Parallel Algorithm for the QR Decomposition with Column Pivoting in Multicore and GPU Processors

Andrés Tomás[1], Zhaojun Bai[2], and Vicente Hernández[3]

[1] University of California, Davis
andres@cs.ucdavis.edu
[2] University of California, Davis
bai@cs.ucdavis.edu
[3] Universitat Politècnica de València
vhernand@dsic.upv.es

Extended Abstract

Abstract. The QR decomposition with column pivoting (QRP) is a popular method for computing a rank revealing QR factorization. Current QRP algorithm implemented in LAPACK is block based, but its performance is limited by the BLAS level 2 operations required for pivoting criteria updates.
In this work an alternative parallel algorithm for QRP is proposed. This algorithm is column based with the columns distributed among processors. Our preliminary results show better performance than optimized LAPACK on current multicore processors. This algorithm is also suitable for GPU processors, where no other implementations of QRP has been presented in the literature.

Topics. Parallel and Distributed Computing

1 QR Decomposition with Column Pivoting

The QR decomposition with pivoting column (QRP) was the first method proposed [6] for computing a rank revealing QR factorization (RRQR). Although QRP could fail to fully reveal the rank of some matrices, it is still a popular method for practical applications. Besides, QRP can be used as a first step to more robust RRQR methods [2, 7]. Recently, QRP has been used for accelerating a practical implementation of the Jacobi SVD method [4, 5].

For $A \in \mathbb{R}^{m\times n}$, the QRP algorithm computes permutation matrix P, orthonormal Q and upper triangular matrix R such that

$$AP = QR,$$

where

$$|R_{ii}| \geq \sqrt{\sum_{k=i}^{j} |R_{kj}|^2}, \text{ for all } 1 \leq i \leq j \leq n.$$

Algorithm 1. QR decomposition with column pivoting

```
$p_{1:n} = 1 : n$
$c_{1:n} = \|Ae_{1:n}\|_2^2$
**for** $j = 1 : n$
    Choose $i$ such that $c_i = \max(c_{j:n})$
    **if** $i \neq j$
        swap($p_i$, $p_j$); swap($A_i$, $A_j$); swap($c_i$, $c_j$)
    **end**
    Determine a Householder matrix $H_j$ such that
        $H_j A_{j:m,j} = \pm\|A_{j:m,j}\|_2 e_1$
    $A_{j:m,j+1:n} = H_j A_{j:m,j+1:n}$
    $c_{j+1:n} = c_{j+1:n} - A_{j,j+1:n} \cdot A_{j,j+1:n}$
**end**
```

The formula for the norm downdate used in Algorithm 1 is simplified for readability, the current LAPACK implementation uses a more robust approach [3].

The main difference among the practical implementations of Algorithm 1 is how the application of the Householder reflectors is made. Since a Householder matrix H is a rank-1 modification of the identity,

$$H = I - \tau v v^T,$$

its application requires the computation

$$HA = A - \tau v v^T A,$$

which can be implemented using the three different BLAS levels:

Level 1 `xQRDC` is from the LINPACK library (predecessor of LAPACK). This implementation is column oriented in the sense that the matrix update is done column by column. For each column j, it uses `xDOT` to compute $v^T A e_j$ and then it updates Ae_j using `xAXPY`.

Level 2 `xGEQPF` is the original LAPACK routine (now deprecated). This implementation is matrix-vector oriented, it first computes the row vector $v^T A$ using `xGEMV` for a matrix-vector product, then applies a rank-1 update with the Level 2 BLAS routine `xGER`.

Level 3 `xGEQP3` is the current block based LAPACK implementation, it groups several rank-1 updates allowing to exploit BLAS level 3 operations [8].

The left plot in Figure 1 shows the performance of `DGEQP3` in three different multicore platforms, two from Intel and one from AMD. All these platforms have two processors but a different number of cores. The Intel Xeon X5530 and X5670 processors have 4 and 6 cores each, while the AMD Opteron 6172 has 12 cores. In these tests, the execution is set up to use one thread per core on the Intel platforms with the optimized MKL library. On the AMD platform the performance is measured using only 6 cores, because this configuration gives the best performance with the optimized Cray LAPACK library. The right plot

