GPU-Accelerated Implementation of the Storage-Efficient QR Decomposition

Peter Benner[1] Martin Köhler[2] Carolin Penke[3]

The LAPACK routines `GEQRT2` and `GEQRT3` can be used to compute the QR decomposition of a matrix of size $m \times n$ as well as the storage-efficient representation of the orthogonal factor $Q = I - VTV^T$. A GPU-accelerated algorithm is presented that expands a blocked CPU-GPU hybrid QR decomposition to compute the triangular matrix T. The storage-efficient representation is used in particular to access blocks of the matrix Q without having to generate all of it. The algorithm runs on one GPU and aims to use memory efficiently in order to process matrices as large as possible. Via the reuse of intermediate results the amount of necessary operations can be reduced significantly. As a result the algorithm outperforms the standard LAPACK routine by a factor of 3 for square matrices, which goes hand in hand with a reduced energy consumption.

Along with the LU decomposition, the QR decomposition [5] is one of the basic matrix factorizations used in many numerical linear algebra algorithms. Especially when orthonormal bases come into play, e.g. in least-squares problems, the QR decomposition is commonly involved. During the last decades many different strategies to compute the orthogonal matrix $Q \in \mathbb{R}^{m\times m}$ and the upper triangular matrix $R \in \mathbb{R}^{m\times n}$ from a general matrix $A \in \mathbb{R}^{m\times n}$ fulfilling $QR = A$ were developed. The most common ones are based on Householder reflections [6]. These algorithms represent the matrix Q as a product of orthonormal Householder matrices $H_i \in \mathbb{R}^{m\times m}$:

$$Q = H_1 \cdots H_n, \tag{1}$$

where $H_i = I_m - 2\frac{v_i v_i^T}{v_i^T v_i}$ and v_i is the i-th Householder vector. Typically Q is not stored explicitly but implicitly in factored-form representation, where the scaled Householder vectors v_i and the scalar factors $\tau_i = \frac{2}{v_i^T v_i}$ are stored explicitly [5, 1].

We consider the case where we are interested in a block partitioning of Q like

$$Q = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix}, \tag{2}$$

where $Q_{ij} \in \mathbb{R}^{\frac{m}{2}\times\frac{m}{2}}$. When Q is stored as the product of Householder matrices (1) in factored-form representation, the explicit setup of the matrix Q is required and the

[1]Computational Methods in Systems and Control Theory, Max Planck Institute for Dynamics of Complex Technical Systems, Sandtor-Str. 1, 39106 Magdeburg, Germany, benner@mpi-magdeburg.mpg.de

[2]Computational Methods in Systems and Control Theory, Max Planck Institute for Dynamics of Complex Technical Systems, Sandtor-Str. 1, 39106 Magdeburg, Germany, koehlerm@mpi-magdeburg.mpg.de

[3]Computational Methods in Systems and Control Theory, Max Planck Institute for Dynamics of Complex Technical Systems, Sandtor-Str. 1, 39106 Magdeburg, Germany, penke@mpi-magdeburg.mpg.de

blocks can only be accessed or applied afterwards. If only one or two blocks are required, e.g. to form a matrix-matrix product, this procedure results in a large overhead. The compact WY or VTV^T variants of the QR decomposition [2, 7, 5] can be used to reduce this overhead. Here, the Householder product (1) changes to

$$Q = I_m - VTV^T \overset{W=VT}{\underset{}{\overset{Y=V}{=}}} I_m - WY^T, \tag{3}$$

