in Figure 8 and 6, both computations evaluate a pivot at each diagonal element (statements $s_1$,$s_2$, $s_3$ for QR and statements $s_1$, $s_2$ for partial-pivoting LU) by reading the lower half of the current column. Both of them then use the pivot value to scale the lower half of the current column (statement $s_4$ for both QR and LU). The scaled values of the current column are then used to update the right-hand side of the matrix(statements $s_6$,$s_7$ for QR and statements $s_5$ for LU). For both computations, the update of each element $a(i, j)$ depends only on the values of other elements $a(i', j')$ when $j' \le j$. Since the value of each element $a(i, j)$ depends only on other elements that are on the left of $a(i, j)$, we can block both factorizations in the column direction by accumulating operations on columns of the matrix.

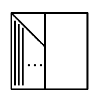
Figure 8(b) shows the blocked left-looking QR factoriza-

```
do j = 1, min( m, n )
s1: ipiv(j) = j
s1: tmp = dabs(a(j,j))
    do i = j+1,m
s2:     if (dabs(a(i,j)).gt.tmp) then
s2:        ipiv(j) = i
s2:        tmp = dabs(a(i,j))
s2:     endif
    enddo
    do k = j, n
s3:     tmp1 = a(j, k)
s3:     a(j,k) = a(ipiv(j), k)
s3:     a(ipiv(j), k) = tmp1
    enddo
    do k = 1, j-1
s3':    tmp2 = a(j, k)
s3':    a(j,k) = a(ipiv(j), k)
s3':    a(ipiv(j), k) = tmp2
    enddo
    do i = j+1, m
s4:     a(i,j) = a(i,j) / a(j,j)
    enddo
    do k = j+1, n
        do i = j+1, m
s5:        a(i,k) = a(i,k) - a(i,j) * a(j,k)
        enddo
    enddo
enddo
```



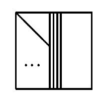
(a) original code

```
do k = 1, n, 1
    if (k .ge. 2) then
       do j = 1, min(m, -1 + k), 1
s3:       tmp1 = a(j, k)
s3:       a(j, k) = a(ipiv(j), k)
s3:       a(ipiv(j), k) = tmp1
          do i = 1 + j, m, 1
s5:          a(i, k) = a(i, k) - a(i, j) * a(j, k)
          enddo
       enddo
    endif
    if (k .le. min(m,n)) then
s1:    ipiv(k) = k
s1:    tmp = dabs(a(k, k))
       do i = 1 + k, m, 1
s2:       if (dabs(a(i, k)) .gt. tmp) then
s2:          ipiv(k) = i
s2:          tmp = dabs(a(i, k))
s2:       endif
       enddo
s3:    tmp1 = a(k, k)
s3:    a(k, k) = a(ipiv(k), k)
s3:    a(ipiv(k), k) = tmp1
       do i = 1 + k, m, 1
s4:       a(i, k) = a(i, k) / a(k, k)
       enddo
    endif
enddo
do k = 1, min(-1 + m, -1 + n), 1
    do j = 1 + k, min(m, n), 1
s3':   tmp2 = a(j, k)
s3':   a(j, k) = a(ipiv(j), k)
s3':   a(ipiv(j), k) = tmp2
    enddo
enddo
```



(b) after dependence hoisting

```
do x = 1, n, 16
    do j = 1, min(m,n-1, x-1), 1
       do k = x, min(n, x + 15)
s3:       tmp1 = a(j, k)
s3:       a(j, k) = a(ipiv(j), k)
s3:       a(ipiv(j), k) = tmp1
          do i = 1+j, m, 1
s5:          a(i, k) = a(i, k) - a(i, j) * a(j, k)
          enddo
       enddo
    enddo
    do j = x, min(m,n, x + 15), 1
s1:    ipiv(k) = k
s1:    tmp = dabs(a(j, j))
       do i = max(1 + j, 1 + x), m, 1
s2:       if (dabs(a(i, j)) .gt. tmp) then
s2:          ipiv(j) = i
s2:          tmp = dabs(a(i, j))
s2:       endif
       enddo
       do k = j, min(n, x + 15)
s3:       tmp1 = a(j, k)
s3:       a(j, k) = a(ipiv(j), k)
s3:       a(ipiv(j), k) = tmp1
       enddo
       do i = 1 + j, m, 1
s4:       a(i, j) = a(i, j) / a(j, j)
       enddo
       do k = 1+j, min(n, x + 15)
          do i = 1+j, m, 1
s5:          a(i, k) = a(i, k) - a(i, j) * a(j, k)
          enddo
       enddo
    enddo
enddo
do k = 1, min(-1 + m, -1 + n), 1
    do j = 1 + k, min(m, n), 1
s3':   tmp2 = a(j, k)
s3':   a(j, k) = a(ipiv(j), k)
s3':   a(ipiv(j), k) = tmp2
    enddo
enddo
```