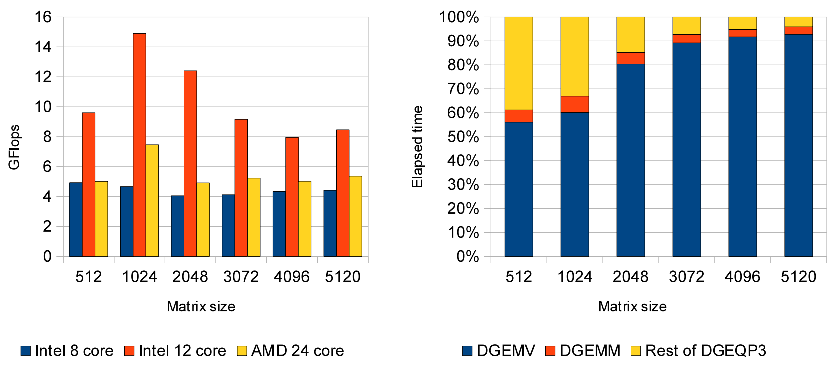

Fig. 1. Performance of DGEQP3 on three different multicore platforms (left). Time expend on DGEMV and DGEMM routines with respect of the total execution time of DGEQP3 on the Intel 12 core platform (right).

in Figure 1 shows the amount of time spent by DGEQP3 on calls to DGEMV and DGEMM with respect of the total execution time on the Intel 12 core platform. The reported amount of time for DGEMV do not include the unblocked part of DGEQP3, which is negligible for all the sizes reported.

The poor performance of DGEQP3 on these multicore platforms is because of the extensive use of DGEMV, which is limited by the memory bandwidth and do not scale with the number of cores. DGEQP3 uses the YTY^T representation [9] like the regular QR decomposition routine DGEQRF, but is not fully blocked because of the norm downdate. This downdate requires to compute the row vector v^TA for each Householder reflection applied to A. Although DGEQP3 only updates the columns of A every k (block size) times, still has to fetch the whole trailing matrix for the matrix-vector product. As the matrix size gets sufficiently big to not fit entirely in cache memory, the performance of DGEQP3 decreases to the level of DGEMV.

The DQRDC and DGEQPF routines have also this limitation, but DQRDC has the potential to have the best data locality of all three algorithms for small matrix sizes. If one column of the matrix fits entirely on cache memory, the DAXPY update in DQRDC would not fetch this column again. Therefore, DQRDC fetches each column of the matrix only once for each Householder reflection applied. In contrast, DGEQPF and DGEQP3 fetch each column of the matrix more than once.

While both DGEQPF and DGEQP3 require for good performance to fit the whole trailing matrix in cache memory, an implementation based on BLAS level 1 only requires to fit one column. With the large cache memory available in current processors, this allows good performance with practical matrix sizes.

2 Parallel QRP for Multicore Processors

In this work we propose a parallel algorithm for the QRP based on BLAS level 1 operations and a data distribution that provides good cache locality in current multicore architectures.

Algorithm 2. OpenMP parallel QR decomposition with pivoting

```
$p_{1:n} = 1 : n$
#pragma omp parallel
{
    $i$ = omp_get_thread_num()
    $t$ = omp_get_num_threads()
    **for** $j = 1 : n$
        **if** $i = j \bmod t$ **then** $c_j = \|Ae_j\|_2^2$
    **end**
    **for** $j = 1 : n$
        #pragma omp barrier
        **if** $i = j \bmod t$
            Choose $k$ such that $c_k = \max(c_{j:n})$
            **if** $k \neq j$
                swap($p_k$, $p_j$); swap($A_k$, $A_j$); swap($c_k$, $c_j$)
            **end**
            Determine a Householder matrix $H_j$ such that
                $H_j A_{j:m,j} = \pm\|A_{j:m,j}\|_2 e_1$
        **end**
        #pragma omp barrier
        **for** $r = j + 1 : n$
            **if** $i = r \bmod t$
                $A_{j:m,r} = H_j A_{j:m,r}$
                $c_r = c_r - A_{j,r}^2$
            **end**
        **end**
    **end**
}
```

The original design of LAPACK is to enclose parallelism inside BLAS routines. In a typical multicore implementation this means that each BLAS routine contains at least one OpenMP parallel section. Therefore, for each call to a BLAS routine a whole set of threads is started and stopped. This thread management overhead is negligible for level 3 routines, but it could be very significant for level 1 and 2 routines which have a lower operation count. In contrast, the parallelism in Algorithm 2 is not inside the BLAS operations, but among the vector operations required for all columns. Therefore, this algorithm only requires one OpenMP parallel section for the whole process. The critical part of the algorithm is implemented with synchronization primitives which should be more efficient than starting and stopping threads.

The Householder QR algorithm processes the columns of the matrix in their natural order from left to right. In a a parallel machine, it is natural to group the processors into a logical ring and deal out columns in a round-robin fashion. This technique staggers the computation across the processors and guarantees a load balanced computation. This distribution was first proposed in the context of the parallel implementation of a QR decomposition with local pivoting [1].

