Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

A Pipeline for the *QR* Update in Digital Signal Processing

Manuel F. Dolz[1] | Fran J. Alventosa[1] | Pedro Alonso-Jordá[1] | Antonio M. Vidal[1]

[1]Computer Science and Engineerying Department, Universitat Jaume I, Castellón, Spain
[2]Dept. of Information Systems and Computation, Universitat Politècnica de València, Valencia, Spain

Correspondence
*Pedro Alonso-Jordá. Email: palonso@upv.es

Summary

The input and output signals of a digital signal processing system can often be represented by a rectangular matrix as it is the case of the Beamformer Algorithm, a very useful particular algorithm that allows to extract the original input signal once cleaned from noise and room reverberation. We use a version of this algorithm in which the system matrix must be factorized to solve a least squares problem. The matrix changes periodically according to the input signal sampled and, therefore, the factorization needs to be recalculated as fast as possible. In this paper we propose to use parallelism through a pipeline pattern. With our pipeline, some partial computations are advanced so that the final time required to update the factorization is highly reduced.

KEYWORDS:
QR factorization, QR Update, real-time, block QR, pipeline QR update

1 | INTRODUCTION

Updating a matrix that represents a digital system is one of the most frequent operations in digital signal processing problems. This update usually consists of appending new rows to the bottom of the matrix while taking the same number of rows out of the top. The matrix is usually factorized in order to solve some linear systems or to compute a least squares problem required to generate the output signal. It is, hence, interesting to update the factorization instead of computing the factorization of the matrix from scratch with the aim of saving computations. In our case, we use the *QR* factorization to obtain the upper triangular factor *R* of the matrix to solve a least squares problem. Updating a factorization can be critical if the result is required in real time. This is the case of many signal processing applications such as 3D audio [1, 2], analysis of multiple input/multiple output detection systems (MIMO systems) [3], etc.

In this paper we work on the Beamformer algorithm [4], an algorithm very used in digital sound processing that is executed in real-time. The most time-consuming part of this algorithm involves the *QR* factorization of a rectangular matrix [4]. We deal with a particular case in which the proportion between the number of rows and columns is 4×3. At each step of the algorithm, the matrix changes by losing 25% of the rows at the top and appending the same number of new entry rows at the bottom. Then, a new *QR* factorization must be calculated. The objective is to recompute or update the *QR* factorization of this "changing" matrix as fast as possible at each processing time. Although we work on the particular case of a 4×3 blocks matrix, all the discussion is generalizable to other different matrix shapes.

Updating the *QR* factorization is a recurrent operation deeply studied in the past. There exist algorithms proposed that reduce the computational cost of the factorization in one order of magnitude [5]. However, these algorithms require to build and update the whole *Q* factor, even in the case this factor is not required by the application. Building Q implies, not only memory consumption, but also some extra computations, to the point that using this tool is useful only if the number of new rows appended

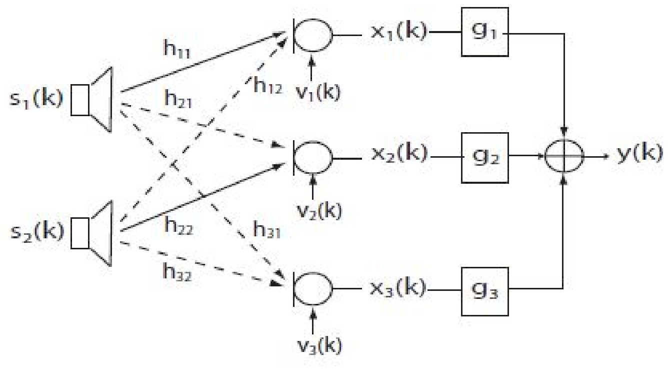


FIGURE 1 Beamformer model

to the bottom of the system matrix are very few (4 or 5) [6]. Thus, it is not clear that this technique to update the *QR* factorization is useful with regard to computing the *QR* factorization from scratch in many applications where the *Q* factor is not required.

The solution to the Beamforming problem can be approximated in different ways. Here, we use the one proposed in [7], whose mathematical background is described in Section 2. In the following sections we show how to tackle the solution of the main computational step, which is the *QR* factorization of a matrix that describes the digital system. We show, in Section 3 how to work on the whole rectangular matrix to perform its *QR* factorization in parallel. Next, in Section 4, we revisit the solution to this problem by using a matrix called *jagged*. Using this type of matrix permits to reduce the computational cost of the factorization and it is the basis for our proposed pipeline. The proposed idea of this paper is explained in Section 5, where we show how to reduce the cost of the factorization using a *pipeline* structure. The paper closes with some conclusions.

