PAQR: Pivoting Avoiding QR factorization

Wissam M. Sid-Lakhdar*, Sebastien Cayrols*, Daniel Bielich*, Ahmad Abdelfattah*, Piotr Luszczek*, Mark Gates*, Stanimire Tomov*, Hans Johansen‡, David Williams-Young‡, Timothy A. Davis†, Jack Dongarra*§
* University of Tennessee
† Texas A&M University
‡ Lawrence Berkeley National Laboratory
§ Oak Ridge National Laboratory

***Abstract*—The solution of linear least-squares problems is at the heart of many scientific and engineering applications. While any method able to minimize the backward error of such problems is considered numerically stable, the theory states that the forward error depends on the condition number of the matrix in the system of equations. On the one hand, the QR factorization is an efficient method to solve such problems, but the solutions it produces may have large forward errors when the matrix is deficient. On the other hand, QR with column pivoting (QRCP) is able to produce smaller forward errors on deficient matrices, but its cost is prohibitive compared to QR. The aim of this paper is to propose PAQR, an alternative solution method with the same cost (or smaller) as QR and as accurate as QRCP in practical cases, for the solution of rank-deficient linear least-squares problems. After presenting the algorithm and its implementations on different architectures, we compare its accuracy and performance results on a variety of application problems.**

***Index Terms*—Linear least-squares, QR factorization, QR decomposition, deficient matrix, rank-deficient, low-rank**

I. Introduction

a) Context: The solution of linear least-squares problems is at the heart of many scientific and engineering fields [1]. Formally, the problem is defined as:

$$\min_{x} ||Ax - b||_2 \tag{1}$$

where A is a (large) rectangular m-by-n matrix, b is the *right-hand side*, and x is the solution of the system.

While several methods exist [1]–[3] [4, ch.5] to solve such a problem, the *QR factorization* plays a key role. It corresponds to the factorization of the matrix A into:

$$A = QR, \tag{2}$$

where Q is an m-by-n orthonormal matrix, and R an upper triangular n-by-n matrix. Given such a decomposition, the solution x is obtained by multiplying the inverse of Q to b:

$$y = Q^T b \tag{3}$$

then solving the triangular system of equations:

$$x = R^{-1} y \tag{4}$$

b) Challenge: The matrices arising in practical scientific and engineering applications like Quantum Chemistry and Weighted Least-squares (as studied in Section V-A1) present multiple challenges. First, the increase in computational power of modern computers has led to the increase in size of the targeted matrices. This challenge has been addressed through the development of numerical libraries that take advantage of hardware accelerators, shared- and distributed-memory parallelism [5]–[7].

Second, in practice, matrices are often *rank-deficient* and *low-rank* [8]. In essence, some columns of the matrix are redundant, or more precisely, they can be expressed as linear combinations of other columns [9]. Moreover, the characterization of rank deficiency can be blurry because of numerical round-off errors. This is due to the limited precision floating-point arithmetic representation of real numbers. The implication is that, not a single, but an infinite number of potential solutions to the least-squares problem may exist. In such cases, although the QR factorization is a numerically stable operation [10, p. 384], the calculated solutions can be arbitrarily far from the true solution. This challenge has been addressed through the development of more robust methods, such as *QR with column pivoting* (QRCP) and *Rank-Revealing QR* (RRQR). Unfortunately, they are very expensive and even unpractical in large-scale settings.

c) Contribution: The two challenges mentioned above have lead us to rethink the traditional methods to solve large-scale deficient linear least-squares problems. We propose a new variant of the QR factorization that we denote *Pivoting Avoiding QR factorization* (PAQR). The guiding idea behind PAQR is that linearly dependant columns of the matrix can be detected and removed on the fly during the factorization process. This new variant yields accurate solutions without the expensive cost of column pivoting induced by QRCP.

The initial goal of PAQR was to be numerically more stable than QR on challenging problems while remaining as fast as QR. However, PAQR turns out to be at least as fast as QR on full-rank problems, but faster than QR on rank-deficient problems. Moreover, it can be empirically as accurate as QRCP on practical rank-deficient least-squares problems.

d) Overview: This paper is organized as follows: Section II describes the QR, SVD, QRCP and RRQR variants, as well as new randomized variants. Section III describes the PAQR algorithm together with its limitations. Section IV describes sequential, batched GPU, distributed-memory implementations of PAQR, focusing on data layout, computation and communication. Section V compares the numerical accuracy of PAQR with that of QR and QRCP, together with the performance of the methods on different computational settings. Section VI summarizes the work and discusses extension of this work that is currently being explored.

II. Background

a) QR: For computing a QR factorization, Householder orthogonalization is the standard implementation within the current state-of-the-art linear algebra libraries, LAPACK, ScaLAPACK for example. The overall flow of Householder orthogonalization follows that of Algorithm 1.

Algorithm 1 QR factorization

```
Input $A \in \mathbb{R}^{m\times n}$
Output $V \in \mathbb{R}^{m\times n}, R, \tau$
1: $V, R \leftarrow [0]$
2: for $i \leftarrow 1 \dots n$ do
3:     $V_{i:m,i}, R_{i,i}, \tau_i \leftarrow$ generate_reflector($A_{i:m,i}$)
4:     $R_{1:i-1,i} \leftarrow A_{1:i-1,i}$
5:     $A_{i:m,i+1:n} =$ apply_reflector($A_{i:m,i+1:n}, V_{i:m,i}, \tau_i$)
6: end for
```

First, the algorithm constructs a reflection (Line 3). There are a variety of methods for computing a reflection, such as Householder. Here, *generate_reflector* takes as input the column to be orthogonalized, and outputs the associated reflection $V_{1:m,i}$, the diagonal element $R_{i,i}$ and τ_i. Algorithm 1 shows a non-blocked variant, which is akin to that of a panel factorization within a blocked scheme. In general, it is better to block the factorization to take advantage of Level 3 BLAS routines. In that instance, once a panel has been factorized, the blocked variant updates the external trailing sub-matrix with the newly constructed block reflector. Once complete with this step, the algorithm would factorize the next panel and repeat.

b) QRCP: A plethora of variants of QR factorization exist that include *QR with Column Pivoting* (QRCP). An important one is *Rank Revealing QR* (RRQR) [11]–[16], as it may be proven to guarantee numerical properties even for rank-deficient matrices [17]. Rank-revealing is not limited to the QR factorization and it is also possible with the LU factorization [18], [19] and ULV decomposition [20].

Algorithm 2 presents a basic implementation of QRCP. It follows that of Algorithm 1 but prior to computing the orthogonalization step, the algorithm compares the 2-norm of each remaining column, pivoting the column with the remaining largest norm to the current iterate position. Although this algorithm is robust and can provide numerical accuracy (c.f. Section III) far greater than QR for rank-deficient matrices, it is far more expensive in terms of floating point operations, as well as a truly blocked variant of QRCP is not possible [21]. Once a reflection has been constructed, the column norms of the entire trailing matrix need to be updated so that the next pivot step can be correctly evaluated.

Algorithm 2 QRCP factorization

```
Input $A \in \mathbb{R}^{m\times n}$
Output $V \in \mathbb{R}^{m\times n}, R \in \mathbb{R}^{n\times n}, \tau, \pi$
1: $V, R, P \leftarrow [0]$
2: for $i \leftarrow 1 \dots n$ do
3:     $\pi_i \leftarrow \arg\max_{i\le j\le n} \|A_{i:m,j}\|_2$    ▷ find largest column norm
4:     if $\pi_i > i$ then    ▷ found larger column norm
5:         $A_{:,i} \leftrightarrow A_{:,\pi_i}$    ▷ swap columns
6:     end if
7:     $V_{i:m,i}, R_{i,i}, \tau_i \leftarrow$ generate_reflector($A_{i:m,i}$)
8:     $R_{1:i-1,i} \leftarrow A_{1:i-1,i}$
9:     $A_{i:m,i+1:n} \leftarrow$ apply_reflector($A_{i:m,i+1:n}, V_{i:m,i}, \tau_i$)
10: end for
```

The RRQR factorization of a matrix $A \in \mathbb{R}^{m\times n}$ represents

$$AP_c = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{bmatrix}, \tag{5}$$

where $Q \in \mathbb{R}^{m\times m}$ orthonoromal, $R_{11} \in \mathbb{R}^{k\times k}$, $R_{12} \in \mathbb{R}^{k\times(n-k)}$, and $R_{22} \in \mathbb{R}^{(m-k)\times(n-k)}$.