where $V = [v_1, \ldots, v_m]$ contains the Householder vectors v_i. The upper triangular matrix T can be computed from the Householder vectors and represents the accumulation of the Householder transformations. When it is additionally stored in explicit form it can be used to apply Q using level-3 BLAS operations, which are the foundation of fast linear algebra algorithms on current computer architectures. Furthermore, regarding the block partitioning of Q in (2), one can extract and apply blocks easily by using T and a partitioned V. The computation of the compact WY representation of the QR decomposition is part of LAPACK [1] in the `GEQRT3` routine. An improved parallel version, employing a Directed-Acyclic-Graph (DAG) for task scheduling, is part of the PLASMA library [3]. Parts of this algorithm have also entered the recent LAPACK 3.7 as the routine `GEQR`. The blocked variant of the QR decomposition typically makes use of the compact representation (3) of decomposed panels to update the trailing submatrix using matrix-matrix products. However, an equivalent of the `GEQRT3` routine, that also exploits the GPU's capabilities to compute the $n \times n$ triangular factor T, does not exist. The MAGMA library [8, 9] only provides QR decompositions which give the scalar factors $\tau_i = \frac{2}{v_i^T v_i}$ of the elementary reflectors or triangular matrices $T_i \in \mathbb{R}^{n_b \times n_b}$ (with n_b as the panel width) that represent the compact QR factorization of the individual panels. To build T from the T_i, a post-processing step would be necessary. Other GPU-accelerated libraries, such as ArrayFire [10], cuSOLVER[4], or CULATools[5], only support the classical representation.

In our contribution, we want to close this gap by presenting a GPU-accelerated approach to not only compute the QR factorization in factored-form representation but also provide T from the compact WY representation of the matrix Q.

We implement the blocked variant of the QR decomposition [5] as a CPU-GPU hybrid with additional operations to compute the T matrix. The matrix A is partitioned into panels. It generally resides in GPU memory during our computations. The current panel A_i is sent to the CPU to be factored:

$$A_i = Q_i R_i. \tag{4}$$

The LAPACK routines `GEQRT2` or `GEQRT3` can be used to compute T_i and V_i defining the compact representation of

$$Q_i = I - V_i T_i V_i^T. \tag{5}$$

The factored panel, consisting of V_i and R_i, and T_i are sent back to GPU memory. Here, T_i is used to update the trailing submatrix.

The following Lemma, which is proved by direct calculation, shows how T describing the compact representation of Q can be computed from the T_i given by the panel factorizations.

[4]http://docs.nvidia.com/cuda/cusolver/

[5]http://www.culatools.com/

Algorithm 1 Block Compact QR Decomposition with Reuse of VT^T

```
Require: $A \in \mathbb{R}^{m \times n}$
Ensure: $V, R, T, S$ such that $A = QR$ with $Q = I_m - VTV^T$, $S = VT^T$.
    $A$ is overwritten by $V, R$. $S, T$ can be stored together in an $m \times n$ array.
1: for $k = 1, \dots$ do
2:     $[R_k, V_k, T_{k,k}] \leftarrow QR(A_{k:p,k})$          ▷ by using GEQRT2 on the host
3:     Build new block column of $T$
           $$T_{1:k-1,k} \leftarrow -\underbrace{T_{1:k-1,1:k-1}V_{1:k-1}^T}_{=S_{1:k-1}^T} V_k T_{k,k}$$
4:     Update $S_{1:k-1}$ which currently holds $V_{1:k-1}T_{1:k-1,1:k-1}^T$
           $$S_{1:k-1} \leftarrow S_{1:k-1} + V_k T_{1:k-1,k}^T$$
5:     Build new block column of $S$
           $$S_k \leftarrow V_k T_{k,k}^T$$
6:     Update trailing submatrix of $A$
           $$A_{k:p,k+1:q} \leftarrow A_{k:p,k+1:q} - \underbrace{V_k T_{k,k}^T}_{=S_k} V_k^T A_{k:p,k+1:q}$$
7: end for
```

Lemma 1 *Let* $Q_1 = I_m - V_1T_1V_1^T \in \mathbb{R}^{m\times m}$ *and* $Q_2 = I_m - V_2T_2V_2^T \in \mathbb{R}^{m\times m}$ *with* $V_1 \in \mathbb{R}^{m\times j}$, $V_2 \in \mathbb{R}^{m\times n_b}$ *and* $T_1 \in \mathbb{R}^{j\times j}$, $T_2 \in \mathbb{R}^{n_b\times n_b}$. *Then*

$$Q_1Q_2 = I_m - V_+T_+V_+^T,$$

where

$$V_+ = \begin{bmatrix} V_1 & V_2 \end{bmatrix} \in \mathbb{R}^{m\times j+n_b}, \qquad T_+ = \begin{bmatrix} T_1 & -T_1V_1^TV_2T_2 \\ 0 & T_2 \end{bmatrix}.$$