2 | THE BEAMFORMER ALGORITHM

The problem of the Beamformer can be described as shown in Figure 1. The figure represents a particular case with two speakers, each one emitting a different signal, and an array of three microphones located at different points of the room. The sound signal coming from the sound sources (loudspeakers) are captured by the microphones. Basically, the digital system should built those filters, real-time, that allow to extract a signal that is coming from a certain direction. There exists a strong constraint for this physical system to work: the number of listeners (microphones) must be larger than the number of sources (loudspeakers).

The Beamformer model can be formally described as follows. In this model, each S_m, for $m \in 1, 2$, represents a speaker, and each V_n, for $n \in 1, 2, 3$, represents a microphone which is located at a different point into the room. The channel between the m-th speaker and the n-th microphone is represented by a variable denoted as h_{mn}. The filter to be applied to microphone z is denoted by the array g_z, for $z \in 1, 2, 3$, and it is the filter calculated so that the sought-after signal k can be obtained. Array $y(k)$, finally, represents the output signal resulting from applying the computed filter g to the signals captured on microphones.

Thus, the goal is to develop N filters g_n, $n = 1, \dots, N$, being N the number of microphones of the system, that allow to rebuild the original source signals $s_m(k)$, $m = 1, \dots, M$, emitted by M loudspeakers in order to get a signal cleaned from room reverberation and noise. To build the system, we need the channel responses of the room, which are represented as h_{nm}.

The output of the n-th microphone is given by:

$$x_n(k) = \sum_{m=1}^{M} \sum_{j=0}^{L_h-1} h_{nm}(j) s_m(k-j) + v_n(k) ,$$

being L_h the length of longest room impulse response of all the acoustic channels h_{nm}, and $v_n(k)$ the noise signal (see [7] for more details). In the definition of the problem it is quite usual, for the sake of clarity, to get rid of the noise term. Also for clarity, it is usual to transform the previous formulation into a matrix/vector form, so that the output signal of each microphone can be written as

$$x_n(k) = \sum_{m=1}^{M} \mathbf{h}_{nm}^T \mathbf{s}_m(k) ,$$

where $\mathbf{s}_m(k)$ is the column vector defined as

$$\mathbf{s}_m(k) = \left[\, s_m(k)\;\; s_m(k-1)\; \cdots \; s_m(k-L_h+1)\right]^T \;,$$

and $\mathbf{h}_{nm}$ is the $\mathbb{R}^{L_h \times 1}$ acoustic channel vector from loudspeaker m to microphone n.

Given the recorded observations $x_n(k)$ the problem of recovering source signals $s_m(k)$ can be described as follows. The aim is to get an output signal $y(k)$ which is a good estimate of $s_m(k)$, i.e. $y(k) = \hat{s}_m(k-\tau)$, for $\tau = k, k-1, \ldots, k-L_h+1$, with minimum error. Thus, the Beamforming filters g_n are designed with this in mind. Let L_g be the maximum length of taps for each of the N filters g_n, the broadband Beamforming output signal can be expressed as

$$y(k) = \sum_{n=1}^{N} \mathbf{g}_n^T \mathbf{x}_n(k) \;,$$

where $\mathbf{g}_n$ is the $\mathbb{R}^{L_g \times 1}$ vector containing the ordered taps of Beamforming filters g_n, and $\mathbf{x}_n(k) = [x_n(k) x_n(k-1) \cdots x_n(k-L_g+1)]^T$. The concatenation of filters $\mathbf{g}_n$ form the solution array $\mathbf{g}^{\mathrm{LCMV}} = [\mathbf{g}_1^T, \ldots, \mathbf{g}_N^T]^T$. This formulation of the Beamformer problem is called *Linearly Constrained Minimum Variance* or LCMV Algorithm.