This factorization is said to be *rank revealing* [17], [22] if the following condition is satisfied

$$\sigma_{min}(R_{11}) \le \frac{\sigma_k(A)}{p(k,n)}, \;\; \sigma_{max}(R_{22}) \le \sigma_{k+j}(A)p(n,k) \tag{6}$$

where $p(n,k)$ is a low degree polynomial in k and n. This algorithm can reveal the rank of A but may not be stable enough because of the elements of $R_{11}^{-1}R_{12}$. To address this problem, a strong Rank Revealing QR factorization has been developed by M. Gu in [23].

c) SVD: The *Singular Value Decomposition* (SVD) [24] is an obvious method of choice for many numerical methods focusing on the variants of the least-squares problem [4, p. 327]. Among its other uses, SVD directly reveals the numerical rank of the matrix and allows selection of the cut-off threshold to admit only a subset of singular values as representative for the matrix. In least-squares problems specifically, SVD benefits from numerical stability and has been a subject of study that resulted in various algorithms that permit high levels of parallelism [25].

d) CARRQR: In the context of distributed-memory environment, the QR factorization is well-known to be communication sub-optimal. To address this issue and improve the performance, authors in [26] developed a Communication-Avoiding (*CA*) variant of the algorithm named CAQR. The idea is to process a panel in a different manner, which is to split the panel into block of rows, call a local QR factorization on each block and then uses a reduction tree to retrieve the final R associated with the panel. The *CA* approach was later extended to the RRQR algorithm to create the CARRQR algorithm. The problem of RRQR is its sequential nature as at each step, the column with the largest norm of the current matrix has to be moved that leading position. To tackle this problem, the authors in [27] use a tournament pivoting strategy to find the best k pivots in a single step. Once the pivots are moved to the leading position, the classical iteration of the blocked QR factorization is applied. The tournament pivoting is a reduction-tree operation. At each node, a matrix of $2k$ columns is factorized using RRQR and only the first k pivots are passed to the parent.

e) Approximate RRQR: The paper [13] by Bischof and Quintana-Ortí summarizes prior work revolving RRQR, while developing new blocked variants for computing an approximation of an RRQR factorization. The main idea is to apply RRQR within panels instead of the entire matrix. This allows for the use of efficient Level 3 BLAS routines. The approximation comes into play by being limited to scanning the column norms only within the current panel. So the selected pivot may not be of largest norm overall. If a column is *rejected*, i.e identified as a linear combination of the previous columns, it is pivoted to the end of the matrix. After computing an R_{11} factor from all panels, the rejected columns need to be reconsidered. Bischof and Quintana-Ortí choose to follow the traditional pivoting strategy proposed by Golub for finishing R_{11} on the rejected columns. Finally, R_{22} is constructed using traditional QR.

Other interesting approximate algorithms [28]–[31] follow the recent randomized linear algebra line of research. These state-of-the-art algorithms on average exhibit great improved performance compared to QRCP variants, while their performance compared with QR varies. In general, for full-factorization experiments, QR outperforms Randomized QRCP (RQRCP). Going further, truncated variants are proposed where experiments exhibit randomized methods outperforming traditional QR [30]. However, they still rely on actually pivoting columns.

III. Proposed algorithm

Two main metrics exist for evaluating the numerical accuracy of algorithms that deal with linear systems of equations and linear least-squares problems: *forward error* and *backward error*. The forward error is defined as [32]:

$$e_{fwd} = \frac{\|x - \hat{x}\|_p}{\|\hat{x}\|_p} \tag{7}$$

where $\hat{x}$ is the true solution and x is the computed solution, for a given p-norm. The backward error is defined as [33]:

$$e_{bwd} = \frac{\|Ax - b\|_p}{\|A\|_p \|x\|_p + \|b\|_p}. \tag{8}$$

Even though numerically stable algorithms are able to minimize the backward error on the residual, the forward error remains bounded by the backward error magnified by the 2-norm *condition number* κ_2 of the input matrix [9]:

$$\kappa_2(A) = \|A\|_2 \|A^+\|_2 = \frac{\sigma_{max}}{\sigma_{min}} \tag{9}$$

where $\|\cdot\|_2$ represents the 2-norm, A^+ is the pseudoinverse, and σ_{max} and σ_{min} are the largest and smallest singular values of A, respectively.

In finite-precision arithmetic, machine precision ϵ indicates the attainable accuracy and unit round-off error [10]. Matrix A is considered numerically rank-deficient iff:

$$\kappa_p(A) > \epsilon^{-1} \tag{10}$$

for a p-norm condition number.

The most accurate (but most expensive) way to solve a rank-deficient least-squares problem is to compute the SVD of A and truncate the smallest singular values, so as to avoid the condition given in Equation (10). However, a more common way to solve such problems is to use RRQR. Given the existing error bounds [17] linking the singular values of a matrix with the R matrix returned by RRQR, an early stopping criterion exists allowing to terminate the factorization prematurely once the remaining singular values of the trailing matrix are known to degrade the conditioning of R. However, the cost of column pivoting induced by this method makes it impractical at scale.

The intuition behind PAQR comes from the observation that a reduction of the condition number of a rank-deficient matrix can be achieved by detecting and removing on the fly columns contributing to the numerical deficiency of the matrix, during a standard QR factorization, without incurring any pivoting. These skipped columns are linearly dependent to the already processed columns on the left. They do not contribute to the linearly independent columns whose count represents matrix rank. For PAQR, we follow the convention used in [13] to define a column as *rejected* when it is identified as a linear combination of the previous columns. As the decision for skipping the columns is made on-the-fly, PAQR is responsive to the numerical properties of matrix data that change over the course of the algorithm steps, which is superior to the passive approach of the classic QR factorization,

The main difference between PAQR and all existing QRCP variants (such as RRQR) is that PAQR completely avoids any kind of pivoting, and thus, does not incur any additional data movement. While QRCP variants focus on choosing columns as pivots, PAQR instead focuses on flagging columns as rejected. Note that, in its current development, PAQR does not guarantee the same properties given by RRQR Equation (6). Indeed, post-processing may be needed on the R matrix in order to reveal the true rank of a matrix.

A. Algorithm details

PAQR is described in Algorithm 3. While its building blocks are the same as those of QR, it differs in three ways. First, at each iteration, PAQR computes a *deficiency criterion* (Line 5) (cf. Section III-B) to decide whether to flag the current column of A as rejected (Line 6). In which case, the algorithm proceeds immediately to the next iteration. Second, once columns are flagged as rejected, subsequent operations need to account for them. This is done by updating the indices of the trailing matrix (Line 9–10) using the index k instead of the original loop index i. k is increased only when a column is estimated to be linearly independent from previous ones (Line 11). Consequently, the size of the output matrices V and R will be smaller (Line 14–15).

Algorithm 3 PAQR factorization

```
Input $A \in \mathbb{R}^{m \times n}$
Output $V, R, \tau, \delta$
 1: $V, R \leftarrow [0]$
 2: $k \leftarrow 1$
 3: for $i = 1 \dots \min\{m, n\}$ do
 4:     $V_{k:m,k}, R_{k,k}, \tau_k \leftarrow$ generate_reflector($A_{k:m,i}$)
 5:     if $i^{th}$ column of $A$ is rejected then
 6:         $\delta_i \leftarrow$ TRUE(1)        ▷ skip the current column
 7:     else
 8:         $\delta_i \leftarrow$ FALSE(0)     ▷ include the current column
 9:         $R_{1:k-1,k} \leftarrow A_{1:k-1,i}$
10:         $A_{k:m,i+1:n} \leftarrow$ apply_reflector($A_{k:m,i+1:n}, V_{k:m,k}, \tau_k$)
11:         $k \leftarrow k + 1$
12:     end if
13: end for
14: $V \leftarrow V_{1:m,1:k-1}$
15: $R \leftarrow R_{1:k-1,1:k-1}$
```

Figure 1 shows an example of the result of the execution of PAQR with the second and fourth columns skipped due to the rank deficiency criterion.

The left matrix in Figure 1 represents the V and R matrices returned by PAQR relatively to the original columns of A. In most modern implementations, the QR factorization is done *in-place*, which means that the output V and R overwrite the original input A. The array of zeros and ones represent the δ vector returned by PAQR. This vector stores the flagged (rejected) columns of A.

The right matrix in Figure 1 represents a compressed (whether explicit or implicit) representation of V and R, where only the linearly independent columns are kept.

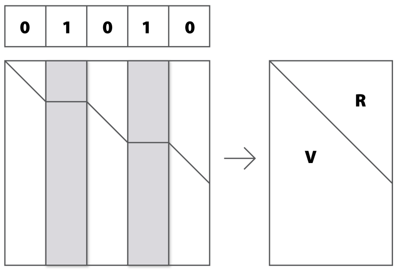


Fig. 1. Example of execution of PAQR factorization. Columns in grey are flagged as rejected.

