Parallelization of the QR Decomposition with Column Pivoting Using Column Cyclic Distribution on Multicore and GPU Processors

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

[1] Department of Computer Science,
University of California, Davis, CA 95616, USA
{andres,bai}@cs.ucdavis.edu
[2] Dept. Sistemas Informáticos y Computaciòn,
Universitat Politècnica de València E-46022 Valencia, Spain
vhernand@dsic.upv.es

Abstract. The QR decomposition with column pivoting (QRP) of a matrix is widely used for numerical rank revealing in applications. The performance of LAPACK implementation (`DGEQP3`) of the Householder QRP algorithm is limited by Level 2 BLAS operations required for updating the column norms. In this paper, we propose an implementation of the QRP algorithm using a distribution of the matrix columns in a round-robin fashion for better data locality and parallel memory bus utilization on multicore architectures. Our performance results show a 60% improvement over the routine `DGEQP3` of Intel MKL (version 10.3) on a 12 core Intel Xeon X5670 machine. In addition, we show that the same data distribution is also suitable for general purpose GPU processors, where our implementation obtains up to 90 GFlops on a NVIDIA GeForce GTX480. This is about 2 times faster than the QRP implementation of MAGMA (version 1.2.1).

Topics. Parallel and Distributed Computing

1 QR decomposition with column pivoting

The QR decomposition with column pivoting (QRP) is proposed for computing a rank revealing QR factorization (RRQR) [7]. Although the QRP decomposition may fail to reveal the numerical rank correctly, it is a popular and economical method in many applications. Furthermore, the QRP is used as the first step to more robust RRQR methods [2, 8] and for accelerating a Jacobi method for computing the singular value decomposition [5, 6].

The QRP decomposition of $A \in \mathbb{R}^{m \times n}$ is defined by an orthonormal Q and a upper triangular matrix R such that

$$AP = QR,$$

where P is a permutation matrix chosen so that

$$|r_{11}| \geq |r_{22}| \geq \cdots \geq |r_{nn}|$$

and moreover, for each i,

$$|R_{ii}| \geq \|R_{(k:j,j)}\|_2 \quad \text{for } j = i+1, \ldots, n.$$

An outline of the Householder QRP algorithm is shown in Algorithm 1. Note that the formula for the column norm updating is simplified here. The current LAPACK implementation uses a more robust approach [4]. The omitted detail is not relevant to the parallelization discussed in this paper.

Algorithm 1. QR decomposition with column pivoting

```
1   p_{1:n} = 1 : n
2   c_{1:n} = ||A e_{1:n}||_2^2
3   for j = 1 : n
4       Choose i such that c_i = max(c_{j:n})
5       if i ≠ j
6           swap(p_i, p_j); swap(A_i, A_j); swap(c_i, c_j)
7       end
8       Determine a Householder matrix H_j such that
9           H_j A_{j:m,j} = ±||A_{j:m,j}||_2 e_1
10      A_{j:m,j+1:n} = H_j A_{j:m,j+1:n}
11      c_{j+1:n} = c_{j+1:n} − A_{j,j+1:n} · A_{j,j+1:n}
12  end
```

The main difference among the various implementations of Algorithm 1 is how the Householder reflectors are applied. Since a Householder reflector H is a rank-one 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 three levels of BLAS. The current LAPACK implementation (`xGEQP3`) is block based. It groups several rank-one updates for exploiting the Level 3 BLAS operations [9].

Figure 1 shows the performance of Intel MKL[3] routine `DGEQP3` on a 12 core (6 cores per socket) Intel Xeon X5670 machine. Here the execution is set up to use one thread per core. The poor performance of `DGEQP3` compared to `DGEQRF` is because of the extensive use of the Level 2 BLAS operation `DGEMV` for column norm updates, which is limited by the memory bandwidth and do not scale with the number of cores. Figure 2 shows the amount of time spent by `DGEQP3` on calls to `DGEMV` and `DGEMM` relative to the total execution time on the same platform. The reported amount of time for `DGEMV` does not include the unblocked part of `DGEQP3` which processes the last block of the matrix.

