A short note on the Householder QR factorization

Alfredo Buttari

October 25, 2016

1 Uses of the QR factorization

This document focuses on the QR factorization of a dense matrix. This method decomposes the input matrix $A \in \mathbb{R}^{m\times n}$[1], assumed to be of full rank, into the product of a square, orthogonal matrix $Q \in \mathbb{R}^{m\times m}$ and an upper triangular matrix $R \in \mathbb{R}^{n\times n}$.

Theorem 1.1 - Björck [2, Theorem 1.3.1].
Let $A \in \mathbb{R}^{m\times n}$, $m \geq n$. Then there is an orthogonal matrix $Q \in \mathbb{R}^{m\times m}$ such that

$$A = Q \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{1}$$

where R is upper triangular with nonnegative diagonal elements. The decomposition (1) is called the QR decomposition of A, and the matrix R will be called the R-factor of A.

The columns of the Q matrix can be split in two groups

$$Q = [Q_1 Q_2] \tag{2}$$

where $Q_1 \in \mathbb{R}^{m\times n}$ is an orthogonal basis for the range of A, $\mathcal{R}(A)$ and $Q_2 \in \mathbb{R}^{m\times(m-n)}$ is an orthogonal basis for the kernel of A^T, $\mathcal{N}(A^T)$.

The QR decomposition can be used to solve square linear system of equations

$$Ax = b, \quad \text{with } A \in \mathbb{R}^{n\times n}, \tag{3}$$

as the solution x can be computed through the following three steps (where we use MATLAB notation)

$$\begin{cases} [QR] = qr(A) \\ z = Q^T b \\ x = R \backslash z \end{cases} \tag{4}$$

where, first, the QR decomposition is computed (e.g., using one of the methods described in the next section), an intermediate result is computed trough a simple matrix-vector product and, finally, solution x is computed through a triangular system solve. It must be noted that the second and the third steps are commonly much cheaper than the first and that the same QR decomposition can be used to solve matrix A against multiple right-hand sides b. If the right hand sides are available all together, then the second and third steps can each be applied at once to all of them. As we will explain the next two sections, the QR decomposition is commonly unattractive in practice for solving square systems mostly due to its excessive cost when compared to other available techniques, despite its desirable numerical properties.

The QR decomposition is instead much more commonly used for solving linear systems where A is overdetemined, i.e. where there are more equations than unknowns. In such cases, unless the right-hand

[1]Without loss of generality here we will assume that the same algorithms and methods can be generalized to the case of complex arithmetic.

side b is in the range of A, the system admits no solution; it is possible, though, to compute a vector x such that Ax is as close as possible to b, or, equivalently, such that the residual $\|Ax - b\|_2$ is minimized:

$$\min_x \|Ax - b\|_2. \tag{5}$$

Such a problem is called a *least-squares* problem and commonly arises in a large variety of applications such as statistics, photogrammetry, geodetics and signal processing. One typical example is given by linear regression where a linear model, say $f(x, y) = a + bx + cy$ has to be fit to a number of observations subject to errors $(f_i, x_i, y_i),\ i = 1, ..., m$. This leads to the overdetermined system

$$\begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ \vdots & \vdots & \vdots \\ 1 & x_m & y_m \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_m \end{bmatrix}$$

Assuming the QR decomposition of A in Equation (1) has been computed and

$$Q^T b = \begin{bmatrix} Q_1^T \\ Q_2^T \end{bmatrix} b = \begin{bmatrix} c \\ d \end{bmatrix}$$

we have

$$\|Ax - b\|_2^2 = \|Q^T Ax - Q^T b\|_2^2 = \|Rx - c\|_2^2 + \|d\|_2^2.$$

This quantity is minimized if $Rx = c$ where x can be found with a simple triangular system solve. This is equivalent to saying that $Ax = Q_1 Q_1^T b$ and thus solving Equation (5) amounts to finding the vector x such that Ax is the orthogonal projection of b over the range of A, as shown in Figure 1. Also note that $r = Q_2 Q_2^T b$ and thus r is the projection of b on the null space of A^T, $\mathcal{N}(A)$.

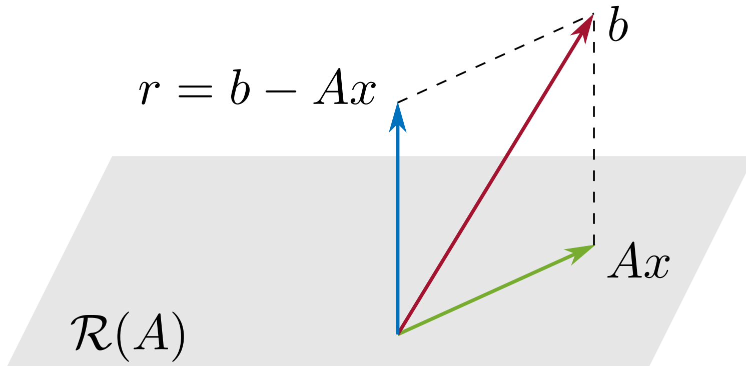


Figure 1: Solution of a least-squares problem.

Another commonly used technique for the solution of the least-squares problem is the *Normal Equations* method. Because the residual r is in $\mathcal{N}(A^T)$

$$A^T(Ax - b) = 0$$

and, thus, the solution x to Equation (5) can be found solving the linear system $A^T Ax = A^T b$. Because $A^T A$ is Symmetric Positive Definite (assuming A has full rank), this can be achieved through the Cholesky factorization. Nonetheless, the method based on the QR factorization is often preferred because the conditioning of $A^T A$ is equal to the square of the conditioning of A, which may lead to excessive error propagation.

The QR factorization is also commonly used to solve underdetermined systems, i.e., with more unknowns than equations, which admit infinite solutions. In such cases the desired solution is the one with minimum 2-norm:

$$min\|x\|_2, \quad Ax = b. \tag{6}$$

The solution of this problem can be achieved by computing the QR factorization of A^T

$$[Q_1 Q_2] \left[\begin{array}{c} R \\ 0 \end{array} \right] = A^T$$

where $Q_1 \in \mathbb{R}^{n\times m}$ and $Q_2 \in \mathbb{R}^{n\times(n-m)}$. Then

$$Ax = R^T Q^T x = [R^T 0] \left[\begin{array}{c} z_1 \\ z_2 \end{array} \right] = b$$

and the minimum 2-norm solution follows by setting $z_2 = 0$. Note that Q_2 is an orthogonal basis for $\mathcal{N}(A)$ and, thus, the minimum 2-norm solution is computed by removing for any admissible solution $\tilde{x}$ all of its components in $\mathcal{N}(A)$.