B. Deficiency criteria

The decision of flagging a column of A to be ignored is a critical component of PAQR. In the literature, most criteria are costly, as they relate the singular value spectrum of a matrix. Hence, we present three criteria. The first criteria is standard (chapter 2 page 89 in [1]), the second and third are novel in that they are column-oriented. These last two do not necessitate the potential costly *a-priori* estimation of the largest singular value of the matrix, as proposed by criteria one. They rely on the observation that the QR factorization applies a series of well-chosen orthogonal projections, relative to the previously factorized columns of A on the trailing matrix. Thus, when we construct the associated Householder reflector of a column, its norm represents the projection of the corresponding original column of A on the space orthogonal to the sub-space spanning the previously linearly independent columns of A. If a vector is a linear combination of a set of other vectors, it should be included in the space spanned by these vectors. In practice, however, this condition can, and should, be relaxed, due to the existence of round-off errors with the limited precision arithmetic representation of the matrix elements.

The first (most costly) deficiency criterion requires the computation of the 2-norm of A.

$$|R_{k,k}| < \alpha * \|A\|_2 \tag{11}$$

Here $|R_{k,k}|$ is the norm of the Householder vector constructed at iteration k (and stored on the k^{th} diagonal of R), $\|A\|_2$ is the 2-norm of A, known to be the largest singular value of A (σ_1). The critical parameter α is *a priori* chosen as an input to the algorithm. Its value can be adjusted according to the numerical arithmetic precision (e.g. $\alpha \equiv \epsilon$). Given that the computation of this quantity can be expensive, a possible alternative to Equation (11) is the following:

$$|R_{k,k}| < \alpha \times \max_{1 \le j \le n} \|A_{:,j}\|. \tag{12}$$

As the column with the largest 2-norm of a matrix A is in general a good approximation of σ_1, the largest singular value [13].

The second (more simple) deficiency criterion we propose is:

$$|R_{k,k}| < \alpha \times \|A_{:,k}\| \tag{13}$$

where $A_{:,k}$ is the k^{th} column of A. The cost induced by this criterion corresponds to computing the norm of each column of A which is performed only once (at the beginning of the factorization). This is not the case for QRCP in which this operation is required at each iteration to guarantee the largest normed column is pivoted to the leading position. The idea for this deficiency criterion is that, once the previous Householder reflections have been applied, if the norm of the new constructed Householder reflection is small relative to its initial value, it is a linear combination of the previous $k - 1$ reflections and is thus rejected.

The third deficiency criterion we propose takes into account how the QR factorization operates: at iteration k, the k^{th} column of A was the target of updates from only the columns on the left. Computing the largest singular values spanning the initial k columns of A is infeasible, we propose the following criterion only in conjunction with the approximation mentioned above:

$$|R_{k,k}| < \alpha \times \max_{1 \le j \le k} \|A_{:,j}\|. \tag{14}$$

C. Limitations

Although in practical applications PAQR is able to reduce the deficiency of a matrix enough to drastically improve the forward error of a rank-deficient least-squares system, we present a synthetic corner case on which PAQR does not detect and remove any column but for which the forward error still grows beyond control of our proposed criteria. We denote this family of generated matrices as *Cliff* matrices, due to the pattern of their singular value spectrum (mostly constant until a suddenly dropping-off from a numerical "Cliff" for the smallest singular values). We define them formally as:

$$\text{Cliff}(m,n,\alpha)_{i,j} = \begin{cases} \sqrt{\frac{1-(\max(m,n)\times\alpha)^2}{j-1}} & \text{if } i<j \\ \max(m,n)\times\alpha & \text{if } i=j \\ 0 & \text{if } i>j \end{cases} \quad (15)$$

An example of this matrix in practical application corresponds to the matrix Gks (see Section V)

Given the uniqueness of the QR decomposition of a matrix (up to the signs of the diagonal elements of R), the product of any orthonormal matrix by this Cliff matrix will generate a QR factorization whose R factors are the Cliff matrices. By construction, the norm of each column of a Cliff matrix is 1. Moreover, deficiency criteria (13) and (11) are both violated at every iteration of PAQR. This means that no column will be flagged as rejected, and that the PAQR factorization will be equivalent to the QR factorization. Unfortunately, from the experimental point of view, the forward error grows with n (the number of columns of a Cliff matrix) relative to the least-squares solution. The accumulation of errors in the computed solution from either QR or PAQR results in NaN (Not-a-Number). Excluding such edge cases, PAQR seems to deliver stable results with high numerical accuracy in practical application cases, as we show in Section V.

IV. Implementations

We propose several implementations of PAQR, each of which targeting a different computer architecture: sequential implementation, designed in LAPACK (Section IV-A), batch GPU implementation, designed in MAGMA (Section IV-B), and distributed-memory implementation, designed in ScaLAPACK (Section IV-C). Note for the remaining of the paper, we are going to use Householder orthogonalization as our method for constructing reflectors.

A. LAPACK (sequential)

A current prerequisite of PAQR (compared to QR) is the computations of the column norms of A. Future work can benefit from *randomized SVD* or iterative methods to quickly approximate the 2-norm of A (in $O(n^2)$ operations). As of now however, we consider (regardless of the selected α) the computation of the 2-norm of each column of A.

Modern linear algebra libraries rely on *blocking* strategies to optimize their performance. While memory-bound Level 1 BLAS (scalar-based) and Level 2 BLAS (vector-based) operations occur within limited size blocks (e.g., panels), the majority of the operations is described through and carried out by highly efficient computation-bound Level 3 BLAS (matrix-based) operations.

Inside a panel, care must be taken during the computation of Householder vectors, as the LAPACK implementation applies a post-processing in case the computed norm of the Householder vector is smaller than a machine-related precision threshold. For this reason, the PAQR deficiency criterion is checked before the potential application of this post-processing, as the computation of the Householder norm may be artificially inflated by an appropriate adaptive scaling factor.

Moreover, in the presence of previously detected rejected columns, the number of effective Householder vectors computed and stored is smaller than the current number of columns of A in the panel. Hence, every new Householder vector would need to be stored at a different location in memory than its originally intended one. Traditionally, the computation of the Householder vectors requires the loading of a sub-column (of A) in registers, its scaling, then its storing at the same location in memory (in V). In our implementation, we avoid an unnecessary copy of this vector to its final destination by storing the result of its scaling at its final correct location right away. This boils down to merging the `xSCAL` and `xCOPY` Level 1 BLAS operations into a `xSCALCOPY` routine.

Outside a panel, because the Householder vectors are stored contiguously in memory, the efficient `xLARFT` and `xLARFB` routines can still be used to build the blocking factor T matrix and update the trailing matrix, respectively. The potentially smaller number of Householder vectors to be applied should be the major source of speedup of PAQR over QR.

While the V matrix can be packed on the fly using the above-mentioned optimization, the R matrix is sparse however. Indeed, the rejected columns are flagged but remain present inside the R matrix. Hence, two strategies may be applied to use this sparse R matrix for the least-squares solution phase. The first one is to compact R, either during the factorization, or as a post-treatment. The drawback, however, is the extra memory traffic of potentially the whole R matrix, even in the presence of a single rejected column, for example, the second column of A. The second strategy is to keep R sparse, but develop a tailored `xTRSM` routine that can accommodate this sparsity pattern.

Figure 1 depicts a representation of R at the end of a PAQR factorization using the first strategy (right) and the second one (left).

B. MAGMA (batched GPU)

For GPU architectures, a kernel is developed to operate on a large set of relatively small matrices in parallel, which is often called *a batch setting*. The kernel takes as input an array of pointers that belong to independent matrices of the same size. Each matrix $A \in \mathbb{R}^{m\times n}$ is assumed to satisfy $m \geq n$. The corresponding output is $RV_{m\times\hat{n}}$ such that $\hat{n} \leq n$. Each RV matrix is condensed such that the $\hat{n}$ columns are adjacent to each other and aligned to the left of the matrix. The input matrices must be of the same size, but can have different degrees of rank deficiency. An additional output is δ, the array of flags of each matrix that point to the column indices that have been ignored during the factorization.

The batch PAQR implementation uses one kernel to perform the whole factorization. Each matrix is assigned to one thread-block. The main advantage of this approach is the optimal memory traffic, since each matrix is read and written exactly once. The implementation concerns small matrices that fit in the GPU shared memory. While the register file provides a faster data access, the shared memory is more flexible, and is our choice for the kernel design. Since detecting rank deficiency occurs at run time, it is non-trivial to maintain constant compile-time indexing of the register file, which is necessary to avoid register spilling. The kernel implements Algorithm 3 in an unblocked manner, which means that the application of the Householder reflectors are performed one column at a time, thus eliminating the need for constructing the T factor. The kernel works with any number of threads in the range $[n{:}m]$.