[3] Intel MKL version 10.3 http://software.intel.com/en-us/intel-mkl

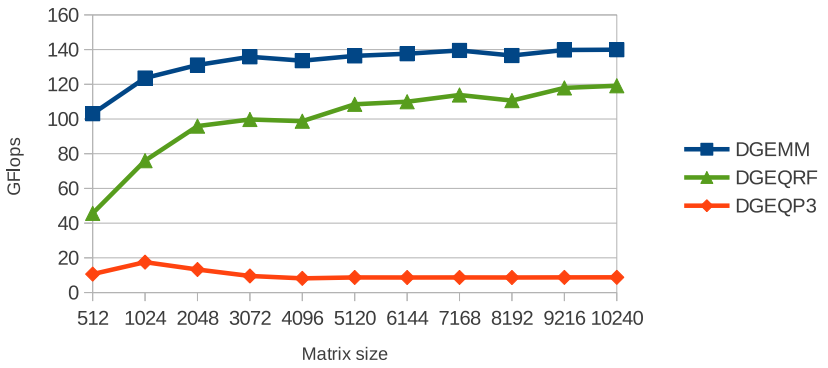

Fig. 1. Performance of Intel MKL routines `DGEMM`, `DGEQRF`, and `DGEQP3` on a 12 core (6 cores per socket) Intel Xeon X5670 machine.

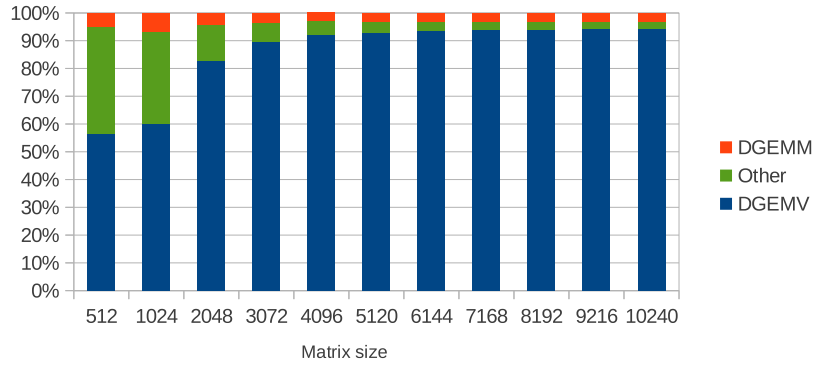

Fig. 2. Execution time in percentage of Intel MKL routines `DGEMV` and `DGEMM` in `DGEQP3` on a 12 core (6 cores per socket) Intel Xeon X5670 machine.

Although DGEQP3 uses the YTY^T representation [10] like DGEQRF, it is not fully blocked because of the column norm updating. The column norm updating requires to compute the row vector $v^T A$ for each Householder reflection applied to A. DGEQP3 updates the columns of A every k (block size) times. However, it still has to fetch the whole trailing matrix for the matrix-vector product. As the matrix size gets too large to fit entirely in cache memory, the performance of DGEQP3 decreases to the level of DGEMV.

2 Parallel QRP for multicore processors

The design of LAPACK DGEQP3 routine 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 BLAS, a whole set of threads is started and stopped. This thread management overhead is negligible for Level 3 BLAS operations, but it could be very significant for Level 1 and 2 BLAS operations due to the low computational intensity, namely, the low average number of floating point operations per memory access.

In contrast, we propose the following Algorithm 2 to use only one OpenMP parallel section. The parallelism here is not inside BLAS operations, but among the vector computations required for all columns. The critical parts of the algorithm are implemented with synchronization primitives, which is more efficient than starting and stopping threads.

Algorithm 2 is a block algorithm, where the block size is denoted as b. We assume without loss of generality that the matrix size is an exact multiple of b. The loop from lines 6 to 29 performs the panel factorization, that is, the QR factorization of the first b columns. The only sequential part of this loop is the pivot selection and computation of the Householder transform (lines 7 to 13). The rest of the loop updates the matrix F used to accumulate part of the Householder matrices application,

$$H_1 H_2 \cdots H_k A = A - YTY^T A = A - YF^T.$$

The last loop in Algorithm 2 applies the Householder matrices to the rest of the matrix (lines 30 to 35). Recent work on using a parallel cache assignment approach to speed up the panel factorization can be found in [3].