We do not discuss here the use of iterative methods such as LSQR [8] or Craig [3] for the solution of problems (5) and (6) but we refer the reader to the book by Björck [2] which contains a thorough description of the classical methods used for solving least-squares and minimum 2-norm problems.

2 Householder QR decomposition

The QR decomposition of a matrix can be computed in different ways; the use of Givens Rotations [4], Householder reflections [7] or the Gram-Schmidt orthogonalization [9] are among the most commonly used and best known ones. We will not cover here the use of Givens Rotations, Gram-Schmidt orthogonalization and their variants and refer, instead, the reader to classic linear algebra textbooks such as Golub et al. [6] or Björck [2] for an exhaustive discussion of such methods. We focus, instead, on the QR factorization based on Householder reflections which has become the most commonly used method especially because of the availability of algorithms capable of achieving very high performance on processors equipped with memory hierarchies.

For a given a vector u, a *Householder Reflection* is defined as

$$H = I - 2\frac{uu^T}{u^T u} \tag{7}$$

and u is commonly referred to as *Householder vector*. It is easy to verify that H is symmetric and orthogonal. Because $P_u = \frac{uu^T}{u^T u}$ is a projector over the space of u, Hx can be regarded as the reflection of a vector x on the hyperplane that has normal vector u. This is depicted in Figure 2 (*left*).

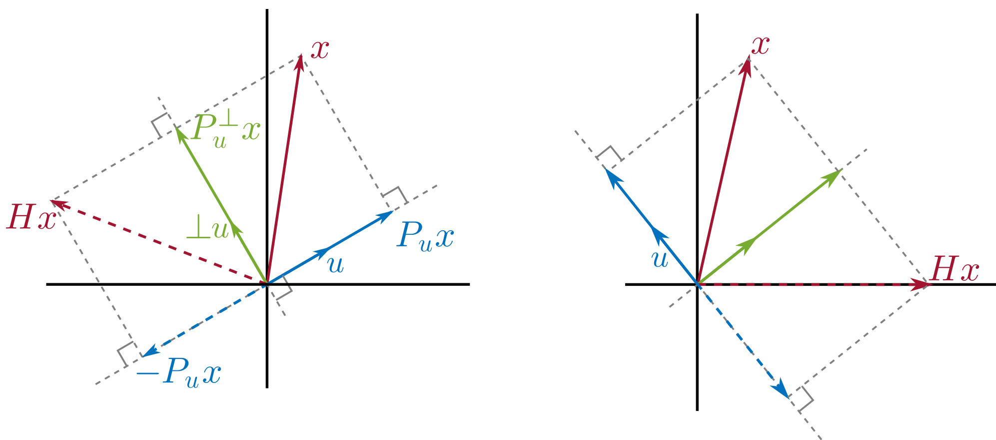


Figure 2: Householder reflection

The Householder reflection can be defined in such a way that Hx has all zero coefficients except the first, which means to say that $Hx = \pm\|x\|_2 e_1$ where e_1 is the first unit vector. This can be achieved if Hx

is the reflection of x on the hyperplane that bisects the angle between x and $\pm\|x\|_2 e_1$. This hyperplane is orthogonal to the difference of these two vectors $x \mp \|x\|_2 e_1$ which can thus be used to construct the vector

$$u = x \mp \|x\|_2 e_1. \tag{8}$$

Note that, if, for example, x is close to a multiple of e_1, then $\|x\|_2 \approx x(1)$ which may lead to a dangerous cancellation in Equation (8); to avoid this problem u is commonly chosen as

$$u = x + sign(x(1))\|x\|_2 e_1. \tag{9}$$

In practice, it is very convenient to scale u in such a way that its first coefficient is equal to 1 (more on this will be said later). Assuming $v = u/u(1)$, through some simple manipulations, the Householder transformation H is defined as

$$H = I - \tau v v^T, \quad \text{where} \quad \tau = \frac{sign(x(1))(x(1) - \|x\|_2)}{\|x\|_2}. \tag{10}$$

Note that the matrix H is never explicitly built neither to store, nor to apply the Householder transformation; the storage is done implicitly by means of v and τ and the transformation can be applied to an entire matrix $A \in \mathbb{R}^{m\times n}$ in $4mn$ flops like this

$$HA = (I - \tau v v^T)A = A - \tau v(v^T A) \tag{11}$$

The Householder reflection can be computed and applied as described above using the `_larfg` and `_larf` routines in the LAPACK[1] library.

A sequence of Householder transformations can be used to zero-out all the coefficients below the diagonal of a dense matrix to compute its QR factorization:

$$H_n H_{n-1} ... H_2 H_1 A = R, \quad \text{where} \quad H_n H_{n-1} ... H_2 H_1 = Q^T.$$

Each transformation H_k annihilates all the coefficients below the diagonal of column k and modifies all the coefficients in the trailing submatrix $A(k : m, k+1 : n)$. The total cost of this algorithm is $2n^2(m - n/3)$. The Q matrix is implicitly represented by means of the v_k vectors and the τ_k coefficients. One extra array has to be allocated to store the τ_k coefficients, whereas the v_k vectors can be stored inside matrix A in the same memory as the zeroed-out coefficients; this is possible because the v_k have been scaled as described above and thus the 1 coefficients along the diagonal must not be explicitly stored. The LAPACK `_geqr2` routine implements this algorithm.

Theorem 2.1 - Björck [2, Remark 2.4.2].
Let $\bar{R}$ denote the computed R. It can be shown that there exists an exactly orthogonal matrix $\bar{Q}$ (*not* the computed Q) such that

$$A + E = \bar{Q}\bar{R}, \quad \|E\|_F \le c_1 u \|A\|_F,$$

where the error constant $c_1 = c_1(m, n)$ is a polynomial in m and n, $\|\cdot\|_F$ denotes the Frobenius norm and u the unit roundoff.

In other words, the Householder QR factorization is normwise backward stable.

The use of a QR factorization by mean of a sequence of orthogonal transformations to solve least-squares problems was introduced by Golub [5]; this method is also proven to be backward stable:

Theorem 2.2 - Björck [2, Remark 2.4.8].
Golub's method for solving the standard least squares problem is normwise backward stable. The computed solution $\hat{x}$ can be shown to be the exact solution of a slightly perturbed least squares problem

$$\min_x \|(A + \delta A)x - (b + \delta b)\|_2,$$

where the perturbations satisfy the bounds

$$\|\delta A\|_2 \leq cun^{1/2}\|A\|_2, \quad \|\delta b\|_2 \leq cu\|b\|_2$$

and $c = (6m - 3n + 41)n$.