There are two main bottlenecks of the design. The first one is computing the norm of the current column. The second is the matrix-vector multiply (v^TA) during the application of the elementary Householder reflector $(I - \tau vv^T)A$, which requires a reduction across the columns of A. At each iteration, the norm of the current column is computed using a standard tree reduction in the shared memory of the GPU. If the computed norm is less than a given threshold (defined by the user through the kernel interface), the whole iteration is skipped and the corresponding flag is set. Otherwise, the kernel proceeds with the application of the Householder reflector to the trailing submatrix. The update step begins with the computation of the v^TA product, for which threads re-organize themselves evenly across the remaining columns of A, and an equivalent number of independent tree reductions are executed all in parallel in shared memory. The output of the product is scaled with τ and stored in a shared memory vector Y, which is used to compute the remaining rank-1 update ($A = A - v \times Y$). As an example, if 64 threads are used to factorize a 128×8 matrix, the first iteration begins with a tree reduction using all 64 threads to compute the norm of 128-element vector. The 64 threads are then reorganized into 7 groups of 9 threads, with one thread remaining idle, to compute the v^TA product. In this case, 7 independent tree reductions are performed, each one using 9 threads to reduce a 128-element vector. Out of each group, one thread writes the corresponding element of Y. The rank-1 update is performed using one thread per row. If there are more rows than threads, a round-robin scheme is used. The kernel interface exposes two important tuning parameters to the user. The first is α, the parameter used in the deficiency criteria, which controls the numerical behavior, and also affects the performance, of the batch PAQR kernel. The second is the number of threads used in the factorization, which controls the occupancy and the performance of the kernel as well.

C. *ScaLAPACK (distributed-memory)*

ScaLAPACK has a similar structure to that of LAPACK, in the sense that the high-level routines resemble a sequential implementation, while they rely on lower-level libraries to handle inter-process communication seamlessly.

A first major difference of this distributed-memory implementation, compared to a sequential or shared-memory one, is that A is distributed over the MPI processes following a *2D-block-cyclic* scheme as shown in Figure 2. Given the characteristics of PAQR (some Householder vectors may contain more rows than would have otherwise been the case in QR), the communication pattern of the factorization might differ. Indeed, now, some processes may be involved in the communication and computation relative to a panel while these processes would not have been involved otherwise. However, this difference does not increase the overall volume of communication and computation.

A second major difference is that the block of Householder vectors computed within each panel is broadcast from the set of processes mapped on it to the set of processes mapped on the trailing matrix. While the number of vectors to be communicated is deterministic in QR (corresponding to the size of the panel) this number is dynamic in PAQR, as it depends on the number of rejected columns encountered, and should be communicated alongside the Householder vectors. While the reduction in computation is the major source of speedup of PAQR over QR in a shared-memory environment, the reduction in communication volume of the Householder vectors should be an important source of speedup as well in a distributed-memory environment.

Figure 2 depicts a representation of R and V at the end of a PAQR factorization.

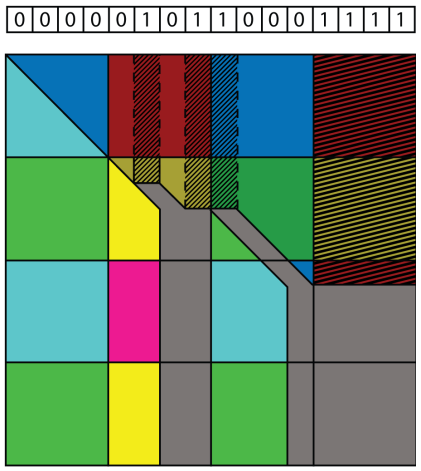


Fig. 2. PAQR on a 2D-block cyclic matrix in distributed-memory environment. The dark colors correspond to R. The light colors correspond to V. Every color (blue, red, green, yellow) correspond to a different process in the grid. The dashed area corresponds to rejected columns in R. The grey areas correspond to unused space freed-up by PAQR.

V. Experiments

This section presents the experiments comparing PAQR with QR and QRCP.

A. Experimental setting

1) Matrices: Three sets of matrices are used: (a) Test matrices; (b) Weighted least-squares (WLS) matrices; (c) Quantum many-body matrices.

a) Test matrices: The first set of matrices summarized in Table I is the same as the test matrices used in [27] for the validation of *Communication Avoiding Rank Revealing QR* (CARRQR). The *Rand* and *Vandermonde* matrices are added, while the *Gks*, *H-C*, *Scale* and *Kahan* matrices are missing.

No.	Matrix	Description
1	Rand	rand function in MATLAB generating random matrices.
2	Vandermonde	vander function in MATLAB generating Vandermonde matrices.
3	Baart	Discretization of the 1^{st} kind Fredholm integral equation [34].
4	Break-1	Break 1 distribution, matrix with prescribed singular values [35].
5	Break-9	Break 9 distribution, matrix with prescribed singular values [35].
6	Deriv2	Computation of second derivative [34].
7	Devil	The devil's stairs, a matrix with gaps in its singular values [36].
8	Exponential	Exponential Distribution, $\sigma_1 = 1$, $\sigma_i = \sigma_i - 1$ $(i = 2, ..., n), \alpha = 10 - 1/11$ [35].
9	Foxgood	Severely ill-posed test problem [34].
10	Gks	An upper-triangular matrix whose j-th diagonal element is $1/\sqrt{j}$ and whose i, j element is $-1/\sqrt{j}$, for $j > i$ [37], [38].
11	Gravity	1D gravity surveying problem [34].
12	H-C	Matrix with prescribed singular values, see description in [39].
13	Heat	Inverse heat equation [34].
14	Phillips	Phillips' famous test problem [34].
15	Random	Random matrix $A = 2 \times \text{rand}(n) - 1$ [38].
16	Scale	Scaled random matrix, a random matrix whose i-th row is scaled by the factor $\theta i/\theta$ [38]. We choose $\theta = 10\cdot$.
17	Shaw	1D image restoration model [34].
18	Spikes	Test problem with a "spiky" solution [34].
19	Stewart	Matrix $A = U\Sigma V^T + 0.1\sigma 50 \times rand(n)$ [36].
20	Ursell	Integral equation with no square integrable solution [34].
21	Wing	Test problem with a discontinuous solution [34].
22	Kahan	Kahan matrix.

TABLE I
Test matrices used in Section V

b) Weighted least-squares (WLS) matrices: The second set of matrices consists of Vandermonde-like matrices that can be used for interpolating 3D polynomials from scattered data using an application of the *weighted least-squares* (WLS) algorithm [40]. These WLS interpolation matrices are derived from finite-volume discretizations on irregular meshes, where m cells, each with n geometric moments, are used to calculate "stencils" X that determine polynomial coefficients:

$$W\,A\,X \approx W\,I\,. \tag{16}$$

Note that this is a least-squares system with multiple solutions $X \in \mathbb{R}^{n\times n}$ for multiple right-hand sides with $W \in \mathbb{R}^{m\times m}$ being a diagonal weight matrix. The weight matrix is designed to decay rapidly to emphasize closer values over distant ones in the interpolation, which can create very small row scaling. This results in least-squares systems that can have very poor conditioning, due to small weights and cells being arbitrarily small or close together, which can create matrix entries beyond the limits of floating point precision: $O(\gamma^m)$, for any $\gamma > 0$. If there are insufficient or co-linear cells, this may also prevent determination of some of the coefficients, making the interpolation matrix A rank-deficient. Finally, to maintain uniform matrix size for the entire batch, missing interpolation data are replaced with zero-padded rows, which may occur in any part of the matrix.

c) Quantum many-body matrices: The third set of matrices relates to quantum many-body problems. Fundamentally, solution of these problems for molecular systems requires manipulation of a high-dimensional tensor which describes the interactions between electrons: the Coulomb tensor, $g_{pq,rs}$, $p, q, r, s \in [0, n)$. The Coulomb tensor is leveraged in virtually all quantum chemistry methods, ranging from mean-field, single-body methods such as Hartree-Fock and density functional theory, to highly accurate many-body methods such as coupled cluster theory, Møller-Plesset perturbation theory, and configuration interaction to name a few. Despite it's large dimension, which grows $O(N_A^2)$ with system size (N_A), the Coulomb tensor is inherently low-rank, and can be straightforwardly shown to exhibit a matrix rank which grows $O(N_A)$ with system size when expressed in an atom-centered basis. This low-rank character has sparked n enormous research effort dedicated to the construction and manipulation of data-sparse representations of the Coulomb tensor, ranging from approximate projection methods such as density fitting (DF) and resolution of the identity (RI), grid-based methods, local-orbital methods, and more recently, formal matrix methods such as the Cholesky factorization and hierarchical decompositions [41]. Projection, grid, and local orbital methods have the benefit of exhibiting a relatively low communication overhead but do not produce the most compact representations. Matrix factorizations generally produce much more compact representations, but are accompanied by a much higher computational complexity and communication requirement which often complicates their usage on massively parallel architectures. Here, we examine the application of PAQR to produce low-rank representations of a representative set of Coulomb tensors generated from a range of molecular systems.