Algorithm 2 processes the columns of the matrix in their natural order from left to right. On a parallel machine, it is natural to group the processors into a logical ring and deal 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]. The selection of which columns are processed by each thread is not left to the OpenMP runtime, but explicitly controlled in lines 8, 15, and 31.

Algorithm 2. OpenMP parallel QRP using column cyclic distribution

```
1  p_{1:n} = 1 : n
2  #pragma omp parallel
3      i = omp_get_thread_num(); t = omp_get_num_threads()
4      c_{1:n} = ‖A e_{1:n}‖_2^2
5      for r = 1 : b : n − 1
6          for k = 1 : b
7              #pragma omp barrier
8              if r + k − 1 mod t = i
9                  Choose u such that c_u = max(c_{j:n})
10                 if u ≠ j then swap(p_u, p_j); swap(A_u, A_j); swap(c_u, c_j)
11                 Apply previous transformations to A_j ← A_j + Y F^T
12                 Determine Householder matrix H_j
13             end
14             #pragma omp barrier
15             for j = r : r + k − 1
16                 if j mod t = i then T_{j−r+1,k} = −τ_{r+k} Y_{r+k−1} Y_j
17             end
18             #pragma omp barrier
19             for j = r : n
20                 if j mod t = i
21                     F_{k,j} = F_{:,j} T_{:,k}
22                     if j > r + k − 1
23                         F_{k,j} ← F_{k,j} − τ_{r+k} Y_{r+k−1} A_j
24                         A_{r+k−1,j} ← Y_{r+k−1,:} F_{:,j}^T
25                         c_r = c_r − A_{j,r}^2
26                     end
27                 end
28             end
29         end
30         #pragma omp barrier
31         for j = r + b : n
32             if j mod t = i
33                 A_{j:m,j} ← A_{j:m,j} + F_{:,j}^T Y_{j,:}
34             end
35         end
36     end
```

With the column cyclic distribution, each thread is ensured to work with the same subset of matrix columns during all the processes. If the OpenMP runtime guarantees processor affinity, this will provide good memory locality at the lower levels of memory hierarchy. This is important in modern multicore processors where each core has typically its own L1 cache. As each thread works only with a subset of the columns, there is a good probability of accessing a column already stored in the L1 cache inside the core associated with this thread.

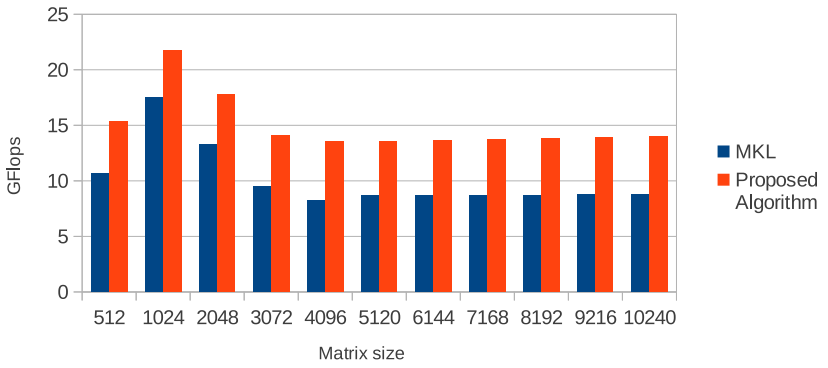

Fig. 3. Performance comparison of the proposed algorithm and Intel MKL routine `DGEQP3` on a 12 core (6 cores per socket) Intel Xeon X5670 machine.

As the number of cores increases on modern multicore processors, the architecture is gearing towards a non-uniform memory access (NUMA) model. On these architectures, each core has direct access to a part of the memory, but the rest of the memory 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 of the memory is accessed. In order to get good performance on these processors, techniques from the distributed memory programming paradigm can be used to reduce the communication among cores.

By the column 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.