The solution $\mathbf{g}^{\mathrm{LCMV}}$ is calculated as follows. Let $\mathbf{A} \in \mathbb{R}^{K \times N L_g}$ be the following matrix

$$\mathbf{A} = \frac{1}{\sqrt{K}} \begin{pmatrix} \mathbf{x}_1^T(k) & \mathbf{x}_2^T(k) & \ldots & \mathbf{x}_N^T(k) \\ \mathbf{x}_1^T(k+1) & \mathbf{x}_2^T(k+1) & \ldots & \mathbf{x}_N^T(k+1) \\ \vdots & \vdots & & \vdots \\ \mathbf{x}_1^T(k+K-1) & \mathbf{x}_2^T(k+K-1) & \ldots & \mathbf{x}_N^T(k+K-1) \end{pmatrix} , \tag{1}$$

where K ($> N L_g$) represents the number of samples. By $\mathbf{C} = \mathbf{A^T A}$ we represent the correlation matrix of the recorded signals. The impulse responses from the m-th source to the N microphones, used in *Sylvester* matrix form, are represented though matrix $\mathbf{H}^{(N L_g) \times (L_g + L_h - 1)}$, which is a partition of the channel impulse matrix. Then, the Beamformer filters can be calculated as:

$$\mathbf{g}^{\mathrm{LCMV}} = \mathbf{C^{-1} H} \left[\mathbf{H^T C^{-1} H}\right]^{\mathbf{-1}} \mathbf{u_m} \;, \tag{2}$$

where $\mathbf{u}_m$ is a vector set to zero except for one entry which is set to one at the proper position in order to compensate the room impulse response delay [7].

The LCMV Algorithm is based on the *QR* factorization with the aim of achieving efficiency and accuracy. Let $\mathbf{A} = \mathbf{QR}$ be the *QR* factorization of $\mathbf{A}$, where $\mathbf{Q}$ is orthogonal and $\mathbf{R}$ is upper triangular, then $\mathbf{C} = \mathbf{R^T R}$ and, consequently, $\mathbf{C^{-1}} = \mathbf{R^{-1} R^{-T}}$, $\mathbf{R} \in \mathbb{R}^{(N \cdot L_g) \times (N \cdot L_g)}$. Equation (2) can be rewritten as

$$\mathbf{g}^{\mathrm{LCMV}} = \mathbf{R}^{-1}\mathbf{R}^{-T}\mathbf{H}\left[\mathbf{H}^T\mathbf{R}^{-1}\mathbf{R}^{-T}\mathbf{H}\right]^{-1}\mathbf{u}_m \;.$$

If $\mathbf{Z} = \mathbf{R}^{-T}\mathbf{H}$ then $\mathbf{g}^{\mathrm{LCMV}} = \mathbf{R}^{-1}\mathbf{Z}\left[\mathbf{Z}^T\mathbf{Z}\right]^{-1}\mathbf{u}_m$. Let $\mathbf{Z} = \mathbf{PL}$ be the *QR* factorization of $\mathbf{Z}$, where $\mathbf{P^T P} = \mathbf{P^T P} = \mathbf{I}$ and $\mathbf{L}$ is upper triangular, then $\mathbf{g}^{\mathrm{LCMV}} = \mathbf{R}^{-1}\mathbf{PL}\left[\mathbf{L^T L}\right]^{\mathbf{-1}}\mathbf{u_m} = \mathbf{R^{-1}PLL^{-1}L^{-T}u_m} = \mathbf{R^{-1}PL^{-T}u_m}$. If we set $\mathbf{PL^{-T}u_m} = \mathbf{b}$, vector b can be obtained by computing

$$\begin{aligned} \mathbf{L^T y} &= \mathbf{u}_m \;, \\ \mathbf{P y} &= b \;. \end{aligned}$$

Finally, the Beamformer filter presented in (2) can be obtained by solving the following upper triangular system

$$\mathbf{g}^{\mathrm{LCMV}} = \mathbf{R}^{-1} b \;.$$

The above mentioned elaboration can be summarized in the following algorithm:

1. Obtain $\mathbf{R}$ from $\mathbf{A} = \mathbf{QR}$.
2. Solve $\mathbf{R}^T\mathbf{Z} = \mathbf{H}$ for $\mathbf{Z}$.
3. Obtain $\mathbf{L}$ from $\mathbf{Z} = \mathbf{PL}$.
4. Solve $\mathbf{L^T y} = \mathbf{u_m}$ for $\mathbf{y}$.
5. Obtain $b = \mathbf{Py}$.
6. Solve $\mathbf{R g}^{\mathrm{LCMV}} = b$ for $\mathbf{g}^{\mathrm{LCMV}}$.