The Coulomb tensor has a natural matrization in $\mathbb{R}^{n^2\times n^2}$ by combining adjacent indices $g_{pq,rs} \rightarrow g_{i,j}$, $i, j \in [0, N)$ with $N = n^2$. For real basis discretizations, the column rank of the g matrization is bounded from above by $\frac{n}{2}(n-1)$ due to the fact that $g_{pq,rs} = g_{pq,sr}$. We have applied our methodology to three molecular test cases, all calculations were carried out within the NWChemEx program using either the 6-31G [42], [43] or 6-31G(d) [44] atom-centered Gaussian basis set. These cases are: a Uracil trimer (36 atoms, $N = 57,600$ within 6-31G), 5-mer (60 atoms, $N = 160,000$ within 6-31G), and the Beta Carotene molecule (96 atoms, $N = 506,944$ within 6-31G(d)).

B. Experimental results

1) Accuracy: Table II summarizes the numerical accuracy results of PAQR compared to that of QR and QRCP. These results are obtained with our MATLAB implementation of PAQR using the set of Test matrices Table I. The QR and QRCP implementations are the MATLAB's own version of the corresponding algorithms and were invoked as [q,r] = qr(A)

TABLE II
ACCURACY RESULTS COMPARING PAQR WITH QR AND RRQR BASED ON FORWARD, BACKWARD, AND ORTHOGONALITY ERRORS ON THE SET OF TEST MATRICES. FOR DOUBLE PRECISION ARITHMETIC, ERROR SHOULD BE AROUND 10^{-16}.

Matrix	$\kappa_2(A)$	Forward error			Backward error			Orthogonality error			*Rncol*	rank(R)	
		QR	PAQR	QRCP	QR	PAQR	QRCP	QR	PAQR	QRCP	PAQR	PAQR	SVD
Random*	10^{+03}	10^{-14}	10^{-13}	10^{-14}	10^{-16}	10^{-15}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	1000	1000	1000
Rand*	10^{+04}	10^{-13}	10^{-12}	10^{-12}	10^{-16}	10^{-15}	10^{-16}	10^{-15}	10^{-16}	10^{-15}	1000	1000	1000
Deriv2	10^{+06}	10^{-09}	10^{-08}	10^{-10}	10^{-15}	10^{-14}	10^{-16}	10^{-15}	10^{-15}	10^{-14}	1000	1000	1000
Stewart*	10^{+06}	10^{-11}	10^{-10}	10^{-11}	10^{-16}	10^{-16}	10^{-16}	10^{-15}	10^{-15}	10^{-15}	1000	1000	1000
Phillips	10^{+10}	10^{-07}	10^{-06}	10^{-06}	10^{-16}	10^{-15}	10^{-16}	10^{-15}	10^{-15}	10^{-14}	1000	1000	1000
Break-1*	10^{+11}	10^{-05}	10^{-04}	10^{-05}	10^{-16}	10^{-15}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	1000	1000	1000
Break-9*	10^{+11}	10^{-05}	10^{-04}	10^{-04}	10^{-16}	10^{-15}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	1000	1000	1000
Ursell	10^{+13}	10^{-03}	10^{-03}	10^{-01}	10^{-16}	10^{-15}	10^{-15}	10^{-15}	10^{-14}	10^{-15}	1000	999	999
H-C*	10^{+13}	10^{-04}	10^{-03}	10^{-01}	10^{-16}	10^{-15}	10^{-14}	10^{-15}	10^{-16}	10^{-14}	1000	999	999
Scale*	10^{+17}	10^{-12}	10^{+00}	10^{+00}	10^{-16}	10^{-16}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	914	794	802
Kahan	10^{+17}	10^{+01}	10^{-02}	10^{-02}	10^{-16}	10^{-14}	10^{-16}	10^{-14}	10^{-14}	10^{-15}	999	999	999
Baart	10^{+19}	10^{+02}	10^{+01}	10^{+01}	10^{-17}	10^{-15}	10^{-14}	10^{-14}	10^{-15}	10^{-14}	163	16	13
Devil*	10^{+19}	10^{+01}	10^{+00}	10^{+00}	10^{-16}	10^{-15}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	469	421	440
Exponential*	10^{+19}	10^{+02}	10^{+00}	10^{+00}	10^{-17}	10^{-15}	10^{-14}	10^{-14}	10^{-15}	10^{-14}	152	136	140
Foxgood	10^{+21}	10^{+03}	10^{+00}	10^{+00}	10^{-17}	10^{-16}	10^{-14}	10^{-13}	10^{-14}	10^{-11}	71	31	30
Gks	10^{+21}	10^{+280}	10^{+280}	10^{-02}	10^{-19}	10^{-19}	10^{-15}	10^{+262}	10^{+262}	10^{-15}	1000	999	999
Gravity	10^{+20}	10^{+03}	10^{+00}	10^{+00}	10^{-18}	10^{-16}	10^{-15}	10^{-14}	10^{-14}	10^{-12}	152	44	45
Shaw	10^{+20}	10^{+03}	10^{+00}	10^{+00}	10^{-18}	10^{-17}	10^{-16}	10^{-13}	10^{-15}	10^{-12}	77	19	20
Spikes	10^{+20}	10^{+03}	10^{+02}	10^{+01}	10^{-18}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	10^{-14}	56	31	31
Wing	10^{+21}	10^{+05}	10^{+01}	10^{+01}	10^{-20}	10^{-16}	10^{-15}	10^{-13}	10^{-14}	10^{-12}	32	8	8
Vandermonde	10^{+22}	10^{+70}	10^{+00}	10^{+00}	10^{-18}	10^{-15}	10^{-15}	10^{+54}	10^{-15}	10^{-11}	103	42	43
Heat	10^{+232}	10^{+215}	10^{+00}	10^{+00}	10^{-230}	10^{-15}	10^{-14}	10^{-15}	10^{-15}	10^{-13}	987	588	588

and [q,r,p] = qr(A), respectively. All matrices are generated of size 1000×1000. For each matrix A, random solution vectors $\hat{x}$ are generated. The corresponding right-hand sides b are computed as $b = Ax$. This ensures the existence of a valid solution to the systems of equations $Ax = b$. The triangular solve routine (xTRSM) is used on the R factors returned by the three variants. While QR returns a full R factor, PAQR and QRCP return a truncated R factor, that is supposed to only capture the rank (as well as possible).

The forward and backward errors are already defined in Equation (7) and Equation (8), respectively. The *orthogonality error* corresponds to:

$$\frac{||A^T(Ax - b)||_2}{||A||_2^2} \tag{17}$$

This metric is more suited to least-squares problems on deficient matrices compared to the backward error which is more appropriate for systems of linear equations.

Among the 22 test matrices, seven are full-rank (*Rand, Break-1, Break-9, Deriv2, Phillips, Random, Stewart*), while the others involve varying degrees of rank deficiency.

From our preliminary tests, we notice that all the deficiency criteria presented in Section III-B give us similar numerical results except for the GKS matrix. Therefore, in the remaining of the paper we use the deficiency criterion from Equation (13) and, unless otherwise mentioned, we set $\alpha = m \times \epsilon$, where m is the row-space dimension of the matrix.

First, all methods on all matrices have a backward error of the order of (or smaller than) the machine precision ϵ. This means that all methods correctly minimize the least-squares error of the system.

Second, on the set of full-rank matrices, all methods have similar forward errors. We note a slight discrepancy of no more than an order of magnitude in disfavor of PAQR. This is attributed to the fact that our MATLAB PAQR implementation differs from the MATLAB QR and QRCP implementations. For instance, we notice a similar order of magnitude difference when switching between different variants of Householder vector computation.

Third, the key result of this paper, on the set of rank-deficient matrices, PAQR exhibits a much more stable behaviour than QR and similar behavior than QRCP in terms of forward error. For instance, we can observe stark differences between PAQR and QR on the *Vandermonde* and *heat* matrices.

Fourth, as expected, PAQR does not remove any column from R when A is full-rank. More importantly, when A is rank-deficient, the number of retained (non-flagged) columns remains greater than the rank of A. For instance, in the case of the *heat* matrix, PAQR only flags 13 columns, while the true rank of the matrix is 588. Nevertheless, the removal of these few columns is still enough to lead to an accurate solution (forward error of 1.01 compared to $1.40 \times 10^{+215}$ with QR).

Among these matrices, we want to discuss two that are of interest: Gks, and Scale. The Gks matrix is an example of the patholigical cases presented in Section III-C. The rank of this matrix is 999 and its spectrum reveals that the first $n - 1$ singular values range from 26.1 and 0.041 while the smallest singular value is equal to 6.6×10^{-20}. In that case, as expected PAQR is not able to detect the single column that QRCP permutes at the end and then is removed before the call to TRSM. Note that when using the criterion one (more strict), PAQR gives similar results as QRCP. The Scale matrix is a perfect illustration that in some cases QRCP can fail to reveal the rank. Here, the spectrum does not have a gap and so the numerical rank 802 is more sensitive to small roundoff errors and approximations. Therefore, the truncation based on the diagonal elements of R causes both PAQR and QRCP to fail, whereas the classical QR factorization does not.