To guarantee that each consecutive column is stored in a different physical memory page, the matrix must padded so the column length is a multiple of the page size. This memory overhead could be important for small matrices, because the typical page size on current platforms is 4096 bytes.

3 Performance results

Figure 3 shows the results from an OpenMP implementation of Algorithm 2 on a 12 core (6 cores per socket) Intel Xeon X5670 machine. Here the execution of

the proposed algorithm is set up to use one thread for each available core. The block size for the proposed algorithm is set to 48, which it has been determined empirically as the optimal for the platform. Moreover, the performance of the proposed algorithm includes the cost of initializing the data distribution from the standard Fortran matrix storage.

The proposed algorithm shows about 60% improvement over the optimized `DGEQP3` in Intel MKL for the matrix sizes tested. This improvement comes from the data distribution, which allows faster memory access as a result of better data locality and parallel memory bus utilization. Moreover, the proposed algorithm has less thread management overhead than the implementation which keeps parallelism inside BLAS operations.

4 Parallel QRP for GPU processors

In order to achieve good performance on a general purpose GPU processor, the computation must be divided into independent parallel subtasks. Moreover, each subtask must be also suitable to efficient parallelization by a certain number of processors (typically a multiple of 32). The parallel distribution of Algorithm 2 can be easily adapted to the GPU parallel model, assigning each column to a block of threads. If the matrix is sufficiently large, there is enough work to keep all processors in a block busy, and enough blocks to keep the whole GPU busy. The main difference with the multicore version is that the memory distribution is not required, because the memory access on current GPU processors is uniform.

Figures 4 and 5 compare the GPU performance of Algorithm 2 and MAGMA's `xGEQP3` routines[4] on two NVIDIA Fermi platforms. In single precision, the GPU implementation of Algorithm 2 obtains about 60 GFlops on Tesla C2050 and 90 GFlops on GeForce GTX480. This is about two times faster than the MAGMA implementation using the same hardware. The improvement is because our implementation runs entirely on the GPU, with no memory transfers from the CPU. In contrast, the panel factorization of `xGEQP3` in MAGMA is performed on the CPU and the trailing matrix update on the GPU. This approach works quite well for the LU decomposition with partial pivoting and the QR without pivoting, because the panel factorization can be computed in parallel while the GPU is still updating the trailing matrix with the previous block. However, the overlap of computation and communication is not possible for the QR decomposition with pivoting. The pivoting criteria requires that the trailing matrix update must be completed before starting the panel factorization.

The performance results confirm that the QRP decomposition is limited by memory speed. Therefore a GPU platform is more adequate for computing the QRP than a traditional CPU because of raw memory bandwidth. Another interesting observation is that the low-end GeForce GTX480 obtains better performance than the high-end Tesla C2050 even in double precision. This is because the C2050 has less memory bandwidth (in part due to ECC checking, which could not be disabled in our experiments).

[4] MAGMA version 1.2.1 released on June 29, 2012. http://icl.cs.utk.edu/magma/

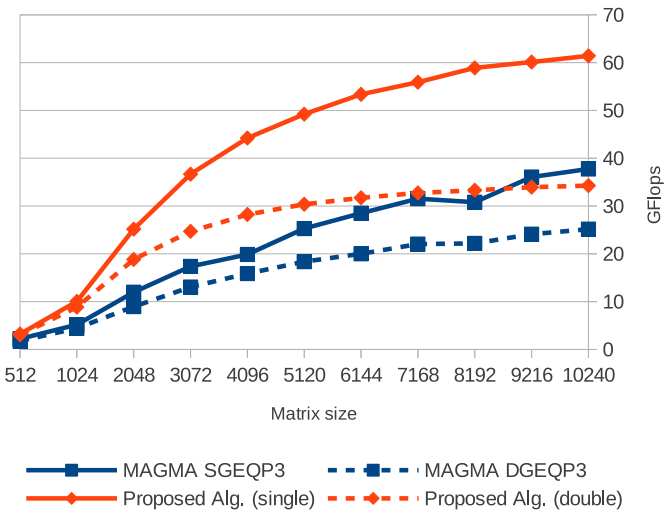

Fig. 4. Performance of the proposed GPU implementation and MAGMA implementation of `xGEQP3` on NVIDIA Tesla C2050.