Despite these very favorable numerical properties, the QR factorization is rarely preferred to the Gaussian Elimination (or LU factorization) with Partial Pivoting (GEPP) for the solution of square systems because its cost is twice the cost of GEPP and because partial pivoting is considered stable in most practical cases.

The above discussed algorithm for computing the QR factorization is basically never used in practice because it can only achieve a modest fraction of the peak performance of a modern processor. This is due to the fact that most of the computations are done in the application of the Householder transformation to the trailing submatrix as in Equation (11) which is based on Level-2 (i.e., matrix-vector) BLAS operations and thus limited by the speed of the memory rather than the speed of the processor. In order to overcome this limitation and considerably improve the performance of the Householder QR factorization on modern computers equipped with memory hierarchies, Schreiber et al. [10] proposed a way of accumulating multiple Householder transformations and applying them at once by means of Level-3 BLAS operations.

Theorem 2.3 - Compact WY representation (Adapted form Schreiber et al. [10]).
Let $Q = H_1 ... H_{k-1}H_k$, with $H_i \in \mathbb{R}^{m\times m}$ an Householder transformation defined as in Equation (10) and $k \leq m$. Then, there exist an upper triangular matrix $T \in \mathbb{R}^{k\times k}$ and a matrix $V \in \mathbb{R}^{m\times k}$ such that

$$Q = I + VTV^T.$$

Proof. The proof is by induction on k. The case $k = 0$ is straightforward. Now assume that a matrix $Q_k = H_1 ... H_{k-1}H_k$ has a compact WY representation $Q_k = I + V_k T_k V_k^T$ and consider

$$Q_{k+1} = Q_k H_{k+1} = Q_k(I - \tau_{k+1}v_{k+1}v_{k+1}^T).$$

Then a compact WY representation for Q_{k+1} is given by $Q_{k+1} = I + V_{k+1}T_{k+1}V_{k+1}^T$ where

$$T_{k+1} = \begin{bmatrix} T_k & -\tau_{k+1}T_k V_k^T v_{k+1} \\ 0 & -\tau_{k+1} \end{bmatrix}, \quad V_{k+1} = \begin{bmatrix} V_k & v_{k+1} \end{bmatrix}$$

which concludes the proof. □

Using this technique, matrix A can be logically split into $\lceil n/b \rceil$ *panels* (block-columns) of size b and the QR factorization achieved in the same number of steps where, at step k, panel k is factorized using the `_geqr2` routine, the corresponding T matrix is built using the `_larft` routine and then the set of b transformations is applied at once to the trailing submatrix through the `_larfb` routine. This last operation/routine is responsible for most of the flops in the QR factorization: because it is based on matrix-matrix operations it can achieve a considerable fraction of the processor's peak performance. This method is implemented in the LAPACK `_geqrf` routine which uses an implicitly defined blocking size b and discards the T matrices computed for each panel. More recently, the `_geqrt` routine has been introduced in LAPACK which employs the same algorithm but takes the block size b as an additional argument and returns the computed T matrices.

Here is the pseudocode for the `_geqrt` routine:

```
subroutine _geqrt(A, T, b)
   input      : A(m,n), b
   output     : A(m,n), T(b,n)
   temporary : tau(n)

   do j=1, n, b
      call _geqr2(A(j:m,j:j+b-1), tau(j:j+b-1) )
      call _larft(tau(j:j+b-1), T(1:b,j:j+b-1))
      call _larfb(A(j:m,j:j+b-1), T(1:b,j:j+b-1),
               & A(j:m,j+b:n))
   end do
```

Note that this routine computes the QR factorization **in-place**: the input matrix A is overwritten by the R factor (in the upper-triangular part) and the matrix V (in the lower-triangular part) that contains the Householder vectors. This is possible because V is unitary-diagonal (i.e., its diagonal coefficients are equal to 1 and thus need not be stored).

Once the QR factorization is computed by means of the `_geqrt` routine, it can be applied to another matrix B using the `_gemqrt` routine whose pseudocode is given below:

```
subroutine _gemqrt(A, T, B, b)
   input     : A(m,n), b, T(b,n), B(m,k)
   output    : B(m,k)

   do j=1, n, b
      call _larfb(A(j:m,j:j+b-1), T(1:b,j:j+b-1),
                & B(j:m,1:k))
   end do
```

Note that a QR factorization can be computed with `_geqrt` and `_gemqrt` as such

```
subroutine myqr(A, T, ib)
   input     : A(m,n), ib
   output    : A(m,n), T(ib,n)

   do k=1, n, b
      call _geqrt (A(k:m,k:k+b-1), T(1:ib,k:k+b-1), ib)
      call _gemqrt(A(k:m,k:k+b-1), T(1:ib,k:k+b-1),
                 & A(k:m,k+b:n), ib)
   end do
```

Note that, in the code above, two different blocking sizes b and ib are used; the only requirement is that $ib \leq b$.

3 Parallelization of the Householder QR decomposition

3.1 Fork-join parallelization by block-columns

This section discusses a basic parallelization of the Householder QR factorization. For this purpose we will refer to the last pseudo-code given in the previous section; in this pseudo-code the QR factorization is achieved through a sequence of `_geqrt` and `_gemqrt` operations which we will also refer to as, respectively

- **panel** factorization: the QR factorization of a block-column (panel) is computed. This block-columns is strongly overdetermined, i.e., it has many more rows than columns.
- trailing submatrix **update**: the Householder vectors computed by the panel factorization are applied to the submatrix on the right of the panel.