$$A^{(k)} = \begin{array}{|c|c|c|}\hline A_{-1,0} & A_{-1,1} & A_{-1,2} \\ \hline A_{0,0} & A_{0,1} & A_{0,2} \\ \hline A_{1,0} & A_{1,1} & A_{1,2} \\ \hline A_{2,0} & A_{2,1} & A_{2,2} \\ \hline \end{array} \quad \Rightarrow \quad A^{(k+1)} = \begin{array}{|c|c|c|}\hline A_{0,0} & A_{0,1} & A_{0,2} \\ \hline A_{1,0} & A_{1,1} & A_{1,2} \\ \hline A_{2,0} & A_{2,1} & A_{2,2} \\ \hline A_{3,0} & A_{3,1} & A_{3,2} \\ \hline \end{array}$$

FIGURE 2 Updating of system matrix A from iteration k to $k+1$.

3 | THE TILED *QR* FACTORIZATION ALGORITHM

The main computational step of the process described above is the *QR* factorization of matrix A (1). As it was shown in [4], this step can approach 80% of the total computational time. A natural step to improve computation consists of using an algorithm-by-blocks. Following the out-of-core algorithm in [8] it is easy to obtain the idea for an algorithm-by-blocks that solve the *QR* factorization problem. The approach followed in this work was based on the techniques introduced in [9]. Furthermore, we modified the code developed in [9] to progressively incorporate our proposals. The basic code views a matrix to operate with as a partition of submatrices (*tiles*) that are set as units of data (algorithms-by-blocks).

Let's consider a matrix as a collection of square *tiles* of size $t_s \times t_s$. Each *tile*, although represents a square block, is really an array of consecutive stored in memory elements. A previous transformation step consisting of reorganizing data can be necessary if the matrix to be factorized is in the usual column-major order format required by BLAS/LAPACK routines (or another regular format). In this case, we need to transform the matrix to a collection of *tiles*. Periodically, the Beamformer algorithm builds matrix $A^{(k+1)}$ by deleting a bunch of t_s rows at the top of matrix $A^{(k)}$ and appending the same number of rows at the bottom using data that has been accumulated with the last samples. Our system matrix A, however, is progressively built with the new upcoming information that will become the new bottom rows of A. This new data will be packed into *tiles* properly so that this step can be integrated and hidden into the usual data acquisition process.

For the case addressed here, the matrix is partitioned in 4×3 *tiles* (Figure 2). However, all the following discussion is also valid for any other grid arrangement of *tiles*. There exist another assumption, in addition, about the problem size, which must be always multiple of the *tile* size. This is not a hard constraint, however, since the underlying physical problem allows a certain degree of freedom in the choice of the *tile* size.

A *tile QR* factorization algorithm, `QRTiled(A)`, can be described as shown in Algorithm 1. This algorithm, which is also annotated with OpenMP macros, is a particular specification of a *tile* algorithm for the *QR* factorization of an `M` × `N` *tiled* matrix. The algorithm can be used, e.g. to perform the factorization of a *tiles* matrix like the one shown in Figure 2. We use OpenMP [10], which allows, through its set of compiler directives, library routines, and environment variables, to write high-level parallel programs for a shared memory architecture like multicores. The sequential version arises from simply deleting the OpenMP directives.

There exist four different task types identified in Algorithm 1. (These types are already identified and taken from [9]). We describe them next, taking into account that the superscript drops, so that A denotes $A^{(k)}$ or $A^{(k+1)}$.

D_QR: This task type computes the *QR* factorization of a square *tile*. In the algorithm, *tiles* $A_{k,k}$ are replaced by the resulting factor, i.e. the upper triangular factor R of the *QR* factorization of the pivot *tile*. This operation is really a call to `xpotrf` LAPACK routine.

D_QT: This task type multiplies the factor Q^T obtained by the previous task type (**D_QR**) applied to the diagonal block $A_{k,k}$ to *tile* $A_{k,j}$. LAPACK routine `xpotrf` stores the Householder reflectors of Q^T into the lower triangle of $A_{k,k}$. This task type uses them to pre- multiply $A_{k,j}$ by Q in the same way as LAPACK routine `xormqr` does. This operation comprises the execution of the outermost loop indexed by `j`.