(c) after blocking

**Figure 6: Blocking LU factorization with partial pivoting**

tion code, automatically generated by following similar steps as those for generating the partial-pivoting LU code in Figure 6(c). First, our translator uses $slice_j$ to shift the $j2(s_6)$ and $j2(s_7)$ loops outside of the original outermost $i$ loop in Figure 8(a). After strip-mining $slice_j$, it then shifts the loops in $slice_i$ outside the strip-enumerating loop of $j2(s_6, s_7)$. The blocked computation in Figure 8(b) effectively defers all the update operations (by statements $s_6$ and $s_7$) by performing the accumulated updates one block of columns at a time right before the column block needs to be pivoted and scaled. As the update of each column block requires reading the left-hand side of the matrix, it is a left-looking computation.

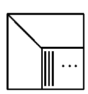
Similarly, by reversing the application order of $slice_j$ and $slice_i$; that is, by first strip-mining the original $i$ loop and then using $slice_j$ to shift the $j2(s_6)$ and $j2(s_7)$ loops outside, we can produce a differently blocked right-looking version of QR. This version delays using the evaluated values at each column to update the right-hand side of the matrix (by statements $s_6$ and $s_7$) until after a block of columns has been evaluated. The values of the whole block of columns are then collectively used to update the right-hand side of the matrix. Our blocked right-looking code is very similar to the manually blocked QR routine *dgeqrf* in LAPACK [3]. Section 6 will present the performance measurements for both automatic forms of blocking.

# 6. EXPERIMENTAL RESULTS

To evaluate the effectiveness of the optimizations described in this paper, this section compares the performance of the automatically blocked QR and LU factorization routines to that of the manually blocked routines(*dgeqrf* and *dgetrf*) in the LAPACK library [3]. We show that the auto-blocked implementations can achieve comparable or even superior performance than the manually blocked LAPACK ones, and that the overall performance of both QR and LU in LAPACK can be further improved through automatic construction and tuning of their optimization spaces.

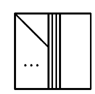
The QR and LU factorization implementations in LAPACK are built on top of BLAS, a collection of lower-level basic linear algebra subprograms including vector-vector operations (level1 BLAS), matrix-vector operations (level2 BLAS) and matrix-matrix operations(level3 BLAS). Many different implementations of BLAS exist and are optimized at different levels. To separate the effect of different blocking strategies from that of different BLAS-level optimizations, we must ensure that for each kernel, all blocked implementations are based on the same BLAS-level routines. Conse-

```
do i = 1, min(m,n)
s1: scale = zero
s1: ssq = one
    do j1 = i+1, m
s2:     if ( a(j1,i).ne.zero) then
s2:        absxi = abs( a( j1,i ) )
s2:        if ( scale.lt.absxi )then
s2:           ssq = one + ssq*( scale/absxi )**2
s2:           scale = absxi
s2:        else
s2:           ssq = ssq + ( absxi/scale )**2
s2:        endif
s2:     endif
    enddo
s3: xnorm = scale * sqrt( ssq )
s3: absxi = abs(a(i,i))
s3: if (absxi .le. xnorm) then
s3:     ssq = xnorm*sqrt( one+( absxi / xnorm )**2 )
s3: else
s3:     ssq = absxi*sqrt( one+( xnorm / absxi )**2 )
s3: endif
s3: beta = -sign( ssq, a(i,i))
s3: tau(i) = ( beta-a(i,i) ) / beta
    do j1 = i+1, m
s4:     a(j1,i) = a(j1,i) / (a(i,i) - beta)
    enddo
s5: a(i,i) = beta
s5: aii(i) = a( i, i )
s5: a( i, i ) = one
    do j2 = i+1, n
s6:     work(j2) = zero
        do j1 = i, m
s6:        work(j2) = work(j2) + a(j1,j2) * a(j1, i)
        enddo
    enddo
    do j2 = i+1, n
        do j1 = i, m
s7:        a(j1,j2) = a(j1,j2) - tau(i) * a(j1,i) * work(j2)
        enddo
    enddo
s5': a( i, i ) = aii(i)
enddo
```