It must be noted that, when a matrix is strongly overdetermined (i.e., it has many more rows than columns), the cost of its QR factorization is dominated by the `_larfg` operation which computes the Householder vectors as defined in Equation (9). This can hardly be parallelized because it is a Level-1 BLAS operation which has a very unfavorable ratio between communication and computation: the norm operation in Equation (9) requires access to a whole column matrix and thus does very few floating-point operations per memory access. Therefore, for the moment, we will assume that the panel factorization cannot be parallelized; we will see how to overcome this limitation later on. On the contrary, the trailing submatrix update is based of Level-3 BLAS operations (matrix-matrix products) and its relative weight is very important, especially in the case where the A matrix to be factorized is square or underdetermined (i.e., more columns than rows). As a result, the update operation, can easily and efficiently be parallelized. One naive way of achieving this parallelization is to split the trailing submatrix in block-columns and update all the block-columns in parallel as in the pseudo-code below; for the sake of simplicity and readability, in this pseudo-code we will make the following assumptions (most will hold for all the following pseudo-codes):

- the matrix A is split in $q = n/b$ block-columns of size b;
- we will ignore the T matrices as well as the inner blocking ib of the `_geqrt` and `_gemqrt` routines;
- we will denote $A[1, i]$ the i-th block-column, i.e. $A(1 : m, (i-1) * b + 1 : i * b)$;
- for clarity, the V and R matrices are explicitly used but it must be noted that this algorithm works in-place, i.e., V and R are stored in the memory that contains A before the factorization.

```
subroutine V,R = block_col_qr(A)
   input : A(m,n)
   output: V(m,n), R(n,n)

   do k=1, q
      V[1,k], R[1,k] = _geqrt(A[1,k])
      !$omp parallel do
      do j=k+1,q
         A[1,j] = _gemqrt(V[1,k], A[1,j])
      end do
   end do
```

Note that in the pseudo-code above the trailing-submatrix was split in block-columns of size b, i.e., the same as the panel size but this doesn't have to be necessarily the case and any other block-size can be chosen. This parallelization approach is commonly referred to as *fork-join* because sequential operations (`_geqrt`) are alternated with parallel ones (`_gemqrt`). This is clearly sub-optimal, by the Amdahl's law, because when the panel factorization is being computed by one process, all the other processes are idle. However, in the case where the matrix A is square or underdetermined, this approach can usually achieve satisfactory results because the relative weight of the panel factorization is very small; moreover, the scalability of this approach can be improved noting that it is not necessary to wait for the completion of all the updates in step `k` to compute the factorization of panel `k+1` but this can be started as soon as the update of block-column `k+1` with respect to panel `k` is finished. This technique is well known under the name of *lookahead* and essentially allows for pipelining two consecutive stages of the blocked QR factorization; the same idea can be pushed further in order to pipeline l stages, in which case we talk of `depth-l` lookahead. The pseudo-code below shows an example of a parallel QR factorization with infinite lookahead obtained with the OpenMP task construct.

```
subroutine V,R = block_col_qr(A)
   input : A(m,n)
   output: V(m,n), R(n,n)

   !$omp parallel private(j,k)
   !$omp master
   do k=1, q
      !$omp task depend(in:A[1,k]) depend(out:V[1,k],R[1,k])
      V[1,k], R[1,k] = _geqrt(A[1,k])
      do j=k+1,q
         !$omp task depend(in:V[1,k]) depend(inout:A[1,j])
         A[1,j] = _gemqrt(V[1,k], A[1,j])
      end do
   end do
   !$omp end master
   !$omp end parallel
```

3.2 Reduction-trees parallelization by block-rows

When the matrix A to be factorized is of small size or strongly overdetermined, the relative weight of the panel factorization becomes more important and the scalability of the fork-join approach is seriously

limited even when lookahead techniques are used. The QR algorithm must be rewritten in order to achieve parallelism also in the panel factorization operation.

In this section we will first introduce, through an example, the concept of reduction trees and see how this can be used to parallelize the panel factorization.

3.2.1 Reduction trees, an example

A reduction tree is a computational pattern used to compute a final result by combining multiple initial data. Computations are arranged in a graph which has the shape of a tree where the leaves are associated with the initial data, the root with the final result and the other nodes with intermediate, partial results; the graph expresses the dependencies among computations and the order in which they are performed. Let's take as an example the computation of the 2-norm of a vector x of size n

$$d = \|x\|_2 = \sqrt{x(1)^2 + ... + x(n)^2}.$$

Assume that the vector is split into four parts of equal size $n/4$:

$$x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix}$$

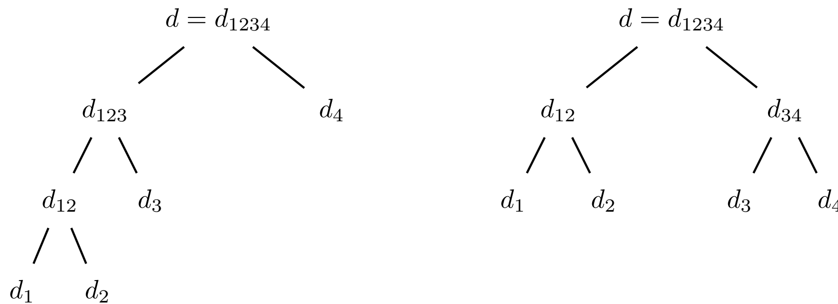
Assuming we have four processes, each process can compute independently and in parallel the norm of a chunk $d_i = \|x_i\|_2$. Once this is done, all is left to do is to **reduce** (i.e., combine) the partial results in order to achieve the final norm d. In order to do so, we introduce a merge operation $\oplus$

$$d_{ij} = d_i \oplus d_j = \sqrt{d_i^2 + d_j^2}$$

which is associative and commutative. The final norm d can be computed, for example, in the following two ways

$$\begin{aligned} d = \quad & d_{1234} = \quad d_{123} \oplus d_4 = (d_{12} \oplus d_3) \oplus d_4 = ((d_1 \oplus d_2) \oplus d_3) \oplus d_4 \\ d = \quad & d_{1234} = \quad d_{12} \oplus d_{34} = (d_1 \oplus d_2) \oplus (d_3 \oplus d_4) \end{aligned}$$

which can be represented using the trees below:



The one on the left is a *flat* tree, whereas the one on the right is a *binary* tree. The tree on the right has one obvious advantage over the one on the left: it delivers more parallelism. In fact, the $d_{12} = d_1 \oplus d_2$ and $d_{34} = d_3 \oplus d_4$ operations are independent and can thus be computed in parallel. As a result the completion of the binary tree is completed in $O(log_2 p)$ steps, where p is the number of initial chunks, whereas the flat tree completes in $O(p)$. In the next section we will see, however, that the flat tree has some other favorable properties.

Note that, because of the commutativity of the $\oplus$ operator, other flat or binary trees are possible; moreover, a n-ary operator can be defined which combines n partial norms which will lead to n-ary trees. It must also be noted that hybrid trees are possible where, for example, one branch is a flat subtree and another branch is a binary subtree.