Algorithm 1 `QRTiled(A)`: factorizes a rectangular matrix `A` partitioned in square *tiles*.

```
#pragma omp parallel
#pragma omp single private(i,j,k)
for( k = 0; k < N; k++ ) {
  #pragma omp task depend( inout: A(k,k) )
  D_QR( A(k,k) )
  for( j = k+1; j < N; j++ ) {
    #pragma omp task depend( in: A(k,k) ) depend( inout: A(k,j) )
    D_QT( A(k,k), A(k,j) )
  }
  for( i = k+1; i < M; i++ ) {
    #pragma omp task depend( inout: A(k,k), A(i,k) )
    TD_QR( A(k,k), A(i,k) )
    for( j = k+1; j < N; j++ ) {
      #pragma omp task depend( in: A(i,k) ) depend( inout: A(k,j), A(i,j) )
      TD_QT( A(i,k), A(k,j), A(i,j) )
    }
  }
}
```

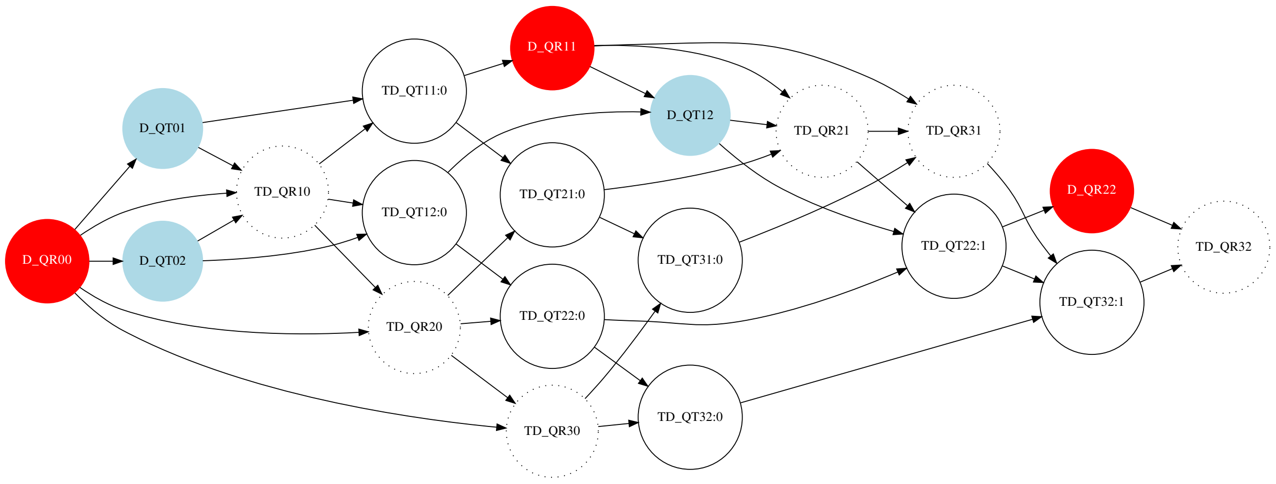


FIGURE 3 Directed Acyclic Graph of tasks featuring the *QR* factorization algorithm of a rectangular matrix of 4×3 *tiles*.

`TD_QR`: This task type computes the *QR* factorization of $\begin{pmatrix} A_{k,k} \\ A_{i,k} \end{pmatrix}$. As a result, *tile* $A_{k,k}$ is replaced by the resulting upper triangular factor and *tile* $A_{i,k}$ is completely zeroed. *Tile* $A_{k,k}$ is also modified by this operation.

`TD_QT`: This task type multiplies the factor Q^T obtained by the previous task type, i.e. **`TD_QR`**, to the *tiles* row i. This task type uses the Householder reflectors stored into *tile* $A_{i,k}$. Both *tiles* $A_{k,j}$ and $A_{i,j}$ are modified by this operation.

The parallel version of Algorithm 1, i.e. when it is compiled with the OpenMP compiler option activated, uses *tasks* with dependencies (`depend` clause). Parallel algorithms designed from the task identification and partitioning point of view, transcend the loop parallelization bounds and allow to build a DAG (Directed Acyclic Graph) of tasks which are executed by the OpenMP runtime as their dependencies are met. Figure 3 shows the DAG generated by Algorithm 1 when performing the *QR* factorization of a 4×3 *tiles* matrix. The four types of tasks are identified by different colors or border line shapes. Also, the numbers identifies the indices (`i`, `j`, or `k`) of the main *tile* that is being modified by the task. After the colon, if it exists, there is a number which denotes the iteration in which the task has been modified, in case this task is modified several times at different iterations.