(a) original code

```
do x = 1, max(n-1, m), 16
    do i = 1, x-1, 1
        do j2 = x, min(n-1, 15 + x), 1
s6:         work(j2 + 1) = zero
            do j1 = i, m, 1
s6:             work(j2 + 1) = work(j2 + 1) + a(j1, j2 + 1) * a(j1, i)
s7:             a(j1, j2 + 1) = a(j1, j2 + 1) - tau(i) * a(j1, i) * work(j2 + 1)
            enddo
        enddo
    enddo
    do i = x, min(m,n, 15+x), 1
s1:     scale = zero
s1:     ssq = one
        do j1 = 1 + i, m, 1
s2:         if (a(j1, i) .ne. zero) then
s2:             absxi = abs(a(j1, i))
s2:             if (scale .lt. absxi) then
s2:                ssq = one + ssq * (scale / absxi) ** 2
s2:                scale = absxi
s2:             else
s2:                ssq = ssq + (absxi / scale) ** 2
s2:             endif
s2:         endif
        enddo
s3:     xnorm = scale * sqrt(ssq)
s3:     absxi = abs(a(i, i))
s3:     if (absxi .le. xnorm) then
s3:         ssq = xnorm * sqrt(one + (absxi / xnorm) ** 2)
s3:     else
s3:         ssq = absxi * sqrt(one + (xnorm / absxi) ** 2)
s3:     endif
s3:     beta = -sign(ssq, a(i, i))
s3:     tau(i) = (beta - a(i, i)) / beta
        do j1 = 1 + i, m, 1
s4:         a(j1, i) = a(j1, i) / (a(i, i) - beta)
        enddo
s5:     a(i, i) = beta
s5:     aii(i) = a(i, i)
s5:     a(i, i) = one
        do j2 = x, min(n -1, 15 + x), 1
s6:         work(j2 + 1) = zero
            do j1 = i, m, 1
s6:             work(j2 + 1) = work(j2 + 1) + a(j1, j2 + 1) * a(j1, i)
s7:             a(j1, j2 + 1) = a(j1, j2 + 1) - tau(i) * a(j1, i) * work(j2 + 1)
            enddo
        enddo
    enddo
enddo
do i = 1, min(m, n), 1
s5': a(i, i) = aii(i)
enddo
```



(b) after blocking

**Figure 8: Blocking QR factorization**

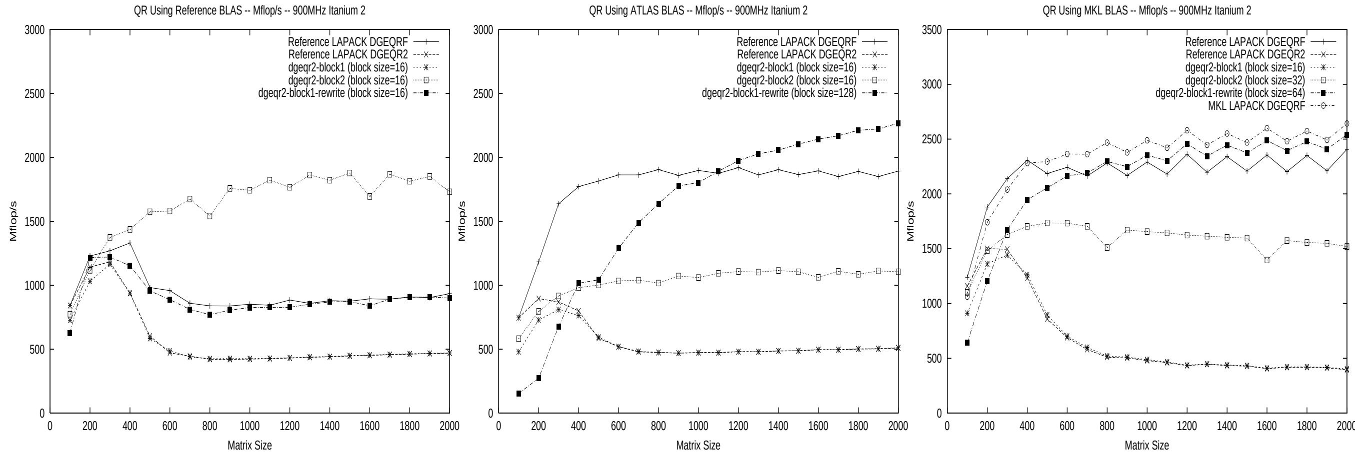
quently, we present measurements after manually rewriting our auto-blocked QR and LU to invoke the same set of BLAS routines as those invoked by the LAPACK implementations.