While the QR factorization is being computed, we continuously build up the block columns of

$$T = \begin{bmatrix} T_{1,1} & \cdots & T_{1,q} \\ & \ddots & \vdots \\ 0 & & T_{q,q} \end{bmatrix}. \tag{6}$$

Lemma 1 gives

$$T_{1:k-1,k} = -T_{1:k-1,1:k-1}V_{1:k-1}^TV_kT_{k,k}, \tag{7}$$

where $V_{1:k-1}$ contains the Householder vectors of the previous panel factorizations and V_k contains the Householder vectors of the current panel factorization. We aim to implement this computation efficiently on the GPU employing cuBLAS routines. $T_{1:k-1,1:k-1}V_{1:k-1}^T$, or its transpose $V_{1:k-1}T_{1:k-1,1:k-1}^T$, is necessary to compute the k-th block column of T. It holds

$$\begin{aligned} T_{1:k,1:k}V_{1:k}^T &= (V_{1:k}T_{1:k,1:k}^T)^T \\ &= \begin{bmatrix} V_{1:k-1}T_{1:k-1,1:k-1}^T + V_kT_{1:k-1,k}^T & V_kT_{k,k}^T \end{bmatrix}^T. \end{aligned}$$

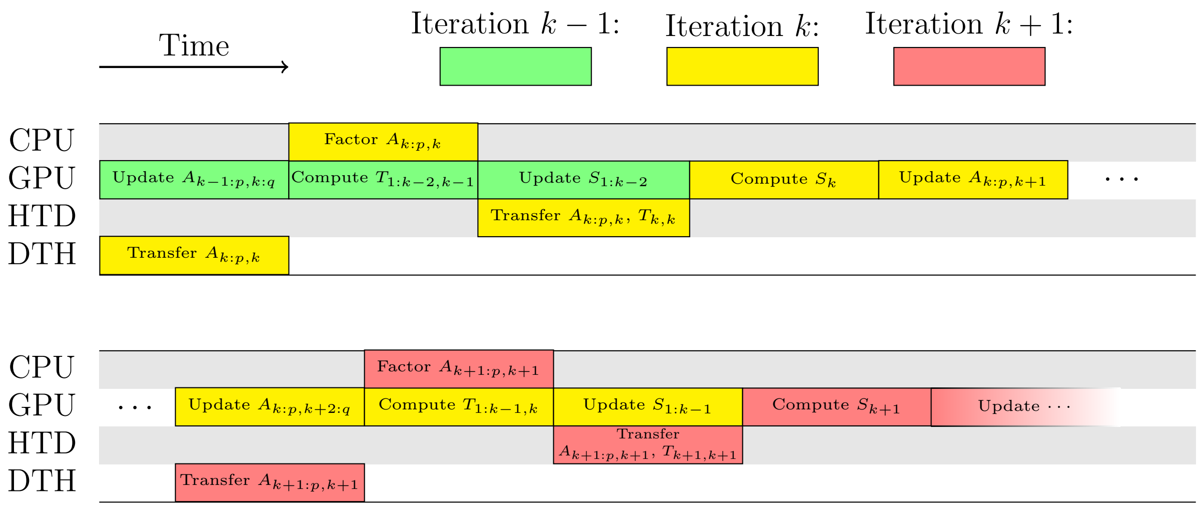

Figure 1: Course of events for the asynchronous version of Algorithm 1.

Table 1: Runtime in [s] for the CPU (LAPACK) and the GPU-accelerated version of the compact WY QR decomposition using a block width of 128.