3.2.2 Parallel QR by block-rows

Let's assume a matrix A of size $m \times n$ strongly overdetermined, i.e. with $m \gg n$. In this case the QR factorization can be achieved through a reduction operation the same as for the norm computation described in the previous section. For this, we have to define a merge operation that allows for combining partial results. Such operation, which we will call `_ttqrt` (where `tt` stands for triangle-triangle), computes the QR of two triangular matrices, one on top of the other:

$$R_{12},\ V_{12}, = \texttt{_ttqrt}(R_1,\ R_2), \quad \textit{where} \quad Q_{12}R_{12} = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$$

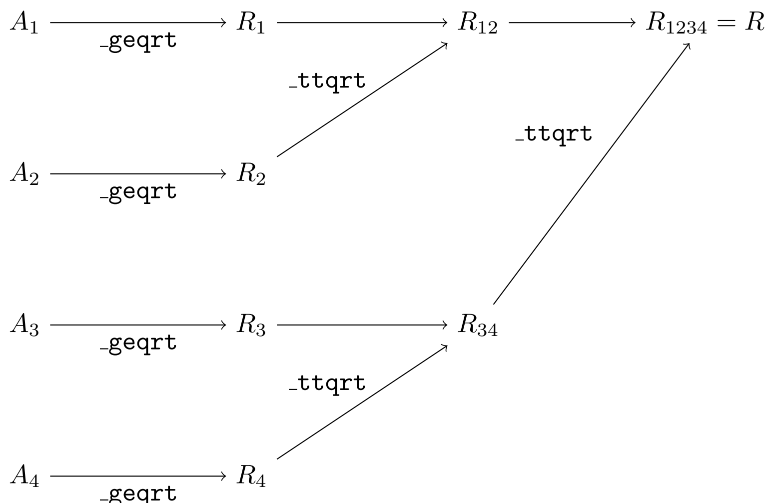
where V_{12} is a $b \times b$ square block containing the Householder vectors that form Q_{12}.

Although mathematically the `_ttqrt` is a standard QR factorization, this operation can be implemented in such a way that the zeroes below the diagonal of R_1 and R_2 are ignored in order to reduce its computational cost. Also note that the `_ttqrt` routine computes the QR factorization in place, .i.e., R_{12} overwrites R_1 and V_{12} (the matrix containing the Householder vectors that form Q_{12}) overwrites R_2.

The QR factorization of A can be computed as follows. The matrix A is split into p block-rows A_p of size m/p (here we will assume that $m/p \geq n$). In a first step, the QR factorization $Q_pR_p = A_p$ of each block is computed using the `_geqrt` routine; because these operations are independent, they can all be computed in parallel. Note that, mathematically this is equivalent to multiplying A by a Q matrix which is block-diagonal with the Q_i^T matrices on the diagonal:

$$\begin{pmatrix} Q_1^T & & & \\ & Q_2^T & & \\ & & \ddots & \\ & & & Q_p^T \end{pmatrix} \cdot \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \\ \vdots \\ R_p \end{pmatrix}.$$

Once this step is over, a reduction process is used to combine all the R_i matrices by means of the `_ttqrt` routine; because `_ttqrt` is commutative and associative many reduction trees can be used. The figure below illustrates this process for the case where $p = 4$ and a binary tree is chosen.



Below a pseudo-code for the parallel QR factorization by block-rows with a binary tree; for the sake of simplicity and readability, in this pseudo-code we will make the following assumptions:

- the matrix A is split in $p = m/n$ block-rows, i.e., into square blocks of size $n \times n$;
- we will ignore the T matrices as well as the inner blocking ib of the `_geqrt` and `_ttqrt` routines;
- we will denote $A[i, 1]$ the i-th block-row, i.e. $A((i-1)*n+1 : i*n, 1 : n)$.

```
subroutine V,R = block_row_qr(A)
   input : A(m,n)
   output: V(m,n), R(n,n)

   !$omp parallel do
   do i=1, p
      V[i,1], R[i,1] = _geqrt (A[i,1])
   end do

   s = 1
   while(s<p) do
      !$omp parallel do
      do i=1, p, 2*s
         V[i+s,1], R[i,1] = _ttqrt(R[i,1], R[i+s,1])
      end do
      s = s*2
   end do
```

Alternatively, the OpenMP task construct can be used to achieve a more asynchronous and efficient execution model:

```
subroutine V,R = block_row_qr(A)
   input : A(m,n)
   output: V(m,n), R(n,n)

   !$omp parallel  private(i,s)
   !$omp master
   do i=1, p
      !$omp task depend(in:A[i,1]) depend(out:V[i,1],R[i,1])
      V[i,1], R[i,1] = _geqrt (A[i,1])
   end do

   s = 1
   while(s<p) do
      do i=1, p, 2*s
         !$omp task depend(in:R[i,1], R[i+s,1]) depend(out:R[i,1], V[i+s,1])
         V[i+s,1], R[i,1] = _ttqrt(R[i,1], R[i+s,1])
      end do
      s = s*2
   end do
   !$omp end master
   !$omp end parallel
```

Note also that hybrid variants of the tiled algorithms can be used (and are actually very common in practice) where, for example, a block-column is logically decomposed into subdomains, a flat subtree is used inside each subdomain and the subdomains are reduced using a binary tree.

3.3 Parallel QR by blocks (or tiles)

Parallel algorithms by blocks or tiles combine the benefits of the parallelization by block-columns and block-rows by decomposing a matrix A of size $m \times n$ into square blocks (or tiles) of size $b \times b$:

$$A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1q} \\ A_{21} & A_{22} & & A_{2q} \\ \vdots & & \ddots & \vdots \\ A_{p1} & A_{p2} & \cdots & A_{pq} \end{pmatrix}.$$

To illustrate these algorithms we have to introduce a new merge operation called `_tsqrt` (where `ts` stands for triangle-square) which computes the QR factorization of a matrix formed by a triangular block R_1 and a square block A_2 one on top of each other:

$$R_{12},\ V_{12} = \texttt{_tsqrt}(R_1,\ A_2), \quad where \quad Q_{12}R_{12} = \begin{pmatrix} R_1 \\ A_2 \end{pmatrix}$$

where V_{12} is a $b \times b$ square block containing the Householder vectors that form Q_{12}. As for the `_ttqrt` routine, `_tsqrt` ignores the zeroes below the diagonal of the R_1 block and computes the factorization in place, i.e., the R_{12} block overwrites the R_1 block and the V_{12} block overwrites the A_2 block. Next we need methods to apply the transformations computed by the `_tsqrt` and `_ttqrt` routines; these are, respectively, the `_tsmqrt` and `_ttmqrt` routines. Imagine we have used the `_tsqrt` routine to compute the QR factorization

$$Q_{12}R_{12} = \begin{pmatrix} R_1 \\ A_2 \end{pmatrix}.$$

The result of this operation are the R_{12} and V_{12} blocks as explained above. The Q_{12}^T transformation can be applied to a couple of blocks B_1 B_2 as follows:

$$\tilde{B}_1,\ \tilde{B}_2 = \texttt{_tsmqrt}(B_1,\ B_2,\ V_{12}), \quad where \quad \begin{pmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{pmatrix} = Q_{12}^T \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

This routine also works in place because the B_1 and B_2 blocks are overwritten by $\tilde{B}_1$ and $\tilde{B}_2$. The `_ttmqrt` works exactly the same for applying a Q_{12}^T transformation computed by the `_ttqrt`. Table 1 contains a list of all the kernels that have been introduced so far along with the related details.