Post-treatment: One might think that the linearly dependent columns identified by PAQR (namely δ_{PAQR}) could be retrieved by *a posteriori* applying the PAQR deficiency criterion on the diagonal elements of R (namely δ_{QR}). To verify this statement, we compare the forward errors obtained after performing a QR factorization on A when: keeping all columns of A; discarding δ_{PAQR}; or discarding δ_{QR}. Table III shows that while the above mentioned statement holds true in the case of the heat matrix, it is incorrect for the two other matrices. Indeed, it does improves the forward error compared to no post-treatment but it does not reach the same numerical stability brought by PAQR. This is due to the post-treatment removing too many columns from A as displayed by $Rncol$. For example, in the case of the Vandermonde matrix, the number of non-rejected columns is reduced to 16 while its rank is 43.

The advantage of PAQR over QR is that the additional post-treatment is not necessary for PAQR for the solution of least-squares problems, while it is in the case of QR. Moreover, in a rank-revealing scenario, even if a post-treatment is needed in both methods, the fact that some columns have already been removed from the R of PAQR lead to less computation than QR for the post-treatment.

	$qr(A)$	$qr(A(:,\sim\delta_{PAQR}))$		$qr(A(:,\sim\delta_{QR}))$	
Matrix	e_{fwd}	e_{fwd}	$Rncol$	e_{fwd}	$Rncol$
Vandermonde	10^{+74}	10^{+00}	143	10^{+13}	6
Heat	10^{+214}	10^{+00}	988	10^{+00}	988
Spikes	10^{+03}	10^{+01}	76	10^{+10}	35

TABLE III
COMPARISON OF THE FORWARD ERROR WHEN COLUMNS OF A ARE REMOVED EITHER FROM PAQR OR FROM A POST-TREATMENT OF THE R MATRIX RETURNED BY THE CLASSICAL QR FACTORIZATION WITH THE FORWARD ERROR OBTAINED WHEN USING THE CLASSICAL QR FACTORIZATION.

2) *Efficiency:*

a) LAPACK: We now compare the performance of PAQR with QR and QRCP in the LAPACK implementation. Given that this implementation is sequential, we want to highlight the importance of the location of the rejected columns in the matrix. To this end, we generate random matrices of size $10,000 \times 10,000$ with the following characteristics: A_{full}, full-rank; A_{beg}, where the first 5000 columns are set to zero; A_{mid}, where the middle 5000 columns are set to zero; A_{end}, where the last 5000 columns are set to zero.

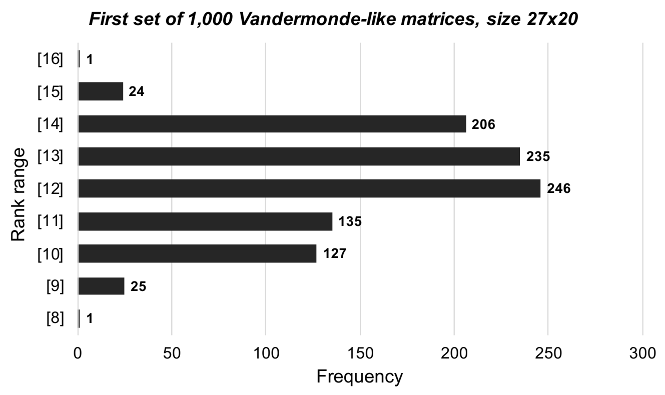
Table IV summarizes the runtimes on one core of a DGX A100 [45] server. We use: 1 *AMD EPYC 7742* CPU core; *MKL 2019.0.3* library as the Level i BLAS mplementation; *GCC 7.3.0* compiler. As expected, the runtime of PAQR is similar to that of QR on the full-rank matrix. Moreover, on the rank-deficient matrices, on matrices of same size and same amount of rejected columns, the performance of PAQR improves with more rejected columns appearing at the beginning of the matrix.

Method	Time (seconds)			
	A_{full}	A_{beg}	A_{mid}	A_{end}
QR	138			
PAQR	138	129	89	43
QRCP	163	211	209	211

TABLE IV
LAPACK PERFORMANCE. IMPACT OF THE LOCATION OF REJECTED COLUMNS ON PAQR FACTORIZATION PERFORMANCE.

b) Batch PAQR factorization on GPUs: We tested the GPU-based batch PAQR factorization on two sets of the set of WLS matrices. Since the current kernel design caches the entire matrix in the shared memory of the GPU, we show only two matrix sizes which satisfy this condition. For each set, 1000 matrices of the same size, but different ranks, were tested. Figure 3 shows the corresponding histograms of the number of non-flagged columns as detected by the batch PAQR kernel. Table V shows the performance of the batch PAQR factorization on an NVIDIA Tesla A100 GPU and on the AMD Instinct MI100 GPU. The results correspond to experiments conducted in double precision. We use cuBLAS and hipBLAS as the two reference implementations for a standard batch QR factorization. The performance of these two libraries does not take advantage of any rank deficiency in the matrices, so their performance is oblivious to this property. We show the timings of a regular QR kernel of our design qr_gpu for full rank matrices that are randomly generated, and then the paqr_gpu kernel timing on the WLS set of matrices. The paqr_gpu kernel is superior to the reference kernels in every test case.

The following results are obtained for the 27×20 and 125×56 matrices, respectively. On the A100 GPU, the speedups against cuBLAS increase from $2.7\times$ (resp. $2.2\times$) for qr_gpu to $2.9\times$ (resp. $2.3\times$) for paqr_gpu. Note that varying timings for paqr_gpu could be observed based on the pattern of rank deficiency, but it should never be slower than qr_gpu. On the MI100 GPU, the speedups against hipBLAS increase from $7.8\times$ (resp. $1.3\times$) for qr_gpu to $8.7\times$ (resp. $1.4\times$) for paqr_gpu. In order to justify the significant speedups against the vendor libraries for the full rank case, we profiled cuBLAS and hipBLAS for the aforementioned experiments. cuBLAS launches one kernel to perform the whole QR factorization, which is similar to our batch qr_gpu code. However, it uses only 63 thread-blocks for 1000 matrices of size 27×20, and 125 thread-blocks for 1000 matrices of size 125×56. Using fewer thread-blocks indicates that one thread-block processes multiple matrices, which affects the kernel's overall occupancy on the GPU. hipBLAS launches several kernels to perform the factorization, which causes serious overhead in terms of unnecessary global memory traffic. We also observe a significant drop in the performance gain for the set of 125×56 matrices on the MI100 GPU. We believe this is mainly due to the relatively small shared memory on the MI100 (64KB),



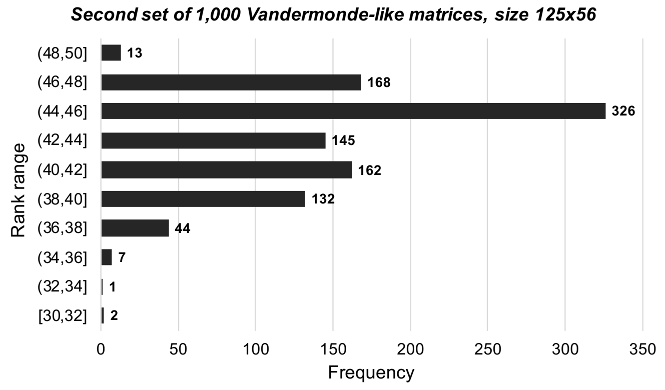


Fig. 3. Histograms of the actual ranks detected by the batch PAQR kernel for the tested two sets of WLS set of matrices.

compared to the A100 (192KB). paqr_gpu requires about 57KB for the larger matrix set, which means that each compute unit in the MI100 GPU can host only one matrix at a time. On the A100, assuming that the CUDA runtime takes the correct scheduling decision, three matrices can be factorized on the same multiprocessor simultaneously.

	Size	Ref. (μs)	qr_gpu (μs)	paqr_gpu (μs)
A100	27×20	294.13	109.33 (2.7×)	100.46 (2.9×)
	125×56	6215.2	2852.4 (2.2×)	2692.2 (2.3×)
MI100	27×20	1508.7	194.31 (7.8×)	174.21 (8.7×)
	125×56	11057.85	8561.13 (1.3×)	8039.02 (1.4×)

TABLE V
PERFORMANCE OF THE BATCH PAQR IN DOUBLE PRECISION ON THE A100 GPU USING CUDA-11.6, AND ON THE MI100 GPU USING ROCM-5.0. RESULTS ARE SHOWN FOR TWO DIFFERENT SIZES OF THE WLS SET OF MATRICES. THE *Ref.* ENTRY REFERS TO CUBLAS OR HIPBLAS.

c) ScaLAPACK: In this section we present results gathered using the ScaLAPACK implementation of PAQR. We compare the time to factorization for QR, QRCP and PAQR using matrices discussed in Section V-A1 regarding the Quantum many-body matrices. A brief discussion of the related performance of each algorithm is presented as well as a comparison for the columns deemed “rejected” according to PAQR.