Dimension	CPU	GPU	Speedup
1 000	0.050	0.048	1.04
2 000	0.228	0.120	1.90
3 000	0.583	0.212	2.75
4 000	1.139	0.335	3.40
5 000	1.984	0.551	3.60
6 000	3.183	0.852	3.76
7 000	4.668	1.260	3.70
8 000	6.373	1.762	3.62
9 000	8.532	2.404	3.55
10 000	11.181	3.202	3.49
11 000	14.504	4.115	3.52
12 000	18.067	5.182	3.49
13 000	22.211	6.566	3.38
14 000	27.064	8.003	3.38
15 000	32.319	9.768	3.31

We see that, if $V_{1:k-1}T^T_{1:k-1,1:k-1}$ is available from the computation associated to the previous panel, it can be updated and expanded to provide $V_{1:k}T^T_{1:k,1:k}$. This is much cheaper than recomputing $T_{1:k,1:k}V^T_{1:k}$ for every panel, which is why we expand our algorithm to successively compute $V_{1:k}T^T_{1:k,1:k}$. The reason for the transpose is the fact that $V_{1:k-1}T^T_{1:k-1,1:k-1}$ is lower trapezoidal and can therefore be stored together with T in an $m \times n$ array. The new block column $V_kT^T_{k,k}$ can also be used in the update of the trailing submatrix.

These ideas are implemented by Algorithm 1 which realizes a QR factorization that includes the computation of T and VT^T. The panel factorization is performed by the CPU. The following steps are the updates of T, VT^T and the trailing submatrix. They are performed on the GPU by a series of cuBLAS routines, so that leading zeros and triangular structures are exploited to reduce the total amount of operations. Only a minimal further work space of size $n_b \times n$ (n_b denoting panel width) is required.

Furthermore we employ asynchronous communication to overlap CPU and GPU work. Hence, while the GPU is still busy updating the remaining trailing submatrix, the panel is transferred between device and host and can already be factored by the CPU. This is visualized by Figure 1. In the optimal case, that is depicted here, it is possible to have the GPU working to full capacity.

For the performance evaluation of the algorithm we use a dual-socket Intel Xeon E5-

2640v3 system (16 cores, 64 GB RAM) and an Nvidia Tesla K20m accelerator. The results are given in Table 1, where all computations are done in IEEE double precision. Here, the CPU reference result is computed using `GEQRT3` from OpenBLAS 0.2.18. For the panel factorization in the GPU-CPU hybrid implementation we use `GEQRT2` which we evaluated to be the fastest variant in LAPACK for the panel decomposition on the host. The GPU code uses the cuBLAS library of CUDA 8.0 and the whole code is compiled using gcc 4.8. In order to avoid the tuning for a special matrix structure the input square matrices consist of random entries distributed uniformly between 0 and 1. In Table 1 we see that the speedup increases very fast with the growing problem dimension to a factor of up to 3.76 while using only one GPU. For a further increasing problem dimension we obtain a stagnation caused by the limited memory bandwidth.