In the following explanations, *Reference LAPACK* refers to the reference implementation of LAPACK version 3.0, *Reference BLAS* refers to the reference implementation of BLAS, both available from Netlib. *ATLAS* refers to the automatically tuned BLAS from ATLAS version 3.6.0. *MKL* refers to the Intel Math Kernel Library version 6.1.1, which includes both LAPACK and BLAS routines.

## 6.1 Benchmark Measurements

Our translator has applied dependence hoisting transformations to perform a single optimization, blocking, to both QR and partial-pivoting LU factorizations. The initial versions of QR and LU are shown in Figure 8(a) and 6(a) respectively, which were hand-inlined versions of the level-2 (non-blocked) LAPACK routines *dgeqr2* and *dgetf2*. Our translator automatically produced two blocked versions for each factorization code (see Sections 4 and 5). In the following, we use *block1* to denote the blocked right-looking versions and use *block2* to denote the blocked left-looking versions.

Blocking is only one of the optimizations performed by LAPACK, which receives other optimizations through building on the BLAS library. To apply the same level of other optimizations to our auto-blocked routines as those received by LAPACK, we manually rewrote the auto-blocked routines to reverse the inlining effect; that is, when possible, we rewrote the code fragments that were originally from inlined BLAS subroutines back into the corresponding BLAS subroutine calls. This strategy allows us to factor out the performance impact from inlining when comparing auto-blocked versions with LAPACK routines. For performance measurements of




(a) Reference BLAS (b) ATLAS BLAS (c) MKL BLAS

**Figure 9: Performance of QR**

the inlined versions, see [25].

After rewriting the auto-blocked routines in a straightforward fashion to reverse the inlining effect, the de-inlined versions invoke only level2 BLAS routines that were in the original non-blocked LAPACK routines, *dgeqr2* and *dgetf2*. This situation places the auto-blocked routines in a serious disadvantage when compared with the manually blocked LAPACK routines, *dgeqrf* and *dgetrf*, which directly invoke level3 BLAS and specifically invoke *dgemm*, which are optimized at a much higher degree in ATLAS and vendor provided BLAS libraries. To separate the impact of blocking from other optimizations, we again manually rewrote the auto-blocked right-looking versions to invoke the level3 BLAS routines that were also invoked by *dgeqrf* and *dgetrf*, both of which also use a right-looking blocking strategy. In the following, we use *block1-rewrite* to denote the auto-blocked right-looking routines that invoke level3 BLAS. We did not produce the corresponding *block2-rewrite* versions for QR and LU because these left-looking routines have dramatically different loop structures than the right-looking ones, and we were unable to identify a similar set of level3 routines to be invoked within them.

We measured the performance of all the routines on a single processor Intel Itanium2 machine with 900MHz clock speed, 4GB memory, L1 instruction and data caches of 16KB each (L1 data cache not involved in floating-point loads), 256KB L2 cache and 1.5MB L3 cache. The operating system is Redhat Linux version 7.2 (kernel 2.4.18). All code was compiled with the Intel Fortran compiler (ifort) version 8.0 using optimization flag "-O3". Since the Itanium2 can issue 4 instructions per cycle, the theoretical peak performance is 3600 Mflop/s.