#Nodes	Method	Matrix size			
		57 600		160 000	
		Time(s)	#Def cols	Time(s)	#Def cols
1	PAQR ϵ	160	45180		
	PAQR 10^{-8}	117	52073		
	QR	336			
	RRQR	4042			
2	PAQR ϵ	109	45217		
	PAQR 10^{-8}	75	52073		
	QR	243			
	RRQR	2087			
4	PAQR ϵ	54	44792	563	135583
	PAQR 10^{-8}	41	52073	454	150673
	QR	100		1779	
	RRQR	1050		*	
8	PAQR ϵ	38	44300	411	134036
	PAQR 10^{-8}	30	52073	220	150673
	QR	63		919	
	RRQR	556		*	
16	PAQR ϵ	31	43996	191	133930
	PAQR 10^{-8}	25	52073	136	150673
	QR	44		498	
	RRQR	304		*	
32	PAQR ϵ	23	43644	138	133005
	PAQR 10^{-8}	20	52073	96	150673
	QR	32		355	
	RRQR	174		2086	
64	PAQR ϵ	-		78	132636
	PAQR 10^{-8}	-		62	150673
	QR	-		162	
	RRQR	-		1103	

TABLE VI
TIME TO FACTORIZATION RESULTS GATHERED ON SUMMIT FOR QR, QRCP, AND PAQR. BLANK CELLS CORRESPOND TO RUNS THAT DO NOT FIT IN MEMORY. $*$ CORRESPONDS TO IRRELEVANT RUNS. $-$ CORRESPONDS TO SKIPPED RUNS. ϵ CORRESPONDS TO DOUBLE-PRECISION.

Table VI presents time to factorization results gathered on the Oak Ridge National Laboratory Summit Supercomputer [46]. We use: 42 *Power9* CPU cores per node, and do not rely on the GPUs, as ScaLAPACK is not compatioble with their use; *ESSL 6.3.0* library as the Level i BLAS mplementation; *Spectrum 10.4.0.3* as the MPI library; *GCC 9.1.0* compiler. The QR and QRCP implementation used are the ScaLAPACK*PDGEQRF* and *PDGEQPF* rourines. PAQR was implemented baed on the *PDGEQRF* routine. For the two problem sizes, we vary the number of nodes used for the factorization and compare the speedups between the different implementations. Whenever a data point is not expressed on Table VI, this is due to either the run being too expensive for us to compute in a reasonable amount of time, or the problem size did not warrant the use of that many nodes.

The results gathered are done with two different matrix sizes. For case (1) we experiment with the dimension $m = n = 57,600$ corresponding to the basis set discussed in Section V-A1. For case (2), the size is $m = n = 160,000$. Given the differences to both case problem sizes, experiments are run using a varying number of nodes. Case (1) is run using from 1 to 32 nodes on Summit (we do not use 64 nodes for this experiment because the time to factorization is already reduced significantly from the factorization on 1 node and as we increase the number of nodes we begin to move from the compute bound spectrum to being communication bound). For case (2), the results are gathered using 4 nodes to 64 nodes on Summit. For this problem size, QRCP does not have a data point for 4, 8 and 16 nodes as the time to factorization exceeded our allocated time to factorize, meaning it is too expensive to compute for that number of nodes.

For PAQR, two thresholds are analyzed to test factorization times under the assumption, with a stricter threshold we desire a lower-rank approximation compared to that of a less strict threshold. In each experiment, we compare against using a tolerance of machine-precision (ϵ) relative to the double precision, or setting the deficiency criteria to 10^{-8}. Section V-A1 discusses that on average, one expects the rank of these matrices to be at most half of their dimension. The column "Def cols" in Table VI shows the number of rejected columns detected by PAQR given the input tolerance given. In every case, PAQR removes a significant number of columns from the factorization. With a more strict threshold, the number of columns dropped is deterministic, regardless of the number of nodes being used. Whereas, when the tolerance is close to that of machine precision, there is a little variation to the number of columns dropped by PAQR. Many of the entries within these input matrices have values that are zero or very close to that, and depending on the setup of the factorization, the algorithm can have varying noise accumulation with these values. Which is why we see some variation for the different ways of breaking up the factorization in accordance to the rejected columns.

As one reviews Table VI, there are two performance comparisons to consider:

1) For each factorization, for a particular number of nodes, we can compare the improvement of PAQR to that of its algorithmic counterparts;
2) One can compare the strong scalability when increasing the number of nodes.

For the problem size of 57,600, PAQR can achieve an overall speedup greater than 3x compared to traditional QR. When we compare the time to factorization of PAQR to that of QRCP, on one node, PAQR is almost 40x faster. This is a significant speedup where for this problem, PAQR removes 52,073 rejected columns. For the problem size of 160,000, we see PAQR can achieve similar results. PAQR on 32 nodes is over 20x faster than QRCP and almost 3.5x faster than QR. For this case, the deficiency criteria is defined as 10^{-8}, PAQR removes 150,673 columns which corresponds to 94% of the number of columns of the input matrix.

Finally, we were able to carry out a PAQR factorization of the third problem from the Quantum many-body matrix set (Beta Carotene) of size $506,944$ on 128 nodes of the Summit supercomputer. The runtime obtained is 1155 seconds. PAQR was able to flag and remove $393,805$ columns. This corresponds to the theoretical expected amount of rejected columns of the matrix. This stunning achievement of PAQR would not be possible to obtain with QRCP in any reasonable amount of time.

VI. Conclusion

A. Summary

This paper presents PAQR, an algorithm for the factorization of rank-deficient matrices arising from linear least-squares problems. It is particularly well suited for large-scale problems. Indeed, PAQR is faster than QR and appears to be as numerically stable as QRCP. This technique can be implemented using different criteria to flag potentially rejected columns in rank-deficient matrices. These criteria are adaptable as they can be conveniently adjusted according to the type of arithmetic precision used and the type of application at hand.

Preliminary tests have revealed the efficiency of the proposed algorithm and its robustness vis-à-vis a variety of cases. Indeed, PAQR was implemented on three types of architectures: sequential, GPU and distributed-memory. It was applied on various matrices of various sizes and arising from various application fields. Further examination of the behaviour of the algorithm will strengthen these results.

B. Future work

The work in this paper has opened up exciting perspectives.

First, the limitations mentioned in Section III-C together with the experimental results described in Section V-B1 revealed that PAQR oftentimes identifies more columns as non-rejected compared to the true rank of a matrix. From the numerical accuracy perspective, it can lead to edge cases where the numerical stability is compromised. From a performance perspective, we believe that there may still be room for improvement, especially at large-scale. This aspect needs to be studied more formally in order to improve our understanding of the pathological cases and to derive improved deficiency criteria.

Second, our preliminary results have shown the impact the parameter α can have on the efficiency of PAQR. Specifically, the number of rejected columns flagged by PAQR can vary widely depending on its value. The default value of α used in this paper was purposefully conservative. This is because we want to avoid the risk of wrongly flagging a column as rejected when it actually does contribute to the rank of the matrix. However, the α parameter can in fact be user-defined to speed-up PAQR by taking advantage of user knowledge. This will allow PAQR to *safely* remove more columns than it otherwise would have. An application-centric study can be done to evaluate the appropriate value of α.

Third, the initial purpose of solving a least-squares problem involving rank-deficient matrices naturally leads us to consider PAQR for low-rank compression purposes. Several applications rely on RRQR and SVD for this purpose but the scalability of such methods is a major bottleneck that PAQR does not suffer from. It becomes possible to construct a low-rank representation that can be used as a preconditioner.

One avenue of interest is the use of PAQR as a first coarse-grain pass of compression on a large matrix, followed by an SVD as a second finer-grain compression scheme on a much smaller matrix.

Fourth, communication avoiding techniques have been successfully applied to QR and RRQR in the CAQR and CARRQR algorithms. PAQR can benefit from such techniques in a very similar way in a CPAQR variant.

Another future direction is to provide a high performance GPU solution for a single PAQR factorization. The algorithmic adaptation and optimizations for one relatively large matrix are different from a batch of small matrices, which requires further investigation.

References

[1] A. Björck, *Numerical Methods for Least Squares Problems*. SIAM Philadelphia, 1996.

[2] C. Lawson and R. Hanson, *Solving Least Squares Problems*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[3] H. Avron, P. Maymounkov, and S. Toledo, "Blendenpik: Supercharging LAPACK's least-squares solver," *SIAM Journal on Scientific Computing*, vol. 32, no. 3, pp. 1217–1236, 2010. [Online]. Available: http://dx.doi.org/10.1137/090767911

[4] G. H. Golub and C. F. V. Loan, *Matrix Computations*, 4th ed. Baltimore, MD, USA: The Johns Hopkins University Press, 2013.