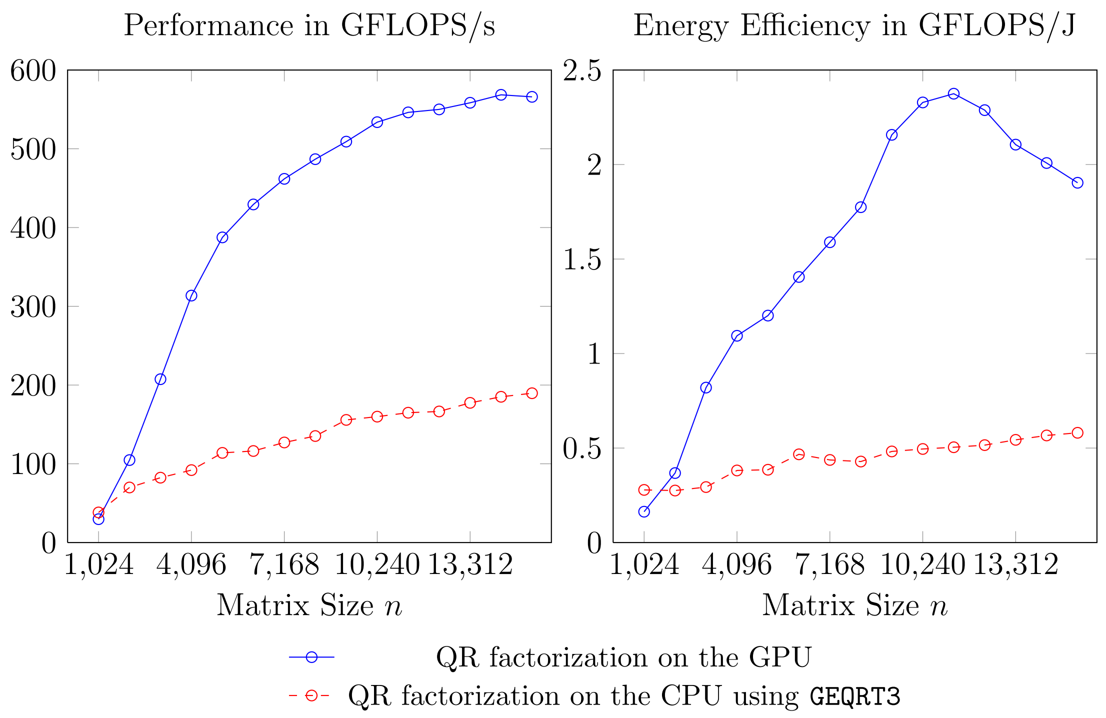

Figure 2: Comparison of approximate performance and energy efficiency of CPU-based and GPU-accelerated algorithm for square matrices.

The performance achieved by the GPU is also highlighted by Figure 2. We approximate the performance by assuming the number of FLOPs to be well represented by $\frac{5}{3}n^3$. This includes $\frac{4}{3}n^3$ from the QR factorizations and $\frac{1}{3}n^3$ from the computation of T ignoring lower order terms. The high performance is due to the GPU's capacities to perform many operations in parallel with little overhead. This also explains its higher energy efficiency which is depicted in the figure as well. Using the same amount of energy the GPU can perform up to 4.75 times as many operations as the CPU. This is achieved for large matrices with sizes of about 10000×10000, because here the GPU cores can be used to full capacity. To explain the following decline for even larger matrices further investigations are necessary.

We showed that the GPU is an excellent tool to compute the storage-efficient QR factorization. This is true with regards to classical FLOP performance as well as energy efficiency. An essential part of our implementation is the simultaneous computation of the VT^T matrix, which to our knowledge is new. To see how this approach compares to the conventional ones implemented in LAPACK [4], an experimental setup is required that implements both variants on the same computer system, i.e., on the GPU or on the CPU. Future research will investigate this relation and provide further insight into the efficiency of the new approach.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, PA, third ed., 1999.

[2] C. Bischof and C. Van Loan, *The WY representation for products of Householder matrices*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. S2–S13.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *Parallel tiled QR factorization for multicore architectures*, Concurr. Comput., 20 (2008), pp. 1573–1590.

[4] E. Elmroth and F. G. Gustavson, *Applying recursion to serial and parallel QR factorization leads to better performance*, IBM J. Res. Dev., 44 (2000), pp. 605–624.

[5] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, fourth ed., 2013.

[6] A. S. Householder, *The Theory of Matrices in Numerical Analysis*, Dover Publications (original Blaisdall 1964), New York, 1975.

[7] R. S. Schreiber and C. Van Loan, *A storage-efficient WY representation for products of Householder transformations*, SIAM J. Sci. Statist. Comput., 10 (1989), pp. 53–57.

[8] S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems*, Parallel Comput., 36 (2010), pp. 232–240.

[9] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, *Dense linear algebra solvers for multicore with GPU accelerators*, in Proc. of the IEEE IPDPS'10, Atlanta, GA, Apr. 2010, IEEE Computer Society, pp. 1–8.

[10] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, and J. Melonakos, *ArrayFire - A high performance software library for parallel computing with an easy-to-use API*, 2015.