## 6.2 Performance of QR

Figure 9 compares the performance of different blocking implementations of QR factorization, using Reference BLAS, ATLAS BLAS and MKL BLAS respectively. The performance measurements for the following versions are presented:

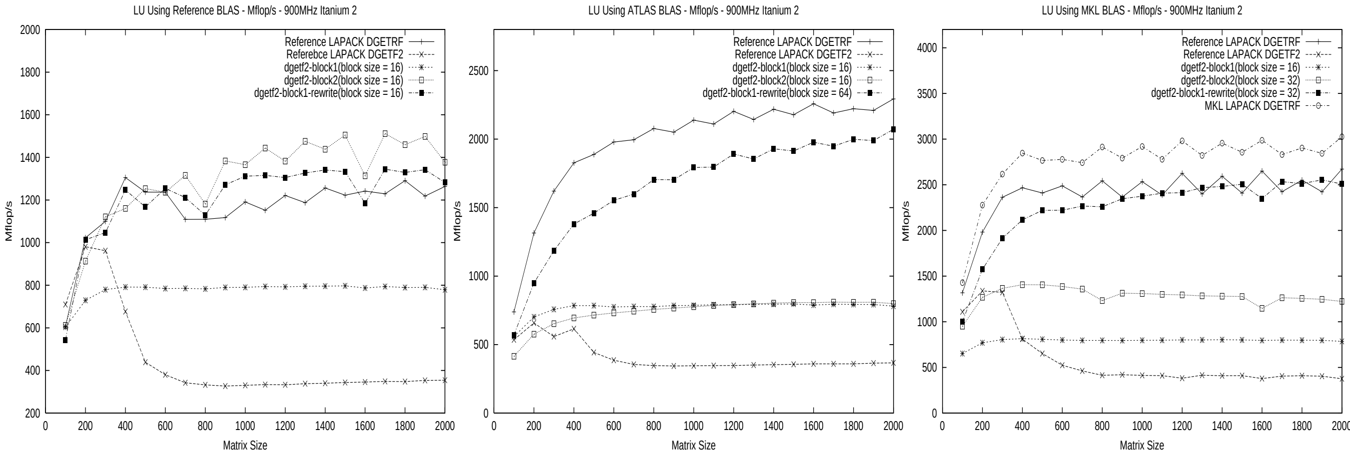
- Reference-LAPACK *dgeqrf*: manually blocked routine *dgeqrf* from Reference LAPACK;
- Reference-LAPACK *dgeqr2*: level2 (non-blocked) QR routine from Reference LAPACK;
- *dgeqr2-block1*: auto-blocked right-looking routine from dgeqr2;
- *dgeqr2-block2*: auto-blocked left-looking routine from dgeqr2;
- *dgeqr2-block1-rewrite*: manually rewritten *dgeqr2-block1* to invoke level3 BLAS.
- MKL LAPACK *dgeqrf*: manually blocked routine *dgeqrf* from MKL LAPACK.

Here because the MKL LAPACK routine *dgeqrf* requires special features that are present only in MKL BLAS, it was not measured using ATLAS or Reference BLAS. Both *dgeqr2-block1* and *dgeqr2-block2* were manually rewritten to reverse the inlining effect. We measured the performance of these versions using different block sizes, and presented the results using the best block sizes. The block size used by the Reference LAPACK routine *dgeqrf* is 32.

From Figure 9(a), using Reference-BLAS, the auto-blocked left-looking version (*dgeqr2-block2*) performs much better than the blocked right-looking versions, including both *dgeqr2-block1-rewrite* and Reference-LAPACK *dgeqrf*, which have very similar performance. Here because both level3 and level2 BLAS routines receive a similar level of optimization by the underlying Itanium2 compiler, the performance impact from invoking different BLAS routines is not significant. Because the blocked left-looking implementation manifests better cache locality, it achieves better performance than the other versions.

Note that *dgeqr2-block1*, the auto-blocked right-looking version without invoking level3 BLAS, did not even outperform the original non-blocked version, *dgeqr2*, in all graphs. This further indicates that the right-looking blocking strategy is not as beneficial as the left-looking one for QR.

When using ATLAS BLAS and MKL BLAS, however, the advantage of invoking level3 BLAS becomes dominant. As the result, the performance of the left-looking *dgeqr2-block2* (which invokes only level2 BLAS) lags behind those of *dgeqr2-block1-rewrite* and *Reference LAPACK*, both of which invoke level3 routines. If the left-looking *dgeqr2-block2* version were able to invoke specially optimized level3 BLAS routines as well, even better performance may pos-




(a) Reference BLAS (b) ATLAS BLAS (c) MKL BLAS

**Figure 10: Performance of partial-pivoting LU**

sibly be achieved. However, since no such routines are yet available, it is left to future investigations.