With this parallel distribution, each thread is ensured to work with the same subset of matrix columns during all the process. If the OpenMP runtime guarantees processor affinity, this will provide good memory locality at the lower levels of memory hierarchy. That is, as each core works only with a subset of the columns, there is a good probability of accessing a column already stored in the cache memories associated with this core.

As the number of cores increases in recent multicore processors, the architecture is gearing towards a NUMA (Non-Uniform Memory Access) model. In these architectures each core has direct access to a small part of the memory, but the rest must be accessed via some communication network to other core. This network is implemented by the cache hardware and is transparent to the user. Therefore, these processors can be still programmed using the same shared memory model as previous multicore processors. However, the memory access latency could have large variations depending on which part is accessed. In order to get good performance in these processors, techniques from the distributed memory programming paradigm can be used to reduce the communication among cores.

From the parallel distribution of Algorithm 2 a straightforward memory distribution can be easily derived. The memory physically close to each core should contain the columns updated by the thread associated to this core. This can be implemented in current operating systems by allocating and filling this memory from the thread itself. This technique is known in the literature as *first touch* policy. As Algorithm 2 creates all threads at the start, this initialization can be efficiently performed at the start of the process.

2.1 Preliminary Performance Results

Figure 2 shows our preliminary results from the OpenMP implementation of Algorithm 2 in the three different multicore platforms studied. In these tests, the execution of the proposed algorithm is set up to use one thread for each available core. To made this comparison fair, the performance of the proposed algorithm includes the cost of initializing the data distribution from the standard Fortran matrix storage.

The data plot in Figure 2 shows that the proposed algorithm is a huge improvement over the optimized `DGEQP3` for small to medium matrix sizes. For large matrix sizes the performance decreases, because a whole column do no fit entirely in cache memory. However, this low performance is still better that `DGEQP3` because of the block computations required. Also, the proposed algorithm has less thread management overhead than an implementation with parallel BLAS routines.

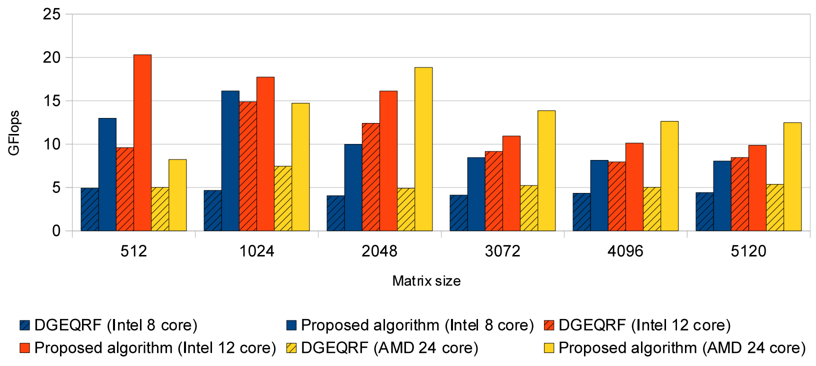

Fig. 2. Performance in GFlops of the optimized `DGEQP3` and the proposed OpenMP algorithm using one thread per core.

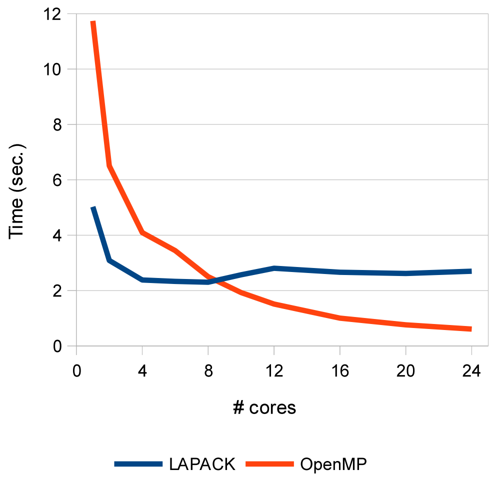

Fig. 3. Elapsed times for the pivoted QR of a 2048 × 2048 matrix using the proposed OpenMP algorithm and the optimized `DGEQP3` routine on the AMD 24 core processor.

On the AMD platform the performance is reported for `DGEQP3` using only 6 cores, because this configuration gives the best performance with the optimized Cray LAPACK library. Figure 3 shows clearly the effect caused by the NUMA architecture in the `DGEQP3` performance, where the memory transfer costs cancel out any performance increase with more than 6 cores. This is precisely the number of cores that physically share a block of memory in this platform. In contrast, the proposed algorithm has good scalability in all possible configurations.