Tiled algorithms advance by block-columns; at each stage the current block-column is reduced using a reduction tree (as explained in Section 3.2.2) and the computed reflections are applied to the concerned blocks on the right of the current block-column. Let's take, as an example, a tiled algorithm based on a flat tree on a matrix of size 4×3 blocks; the first steps of the algorithm are reported on the left column of Table 2. The algorithm proceeds by steps $k = 1, ..., q$ where, at each step

1. The diagonal block A_{kk} is factorized using the `_geqrt` operation;

2. The transformation computed at the previous step is applied to all the blocks $A_{kj},\ j = k+1, ..., q$ along the k-th row using the `_gemqrt` operation. Note that all these blocks can be updated independently and, thus, in parallel.

3. The first sub-diagonal block $A_{k+1,k}$ is put to zero using the `_tsqrt` routine.

4. The transformation computed at the previous step is applied to all the blocks $A_{kj}, A_{k+1,j}\ j = k+1, ..., q$ along the k-th and $(k+1)$-th rows using the `_tsmqrt` operation. Note that all these couples of blocks can be updated independently and, thus, in parallel. Note, also, that these updates only involve the blocks along the rows k and $k+1$.

5-8 Steps 3 and 4 are repeated to annihilate all the sub-diagonal blocks $A_{ik},\ i = k+2, ..., p$.

Here is a **sequential** code (we will see hot to parallelize it later on) that achieves the tiled QR factorization of a matrix using a flat reduction tree:

```
subroutine V,R = tiled_qr_flattree(A)
   input : A(m,n)
   output: V(m,n), R(n,n)

   do k=1, q
      V[k,k], R[k,k] = _gemqrt(A[k,k])

      do j=k+1, q
         A[k,j] = _gemqrt(A[k,j], V[k,k])
      end do

      do i=k+1, p
         V[i,k], R[k,k] = _tsqrt(R[k,k], A[i,k])
         do j=k+1, q
            A[k,j], A[i,j] = _tsmqrt(A[k,j], A[i,j], V[i,k])
         end do
      end do
   end do
```

The algorithm on the right-column of Table 2 works pretty much the same except that the k-th column is reduced using a binary tree:

1. at the first step all the blocks $A_{ik},\ i = k, ..., p$ are factorized using the `_geqrt` routine. Note that each of these blocks is factorized independently of the others and thus all these operations can be done in parallel.
2. the trailing submatrix is updated using the `_gemqrt` routine. Note that each block $A_{ij},\ i = k, ..., p,\ j = k+1, ..., q$ is updated independently of the others and, thus, all these operations can be done in parallel.

3-8 All the triangular blocks in the k-th column are reduced using a binary tree through the `_ttqrt`. The transformation computed by each `_ttqrt`, is applied to the blocks along the two corresponding block-rows using the `_ttmqrt`. As usual, all the `_ttmqrt` operations related to a `_ttqrt` one are independent and can be done in parallel. Note, also, that the `_ttqrt` at step 3 and the one at step 5 are independent (they work on different block) and can be done in parallel. similarly, the `_ttmqrts` in step 4 and those in step 6 are independent and parallelizable. The `_ttqrt` (respectively, `_ttmqrt`) in step 7 (resp. 8) depends on those in steps 3 and 5 (resp. 4 and 6).

Let's compare the two approaches:

- **Flat tree**
  - **Advantage**: good pipelining between stages. It must be noted that, although the reduction of a block-column is sequential (all the `_tsqrt` operations on column k must be done one after the other), the reduction of block-column $k+1$ can be started early and before the reduction of block-column k is finished. This means, for example, that the `_geqrt` on block A_{22} can be executed right after step 4 in Table 2 and before steps 5-8 are completed.
  - **Disadvantage**: it produces less parallelism than the tiled algorithm based on the binary tree especially on strongly overdetermined matrices (i.e., with $p \gg q$).
- **Binary tree**
  - **Advantage**: it produces more parallelism than the method based on the flat tree. This is because some `_ttqrt` (and the associated `_ttmqrt`) operations can be executed in parallel as explained above. This is especially important in strongly over-determined matrices.
  - **Disadvantage**: many more and smaller operations. Compare the left and right columns of Table 2. On both sides the same result is reached at the end of 8 steps, i.e., the reduction of the first block-column and the corresponding update of the trailing submatrix. Nonetheless, the

method based on the flat panel achieved this in 12 operations (1 _geqrt, 2 _gemqrt, 3 _tsqrt and 6 _tsmqrt); the method based on the binary tree, instead needs 21 operations (4 _geqrt, 8 _gemqrt, 3 _ttqrt and 6 _ttmqrt). This implies a higher overhead, a more complex code. Because the overall number of floating-point operations is the same as for the flat tree case, this implies that operations are of smaller granularity and, thus, less efficient.

– **Disadvantage**: because of the more complex pattern of the binary reduction tree, successive stages of the factorization are less efficiently overlapped and stalls may appear in the piepeline.

In conclusion, the method should be chosen depending on the input data (size and shape of the matrix) and the number of available cores:

- the method based on block-column partitioning (see Section 3.1 may be best suited for factorizing large size square or underdetermined matrices, or when the number of cores is very small;
- the tiled method based on a flat tree delivers much more parallelism that the one based on a block-column partitioning and uses operations of relatively good granularity. It is thus very efficient on square or moderately overdetermined matrices even of relatively small size;
- the tiled method based on a binary tree is the variant that produces the most parallelism but should be used with care because operations have much lower efficiency; it is thus suitable only for extremely over-determined matrices or when the number of cores is very high.