Note that *dgeqr2-block1-rewrite* is able to perform better than Reference-LAPACK *dgeqrf* in both Figure 9(b) and (c). In certain cases, it can even approach the performance of MKL LAPACK routine, which has been specially tuned for the Itanium2 machine. Here because *dgeqr2-block1-rewrite* has a very similar loop structure as that of the Reference LAPACK routine *dgeqrf*, the performance difference comes from our ability to manually select the best block sizes for the auto-blocked version (note that different block sizes were selected when using ATLAS and MKL BLAS). This indicates that *Reference LAPACK* can further benefit from finer tunings of different block sizes.

In summary, the auto-blocked versions were able to achieve similar or even superior performance than the Reference LAPACK versions for QR, though manual rewrite to invoke level3 BLAS routines is required in most cases. It is our future research to automate this process so that the auto-blocked versions can directly invoke specially optimized level3 BLAS. The automated translation may require techniques similar to the traditional compiler back-end peep-hole optimizations in selecting better machine instructions, except here the instructions are library routine calls, so knowledge about the semantics of the library routines as well as further loop restructuring may be required.

## 6.3 Performance of LU

Figure 9 presents the performance measurements for partial-pivoting LU, with a similar set of different implementations:

- Reference-LAPACK *dgetrf*: manually blocked routine *dgetrf* from Reference LAPACK;
- Reference-LAPACK *dgetf2*: level2 non-blocked LU routine from Reference LAPACK;
- *dgetf2-block1*: auto-blocked right-looking routine from *dgetf2*;
- *dgetf2-block2*: auto-blocked left-looking routine from *dgetf2*;
- *dgetf2-block1-rewrite*: manually rewritten *dgetf2-block1* to invoke level3 BLAS.
- MKL LAPACK *dgetrf*: manually blocked routine *dgetrf* from MKL LAPACK

Here the block size for the Reference LAPACK routine *dgetrf* is 64. Similar to QR, the MKL LAPACK routine *dgetrf* was measured using MKL BLAS only. When we manually generated *dgetf2-block1-rewrite*, we assumed the knowledge that row-interchange and column-update matrix operations commute, an insight also assumed by the manually blocked LAPACK routine *dgetrf*. Note that this knowledge was not assumed by our translator when automatically generating *dgetf2-block1* and *dgetf2-block2*.

When comparing different LU implementations in Figure 10, the patterns are very similar to those for QR in Figure 9 except that here all blocked routines consistently out-perform the non-blocked *dgetf2* routine. When using Reference-BLAS in Figure 10(a), the auto-blocked left-looking version *dgetf2-block2* performs better than all the right-looking ones for large matrices due to better cache locality. However, when using ATLAS BLAS and MKL BLAS in (b) and (c), the advantage of invoking level3 BLAS becomes dominant, and the performance of all versions that invoke level2 BLAS lag behind. After being manually rewritten to invoke level3 BLAS routines, the auto-blocked right-looking version *dgetf2-block1-rewrite* is able to perform comparably as Reference-LAPACK *dgetrf* when using *MKL BLAS* and perform slightly worse than Reference-LAPACK *dgetrf* when using ATLAS BLAS. Here although both *dgetf2-block1-rewrite* and Reference-LAPACK *dgetrf* invoke the level3 subroutine, *dgemm*, the LAPACK routine *dgetrf* additionally invokes another level3 routine *dlaswp* and thus has a slight advantage. In contrast, for QR, the rewritten *dgeqr2-block1-rewrite* invokes the same set of subroutines as those invoked by the LAPACK blocked routine *dgeqrf*.

## 6.4 Summary of Results

From Figure 9 and 10, the automatically generated blocked routines are quite effective for both QR and LU. Specifically, in all the performance graphs. at least one of the auto-blocked QR routines(*dgeqr2-block1*, *dgeqr2-block2* and *dgeqr2-block1-rewrite*) has achieved better performance than the Reference-LAPACK level3 routine *dgeqrf*. Similarly, at least one of the auto-blocked LU routines (*dgetf2-block1*, *dgetf2-block2* and *dgetf2-block1-rewrite*) has achieved comparable or slightly worse performance than the corresponding

Reference-LAPACK implementation *dgetrf*.

Note that the overall performance of both the auto-blocked and Reference-LAPACK level3 routines can be further improved. As shown in Figure 9(c) and 10(c), the vendor-provided MKL-LAPACK implementations of QR and LU have achieved better performance than both the auto-blocked and Reference-LAPACK versions. The high performance of MKL-LAPACK, however, is not portable. In fact, because MKL-LAPACK uses special features of the underlying machine, it can be built only on top of MKL-BLAS, and is likely to achieve poor performance if built on a different machine architecture.