3 Parallel QRP for GPU Processors

In order to achieve good performance on a GPU processor the computation must be divided in independent parallel subtasks. Where each subtask must be also suitable to efficient parallelization by a not so small number of processors (typically a multiple of 32). The parallel distribution of Algorithm 2 can be easily adapted to this GPU parallel model. The application of the Householder matrix is totally independent among columns, and the vector operations for each column have enough work for an efficient parallelization. Therefore, in terms of a CUDA implementation, each block j of threads computes first the dot product $\alpha = v^T Ae_j$ and then updates the column $Ae_j = Ae_j - \alpha\tau v$. If the matrix is sufficiently big, there is enough work to efficiently compute these two vector operations using all the threads in a wrap. As the memory access in current GPU processors is uniform, there is no need for the memory distribution used in multicore processors.

3.1 Preliminary Performance Results

Figure 4 show our preliminary results from the CUDA implementation of Algorithm 2 in two different GPU platforms. In contrast to the CPU case, The performance of the proposed algorithm improves as the matrix size increases. This is because with small sizes there is not sufficient work to exploit GPU parallelism and get the full bandwidth from graphics memory. The performance of the Tesla C2050 is not as good as the GeForce GTX480 because of the impact of ECC checking on memory bandwidth.

4 Summary and Concluding Remarks

The preliminary results show that depending on the platform and matrix size the proposed algorithm can be more than three times faster than the optimized `DGEQP3` routine. Besides, it always outperforms the optimized LAPACK versions for all three multicore platforms studied.

This algorithm can also be efficiently parallelized on a GPU processor, which is the first GPU implementation of the QR decomposition with pivoting in the literature. This implementation shows promising results, obtaining better performance than `DGEQP3` for big matrices.

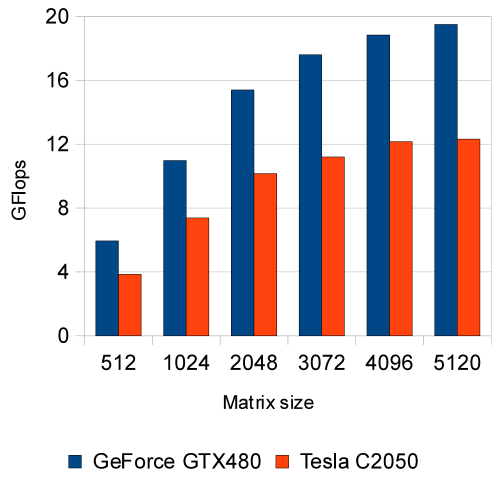

Fig. 4. Performance of the proposed GPU implementation on a GeForce GTX 480 and a Tesla C2050 graphics card from Nvidia.

As future work we want to include some blocking into the algorithm while keeping its parallel distribution. In particular, the `xAPXY` part of the Householder matrix application can be grouped reducing the number of memory writes. This should give a performance boost of the GPU implementation because graphic memory writes usually have bigger latency than reads.

References

1. Bischof, C.H.: A parallel QR factorization algorithm with controlled local pivoting. SIAM J. Sci. Stat. Comput. 12, 36–57 (January 1991)
2. Chandrasekaran, S., Ipsen, I.C.F.: On rank-revealing factorisations. SIAM J. Matrix Anal. Appl. 15, 592–622 (April 1994)
3. Drmač, Z., Bujanović, Z.: On the failure of rank-revealing QR factorization software – a case study. ACM Trans. Math. Softw. 35, 12:1–12:28 (July 2008)
4. Drmač, Z., Veselić, K.: New fast and accurate Jacobi SVD algorithm I. SIAM J. Matrix Anal. Appl. 29, 1322–1342 (January 2008)
5. Drmač, Z., Veselić, K.: New fast and accurate Jacobi SVD algorithm II. SIAM J. Matrix Anal. Appl. 29, 1343–1362 (January 2008)
6. Golub, G.H.: Numerical methods for solving linear least squares problems. Numer. Math. 7, 206–216 (1965)
7. Gu, M., Eisenstat, S.: Efficient algorithms for computing a strong rank-revealing QR factorization. SIAM J. Sci. Comput. 17(4), 848–869 (July 1996)
8. Quintana-Orti, G., Sun, X., Bischof, C.H.: A BLAS-3 version of the QR factorization with column pivoting. SIAM J. Sci. Comput. 19(5), 1486–1494 (1998)
9. Schreiber, R., van Loan, C.: A storage-efficient WY representation for products of Householder transformations. SIAM J. Sci. Stat. Comput. 10, 53–57 (January 1989)