As explained in Section 3.2.2, it is also possible (and is very common in practice) to use hybrid trees that combine the advantages of both methods.

As for the parallelization, let's take the tiled algorithm based on a flat tree. Obviously, a straightforward parallelization consists in executing in parallel the update operations _gemqrt and _tsmqrt related to a, respectively, _geqrt and _tsqrt ones. This can be easily achieved by using the `!$omp parallel do` construct on the j loops of the pseudo-code given above. This fork-join approach, however, results in a sub-optimal use of the available concurrency because multiple stages $k = 1, ..., q$ of the factorization cannot be pipelined as explained above (which is one of the strengths of the tiled method based on the flat panel). A more efficient parallelization can be achieved using the OpenMP task construct as in the pseudo-code below:

```
subroutine V,R = tiled_qr_flattree(A)
   input : A(m,n)
   output: V(m,n), R(n,n)

   !$omp parallel private(i,j,k)
   !$omp master
   do k=1, q
      !$omp task depend(in:A[k,k]) depend(out:V[k,k],R[k,k])
      V[k,k], R[k,k] = _gemqrt(A[k,k])

      do j=k+1, q
         !$omp task depend(in:V[k,k]) depend(inout:A[k,j])
         A[k,j] = _gemqrt(A[k,j], V[k,k])
      end do

      do i=k+1, p
         !$omp task depend(in:R[k,k], A[i,k]) depend(out:V[i,k], R[k,k])
         V[i,k], R[k,k] = _tsqrt(R[k,k], A[i,k])
         do j=k+1, q
            !$omp task depend(in:V[i,k]) depend(inout:A[k,j],A[i,j])
             A[k,j], A[i,j] = _tsmqrt(A[k,j], A[i,j], V[i,k])
         end do
      end do
   end do
   !$omp end master
   !$omp end parallel
```

References

[1] E. Anderson et al. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[2] Å. Björck. *Numerical methods for Least Squares Problems*. Philadelphia: SIAM, 1996.

[3] E. J. Craig. "The N-step iteration procedures". In: *Journal of Mathematics and Physics* 34 (1955), pp. 64–73.

[4] W. Givens. "Computation of Plane Unitary Rotations Transforming a General Matrix to Triangular Form". English. In: *Journal of the Society for Industrial and Applied Mathematics* 6.1 (1958), pp. 26–50. ISSN: 03684245. URL: http://www.jstor.org/stable/2098861.

[5] G. Golub. "Numerical methods for solving linear least squares problems". English. In: *Numerische Mathematik* 7.3 (1965), pp. 206–216. ISSN: 0029-599X. DOI: 10.1007/BF01436075. URL: http://dx.doi.org/10.1007/BF01436075.

[6] G. H. Golub and C. F. Van Loan. *Matrix Computations. 4th ed.* Baltimore, MD.: Johns Hopkins Press, 2012.

[7] A. S. Householder. "Unitary Triangularization of a Nonsymmetric Matrix". In: *J. ACM* 5.4 (Oct. 1958), pp. 339–342. ISSN: 0004-5411. DOI: 10.1145/320941.320947. URL: http://doi.acm.org/10.1145/320941.320947.

[8] C. C. Paige and M. A. Saunders. "LSQR: An Algorithm for Sparse Linear Equations and Sparse Least Squares". In: *ACM Trans. Math. Softw.* 8.1 (Mar. 1982), pp. 43–71. ISSN: 0098-3500. DOI: 10.1145/355984.355989. URL: http://doi.acm.org/10.1145/355984.355989.

[9] E. Schmidt. "Über die Auflösung linearer Gleichungen mit Unendlich vielen unbekannten". German. In: *Rendiconti del Circolo Matematico di Palermo (1884-1940)* 25.1 (1908), pp. 53–77. DOI: 10.1007/BF03029116. URL: http://dx.doi.org/10.1007/BF03029116.

[10] R. Schreiber and C. Van Loan. "A storage-efficient WY representation for products of Householder transformations". In: *SIAM J. Sci. Stat. Comput.* 10 (1989), pp. 52–57.

Routine	Operation	Description	Cost
_geqrt	$V,\ R = _\texttt{geqrt}(A)$	where $A \in R^{m\times n}$, V contains the Householder vectors that form Q and $QR = A$	$2n^2(m - \frac{n}{3})$
_gemqrt	$\tilde{B} = _\texttt{gemqrt}(B,\ V)$	where $V \in R^{m\times n}$, $B \in R^{m\times k}$, V contains the Householder vectors that form Q and results from _geqrt and $\tilde{B} = Q^T B$	$3n^2k + 4(m-n)nk$
_tsqrt	$V_{12},\ R_{12} = _\texttt{tsqrt}(R_1, A_2)$	where $R_1, A_2, V_{12}, R_{12} \in R^{b\times b}$, R_1 is upper triangular, V_{12} contains the Householder vectors that form Q_{12} and $Q_{12}R_{12} = \begin{pmatrix} R_1 \\ A_2 \end{pmatrix}$	$2b^3$
_tsmqrt	$\tilde{B}_1,\ \tilde{B}_2 = _\texttt{tsmqrt}(B_1,\ B_2,\ V_{12})$	where $B_1, B_2, V_{12} \in R^{b\times b}$, V_{12} contains the Householder vectors that form Q_{12} and results from _tsqrt and $\begin{pmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{pmatrix} = Q_{12}^T \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$	$4b^3$
_ttqrt	$V_{12},\ R_{12} = _\texttt{ttqrt}(R_1, R_2)$	where $R_1, R_2, V_{12}, R_{12} \in R^{b\times b}$, $R_1,\ R_2$ are upper triangular, V_{12} contains the Householder vectors that form Q_{12} and $Q_{12}R_{12} = \begin{pmatrix} R_1 \\ R_2 \end{pmatrix}$	$\frac{2}{3}b^3$
_ttmqrt	$\tilde{B}_1,\ \tilde{B}_2 = _\texttt{ttmqrt}(B_1,\ B_2,\ V_{12})$	where $B_1, B_2, V_{12} \in R^{b\times b}$, V_{12} contains the Householder vectors that form Q_{12} and results from _ttqrt and $\begin{pmatrix} \tilde{B}_1 \\ \tilde{B}_2 \end{pmatrix} = Q_{12}^T \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$	$2b^3$

Table 1: A list of the various kernels introduced in this document.