Since we cannot expect all computer vendors to supply their specially optimized LAPACK and BLAS libraries, a more general approach is to combine Reference-LAPACK or automatic-blocking with empirical tuning approaches to achieve portable high performance. Figure 9(b) and 10(b) present the performance measurements of linking different versions of QR and LU with ATLAS BLAS (empirically tuned BLAS library), where the best blocked versions have achieved 75-85% of the MFLOP achieved by MKL-LAPACK + MKL-BLAS. We believe that much better performance can be achieved if empirical tuning is applied beyond BLAS. The need for better tuning is demonstrated by the performance of dgeqr2-block2 and dgetf2-block2, for which linking with ATLAS BLAS have actually degraded performance when compared with linking with Reference-BLAS. Further, better block-size selection is necessary because when linked with different versions of BLAS, each blocking strategy must use different block-sizes to achieve the best performance. These results indicate that the interactions between the blocking optimization in LAPACK and the other optimizations in BLAS need to be empirically tuned to achieve better performance.

In general, when optimizing the performance of applications, different optimizations often interact with each other and the overall search space is explosively large. The empirically tuned ATLAS BLAS library has been successful in achieving portable high performance. However, because the optimization space of ATLAS BLAS was constructed manually, it does not cover the entire optimization space and is not always optimal when linked with different applications. By providing better compiler techniques to automate the blocking of QR and LU, we potentially facilitate the fully automated generation and exploitation of their optimization spaces, and thus allow a much bigger search space to be exploited for better performance.

## 7. CONCLUSIONS

This paper illustrates how to apply a novel loop transformation technique, *dependence hoisting*, to automatically produce efficient blocking optimizations for both dense matrix QR and LU factorizations. By demonstrating our ability to automatically block arbitrarily complex loop structures for locality, we present the possibility of using compiler techniques to automatically adapt different loop structures for scientific applications.

## 8. REFERENCES


[1] N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Proceedings of the 2000 ACM International Conference on Supercomputing*, Santa Fe, New Mexico, May 2000.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, San Francisco, October 2001.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. The Society for Industrial and Applied Mathematics, 1999.

[4] S. Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Proceedings of Supercomputing*, Minneapolis, Nov. 1992.

[5] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, 1994.

[6] S. Carr and R. Lehoucq. Compiler blockability of dense matrix factorizations. *ACM Transactions on Mathematical Software*, 23(3), 1997.

[7] L. Carter, J. Ferrante, and S. F. Hummel. Hierarchical Tiling for Improved Superscalar Performance. In *Proc. 9th International Parallel Processing Symposium*, Santa Barbara, CA, Apr. 1995.

[8] S. Coleman and K. S. McKinley. Tile size selection using cache organization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[9] J. Dongarra, J. Bunch, C. Moler, and G.Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.

[10] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, Jan. 1984.

[11] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5(5):587–616, Oct. 1988.

[12] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical report, Dept. of Computer Science, Univ. of Maryland, College Park, Apr. 1996.

[13] K. Kennedy. Fast greedy weighted fusion. In *Proceedings of the International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[14] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, Aug. 1993. (also available as CRPC-TR94370).

[15] I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation*, Las Vegas, NV, June 1997.

[16] M. Lam, E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Santa Clara, Apr. 1991.

[17] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing*, Rhodes, Greece, June 1999.

[18] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[19] N. Mitchell, L. Carter, J. Ferrante, and K. Hgstedt. Quantifying the multi-level nature of tiling interactions. In *10th International Workshop on Languages and Compilers for Parallel Computing*, August 1997.

[20] W. Pugh. Uniform techniques for loop optimization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[21] G. Rivera and C.-W. Tseng. Data transformations for eliminating conflict misses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Montreal, Canada, June 1998.

[22] William Pugh and Evan Rosser. Iteration Space Slicing For Locality. In *LCPC 99*, July 1999.

[23] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, June 1991.

[24] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, Cambridge, MA, 1989.

[25] Q. Yi, K. Kennedy, and V. Adve. Transforming complex loop nests for locality. *The Journal Of Supercomputing*, 27:219–264, 2004.