5 Conclusions

We proposed a parallel algorithm for computing the QRP decomposition on multicores. This algorithm uses a column cyclic memory distribution and only one parallel OpenMP section. With the column cyclic distribution each processor works with a subset of the columns, improving memory access bandwidth and data locality. Moreover, it has lower thread management overhead than the implementations which only use parallelism inside BLAS operations. The proposed algorithm is about 60% faster than Intel MKL routine `DGEQP3` on a 12 core Intel Xeon X5670 machine.

Although the column cyclic data distribution is not required on a GPU, the same strategy is employed to allocate work among the processors. Our CUDA implementation of the QRP is about 2 times faster than the MAGMA version of `xGEQP3` on NVIDIA Fermi GPUs. Our implementation runs entirely on the GPU, while MAGMA's implementation splits the work between CPU and GPU, which requires expensive data transfers. In other decompositions, such as LU or QR, these transfers can be overlapped with computations, but this optimization cannot be applied to QRP because of the pivoting selection criteria.

Acknowledgment. Tomás and Bai were supported in part by the U.S. DOE SciDAC grant DOE-DE-FC0206ER25793 and NSF grant PHY1005502. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC02-05CH11231.

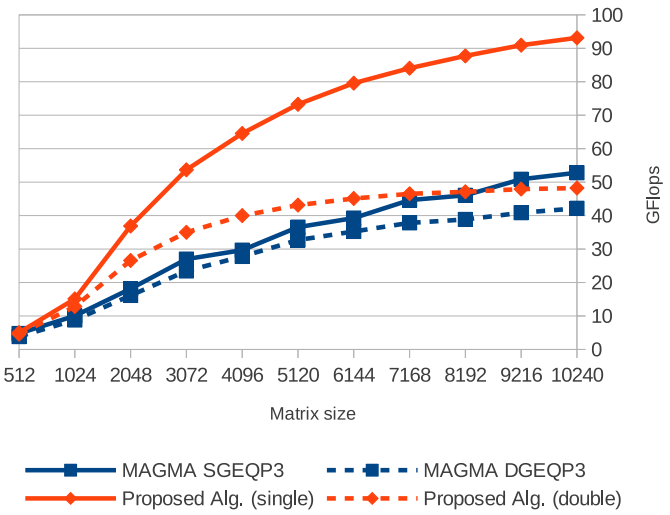

Fig. 5. Performance of the proposed GPU implementation and MAGMA implementation of `xGEQP3` on NVIDIA GeForce GTX480.

References

1. Bischof, C.H.: A parallel QR factorization algorithm with controlled local pivoting. SIAM J. Sci. Stat. Comput. 12, 36–57, 1991.
2. Chandrasekaran, S., Ipsen, I.C.F.: On rank-revealing factorisations. SIAM J. Matrix Anal. Appl. 15, 592–622, 1994.
3. Castaldo, A.M., Whaley, R.C.: Scaling LAPACK panel operations using parallel cache assignment. In 15th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 223-231, 2010.
4. 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, 2008.
5. Drmač, Z., Veselić, K.: New fast and accurate Jacobi SVD algorithm I. SIAM J. Matrix Anal. Appl. 29, 1322–1342, 2008.
6. Drmač, Z., Veselić, K.: New fast and accurate Jacobi SVD algorithm II. SIAM J. Matrix Anal. Appl. 29, 1343–1362, 2008.
7. Golub, G.H.: Numerical methods for solving linear least squares problems. Numer. Math. 7, 206–216, 1965.
8. Gu, M., Eisenstat, S.: Efficient algorithms for computing a strong rank-revealing QR factorization. SIAM J. Sci. Comput. 17, 848–869, 1996.
9. Quintana-Orti, G., Sun, X., Bischof, C.H.: A BLAS-3 version of the QR factorization with column pivoting. SIAM J. Sci. Comput. 19, 1486–1494, 1998.
10. Schreiber, R., van Loan, C.: A storage-efficient WY representation for products of Householder transformations. SIAM J. Sci. Stat. Comput. 10, 53–57, 1989.