	Flat tree	Binary tree
1	$V_{11},\ R_{11}{=}_\texttt{geqrt}(A_{11})$ $\begin{pmatrix} \tilde{R}_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{pmatrix}$	$V_{i1},\ R_{i1} = _\texttt{geqrt}(A_{i1}),\ i = 1,..,4$ $\begin{pmatrix} \tilde{R}_{11} & A_{12} & A_{13} \\ \tilde{R}_{21} & A_{22} & A_{23} \\ \tilde{R}_{31} & A_{32} & A_{33} \\ \tilde{R}_{41} & A_{42} & A_{43} \end{pmatrix}$
2	$\tilde{A}_{1j} = _\texttt{gemqrt}(A_{1j},\ V_{11}),\ j = 2,3$ $\begin{pmatrix} \tilde{R}_{11} & \tilde{A}_{12} & \tilde{A}_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{pmatrix}$	$\tilde{A}_{ij} = _\texttt{gemqrt}(A_{ij},\ V_{i1}),\ i = 1,..,4\ \ j = 2,3$ $\begin{pmatrix} \tilde{R}_{11} & \tilde{A}_{12} & \tilde{A}_{13} \\ \tilde{R}_{21} & \tilde{A}_{22} & \tilde{A}_{23} \\ \tilde{R}_{31} & \tilde{A}_{32} & \tilde{A}_{33} \\ \tilde{R}_{41} & \tilde{A}_{42} & \tilde{A}_{43} \end{pmatrix}$
3	$\hat{R}_{11},\ V_{21} = _\texttt{tsqrt}(\tilde{R}_{11},\ A_{21})$ $\begin{pmatrix} \hat{R}_{11} & \tilde{A}_{12} & \tilde{A}_{13} \\ & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{pmatrix}$	$\hat{R}_{11},\ V_{21} = _\texttt{ttqrt}(\tilde{R}_{11},\ \tilde{R}_{21})$ $\begin{pmatrix} \hat{R}_{11} & \tilde{A}_{12} & \tilde{A}_{13} \\ & \tilde{A}_{22} & \tilde{A}_{23} \\ \tilde{R}_{31} & \tilde{A}_{32} & \tilde{A}_{33} \\ \tilde{R}_{41} & \tilde{A}_{42} & \tilde{A}_{43} \end{pmatrix}$
4	$\hat{A}_{1j},\ \hat{A}_{2j} = _\texttt{tsmqrt}(\tilde{A}_{1j},\ A_{2j},\ V_{21}),\ j = 2,3$ $\begin{pmatrix} \hat{R}_{11} & \hat{A}_{12} & \hat{A}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ A_{31} & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{pmatrix}$	$\hat{A}_{1j},\ \hat{A}_{2j} = _\texttt{ttmqrt}(\tilde{A}_{1j}, \tilde{A}_{2j},\ V_{21}),\ j = 2,3$ $\begin{pmatrix} \hat{R}_{11} & \hat{A}_{12} & \hat{A}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ \tilde{R}_{31} & \tilde{A}_{32} & \tilde{A}_{33} \\ \tilde{R}_{41} & \tilde{A}_{42} & \tilde{A}_{43} \end{pmatrix}$
5	$\ddot{R}_{11},\ V_{31} = _\texttt{tsqrt}(\hat{R}_{11},\ A_{31})$ $\begin{pmatrix} \ddot{R}_{11} & \hat{A}_{12} & \hat{A}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & A_{32} & A_{33} \\ A_{41} & A_{42} & A_{43} \end{pmatrix}$	$\hat{R}_{31},\ V_{41} = _\texttt{ttqrt}(\tilde{R}_{31},\ \tilde{R}_{41})$ $\begin{pmatrix} \hat{R}_{11} & \hat{A}_{12} & \hat{A}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ \hat{R}_{31} & \tilde{A}_{32} & \tilde{A}_{33} \\ & \tilde{A}_{42} & \tilde{A}_{43} \end{pmatrix}$
6	$\ddot{A}_{1j},\ \ddot{A}_{3j} = _\texttt{tsmqrt}(\hat{A}_{1j},\ A_{3j},\ V_{31}),\ j = 2,3$ $\begin{pmatrix} \ddot{R}_{11} & \ddot{A}_{12} & \ddot{R}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & \ddot{A}_{32} & \ddot{A}_{33} \\ A_{41} & A_{42} & A_{43} \end{pmatrix}$	$\hat{A}_{3j},\ \hat{A}_{4j} = _\texttt{ttmqrt}(\tilde{A}_{3j}, \tilde{A}_{4j},\ V_{41}),\ j = 2,3$ $\begin{pmatrix} \hat{R}_{11} & \hat{A}_{12} & \hat{A}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ \hat{R}_{31} & \hat{A}_{32} & \hat{A}_{33} \\ & \hat{A}_{42} & \hat{A}_{43} \end{pmatrix}$
7	$\bar{R}_{11},\ V_{41} = _\texttt{tsqrt}(\ddot{R}_{11},\ A_{41})$ $\begin{pmatrix} \bar{R}_{11} & \ddot{A}_{12} & \ddot{A}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & \ddot{A}_{32} & \ddot{A}_{33} \\ & A_{42} & A_{43} \end{pmatrix}$	$\ddot{R}_{11},\ V_{31} = _\texttt{ttqrt}(\tilde{R}_{11},\ \tilde{R}_{31})$ $\begin{pmatrix} \ddot{R}_{11} & \hat{A}_{12} & \hat{A}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & \hat{A}_{32} & \hat{A}_{33} \\ & \hat{A}_{42} & \hat{A}_{43} \end{pmatrix}$
8	$\bar{R}_{1j},\ \bar{A}_{4j} = _\texttt{tsmqrt}(\ddot{A}_{1j},\ A_{4j},\ V_{41}),\ j = 2,3$ $\begin{pmatrix} \bar{R}_{11} & \bar{R}_{12} & \bar{R}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & \ddot{A}_{32} & \ddot{A}_{33} \\ & \bar{A}_{42} & \bar{A}_{43} \end{pmatrix}$	$\ddot{R}_{1j},\ \ddot{A}_{3j} = _\texttt{ttmqrt}(\hat{A}_{1j}, \hat{A}_{3j},\ V_{31}),\ j = 2,3$ $\begin{pmatrix} \ddot{R}_{11} & \ddot{R}_{12} & \ddot{R}_{13} \\ & \hat{A}_{22} & \hat{A}_{23} \\ & \ddot{A}_{32} & \ddot{A}_{33} \\ & \hat{A}_{42} & \hat{A}_{43} \end{pmatrix}$

Table 2: The first few steps of tiled algorithms with flat and binary reduction tree.