[5] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer, "The ELPA library: scalable parallel eigenvalue solutions for electronic structure theory and computational science," *Journal of Physics: Condensed Matter*, vol. 26, p. 213201, 2014.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997, http://www.netlib.org/scalapack/slug/.

[7] J. Poulson, B. Marker, R. van de Geijn, J. Hammond, and N. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Trans. Math. Software*, vol. 39, 2013.

[8] J. Xiao, M. Gu, and J. Langou, "Fast parallel randomized QR with column pivoting algorithms for reliable low-rank matrix approximations," in *24th IEEE International Conference on High Performance Computing, Data, and Analytics (HIPC), Jaipur, India*, 2017, best Paper Award.

[9] J. Todd, "On condition numbers," *Programmation en Mathématiques Numériques*, vol. 7, no. 165, pp. 141–159, 1966, Éditions Centre Nat. Recherche Sci., Paris, 1968.

[10] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002.

[11] T. F. Chan, "Rank-revealing QR factorizations," *Linear Algebra and its Applications*, vol. 88/89, pp. 67–82, 1987.

[12] R. da Cunha, D. Becker, and J. Patterson, "New parallel (rank-revealing) QR factorization algorithms," in *Euro-Par 2002. Parallel Processing: Eighth International Euro-Par Conference, Paderborn, Germany, August 27–30*, 2002.

[13] C. Bischof and G. Quintana-Orti, "Computing rank-revealing QR factorizations of dense matrices," *ACM Trans. Math. Soft.*, vol. 24, no. 2, pp. 226–253, 1998.

[14] ——, "Algorithm 782: Codes for rank-revealing QR factorizations of dense matrices," *ACM Trans. Math. Soft.*, vol. 24, no. 2, pp. 254–257, 1998.

[15] M. Gu, "Some new rank-revealing factorizations," Department of Mathematics, University of California at Los Angeles, Tech. Rep. CAM Report 98-33, August 1998.

[16] M. Gu and S. Eisenstat, "An efficient algorithm for computing a rank-revealing QR decomposition," Yale University, Computer Science Dept. Report YALEU/DCS/RR-967, June 1993.

[17] S. Chandrasekaran and I. Ipsen, "On rank-revealing QR factorizations," *SIAM Journal on Matrix Analysis and Applications*, vol. 15, 1994.

[18] T.-M. Hwang, W.-W. Lin, and E. K. Yang, "Rank revealing LU factorization," *Linear Algebra and Its Applications*, vol. 175, pp. 115–141, 1992.

[19] C. T. Pan, "On the existence and computation of rank-revealing LU factorizations," submitted to Lin. Alg. Appl., 1996.

[20] G. W. Stewart, "Updating a rank-revealing ULV decomposition," *SIAM J. Mat. Anal. Appl.*, vol. 14, no. 2, pp. 494–499, April 1993.

[21] G. Quintana-Ortí, X. Sun, and C. H. Bischof, "A BLAS-3 version of the QR factorization with column pivoting," *SIAM Journal on Scientific Computing*, vol. 19, no. 5, pp. 1486–1494, 1998.

[22] Y. P. Hong and C.-T. Pan, "Rank-revealing qr factorizations and the singular value decomposition," *Mathematics of Computation*, vol. 58, no. 197, pp. 213–232, 1992. [Online]. Available: http://www.jstor.org/stable/2153029

[23] M. Gu and S. Eisenstat, "Efficient algorithms for computing a strong rank-revealing QR factorization," *SIAMX*, vol. 17, no. 4, pp. pp. 848–869, 1996.

[24] G. H. Golub and W. Kahan, "Calculating the singular values and pseudoinverse of a matrix," *SIAM J. Numer. Anal.*, vol. 2, pp. 205–224, 1965.

[25] J. J. M. Cuppen, "The singular value decomposition in product form," *SIAM J. Sci. Statist. Comput.*, vol. 4, pp. 216–221, 1983.

[26] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential qr and lu factorizations," *SIAM Journal on Scientific Computing*, vol. 34, no. 1, pp. A206–A239, 2012. [Online]. Available: https://doi.org/10.1137/080731992

[27] J. W. Demmel, L. Grigori, M. Gu, and H. Xiang, "Communication avoiding rank revealing qr factorization with column pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 36, no. 1, pp. 55–89, 2015. [Online]. Available: https://doi.org/10.1137/13092157X

[28] P.-G. Martinsson, G. Quintana OrtÍ, N. Heavner, and R. van de Geijn, "Householder qr factorization with randomization for column pivoting (hqrrp)," *SIAM Journal on Scientific Computing*, vol. 39, no. 2, pp. C96–C115, 2017. [Online]. Available: https://doi.org/10.1137/16M1081270

[29] J. A. Duersch and M. Gu, "Randomized qr with column pivoting," *SIAM Journal on Scientific Computing*, vol. 39, no. 4, pp. C263–C291, 2017. [Online]. Available: https://doi.org/10.1137/15M1044680

[30] J. Xiao, M. Gu, and J. Langou, "Fast parallel randomized qr with column pivoting algorithms for reliable low-rank matrix approximations," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 233–242.

[31] J. A. Duersch and M. Gu, "Randomized projection for rank-revealing matrix factorizations and low-rank approximations," *SIAM Review*, vol. 62, no. 3, pp. 661–682, 2020. [Online]. Available: https://doi.org/10.1137/20M1335571

[32] W. Hayes and K. R. Jackson, "A survey of shadowing methods for numerical solutions of ordinary differential equations," *Appl. Numer. Math.*, vol. 53, p. 299–321, 2005.

[33] R. M. Corless and N. Fillion, *A Graduate Introduction to Numerical Methods from the Viewpoint of Backward Error Analysis*. Springer, 2013.

[34] P. C. Hansen, "Regularization tools version 4.0 for Matlab 7.3," *Numerical Algorithms*, vol. 46, no. 2, pp. 189–194, Oct 2007. [Online]. Available: https://doi.org/10.1007/s11075-007-9136-9

[35] C. H. Bischof, "A parallel QR factorization algorithm with controlled local pivoting," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 1, pp. 36–57, 1991. [Online]. Available: https://doi.org/10.1137/0912002

[36] G. W. Stewart, "The QLP approximation to the singular value decomposition," *SIAM Journal on Scientific Computing*, vol. 20, no. 4, pp. 1336–1348, 1999. [Online]. Available: https://doi.org/10.1137/S1064827597319519

[37] G. Golub, V. Klema, and G. W. Stewart, "Rank degeneracy and least squares problems," Tech. Rep., 1976.

[38] M. Gu and S. C. Eisenstat, "Efficient algorithms for computing a strong rank-revealing QR factorization," *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 848–869, 1996. [Online]. Available: https://doi.org/10.1137/0917055

[39] D. A. Huckaby and T. F. Chan, "Stewart's pivoted QLP decomposition for low-rank matrices," *Numerical Linear Algebra with Applications*, vol. 12, no. 2-3, pp. 153–159, 2005. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/nla.404

[40] D. Devendran, D. Graves, H. Johansen, and T. Ligocki, "A fourth-order Cartesian grid embedded boundary method for Poisson's equation," *Communications in Applied Mathematics and Computational Science*, vol. 12, no. 1, pp. 51–79, May 2017. [Online]. Available: https://doi.org/10.2140/camcos.2017.12.51

[41] X. Xing, H. Huang, and E. Chow, “A linear scaling hierarchical block low-rank representation of the electron repulsion integral tensor,” *The Journal of Chemical Physics*, vol. 153, no. 8, p. 084119, 2020. [Online]. Available: https://doi.org/10.1063/5.0010732

[42] R. Ditchfield, W. J. Hehre, and J. A. Pople, “Self-consistent molecular-orbital methods. IX. An extended Gaussian-type basis for molecular-orbital studies of organic molecules,” *J. Chem. Phys.*, vol. 54, pp. 724–728, 1971.

[43] W. J. Hehre, R. Ditchfield, and J. A. Pople, “Self-consistent molecular orbital methods. XII. further extensions of Gaussian-type basis sets for use in molecular orbital studies of organic molecules,” *J. Chem. Phys.*, vol. 56, pp. 2257–2261, 1972.

[44] P. C. Hariharan and J. A. Pople, “The influence of polarization functions on molecular orbital hydrogenation energies,” *Theor. Chim. Acta*, vol. 28, pp. 213–222, 1973.

[45] “Nvidia gtx a100 datasheet,” https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-dgx-a100-datasheet.pdf.

[46] “Nvidia gtx a100 datasheet,” https://www.olcf.ornl.gov/olcf-resources/compute-systems/summit/.