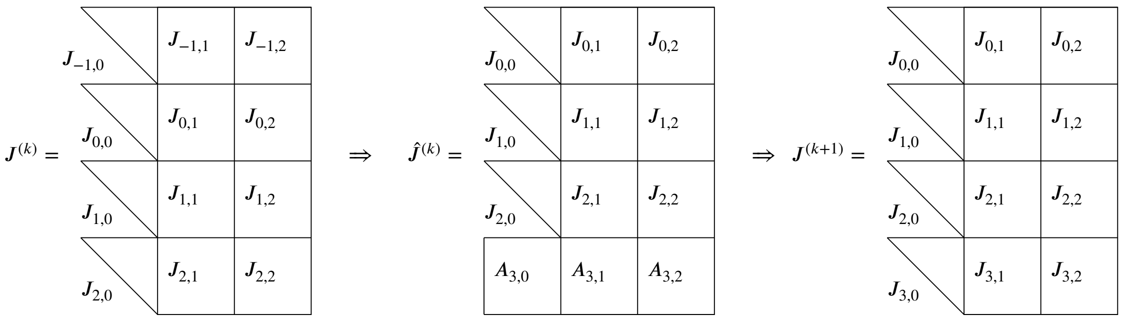


FIGURE 4 Updating the *jagged* system matrix J from iteration k to $k+1$. Matrix $\hat{J}^{(k)}$ is formed between *jagged* matrices $J^{(k)}$ and $J^{(k+1)}$ in the process.

4 | USING *JAGGED* MATRICES TO IMPROVE THE *QR* FACTORIZATION

As it has been seen previously, the process is iterative. Let's denote by k, $k = 0, 1, \dots$, each iteration. At each iteration, system matrix $A^{(k)}$ (Figure 2) is "updated" with new data to form the matrix of the next iteration, i.e. $A^{(k+1)}$. The new system matrix is formed by deleting the top t_s rows ("oldest" rows) of $A^{(k)}$, and a bunch of t_s new rows ("newest" rows), built with information sampled from the signal, is appended to the bottom of $A^{(k)}$.

Computing the *QR* factorization from scratch is an expensive operation so, with the aim of saving flops, we proposed in [11] to work on a different type of matrix that we called *jagged*. A *jagged* matrix is a rectangular matrix of square *tiles* where the first column is made of upper triangular *tiles*. The idea is to keep the system matrix in this form from one iteration to the next, so that matrices of the form shown in Figure 2 are now always kept on the form shown in Figure 4 (left and right matrices). It is not difficult to guess that the upper triangular blocks at the first column of $J^{(k)}$ are going to be the upper triangular factor (R) of the *QR* factorization of the corresponding block in $A^{(k)}$. To obtain matrix $J^{(k+1)}$ from matrix $J^{(k)}$, we depict an "intermediate" (middle matrix in Figure 4) that represents the three new $t_s \times t_s$ *tiles* appended at the bottom of $J^{(k)}$ (that would lead to form $A^{(k+1)}$ in the previous case). Now, we need to form $J^{(k+1)}$ instead of $A^{(k+1)}$. This is simply carried out by computing the *QR* factorization of these new t_s rows, i.e. the *QR* factorization of $\begin{pmatrix} A_{3,0} & A_{3,1} & A_{3,2} \end{pmatrix}$, what leads us to obtain the last t_s rows of $J^{(k+1)}$, i.e. $\begin{pmatrix} J_{3,0} & J_{3,1} & J_{3,2} \end{pmatrix}$. Keeping the system matrix in this form results in some savings in computational cost.

Algorithm 2 shows the steps to obtain the *QR* factorization of a *jagged* matrix. The first operations, lines 4–9, are devoted to the computation of the *QR* factorization of $\begin{pmatrix} A_{3,0} & A_{3,1} & A_{3,2} \end{pmatrix}$ that results in $\begin{pmatrix} J_{3,0} & J_{3,1} & J_{3,2} \end{pmatrix}$, i.e. the *QR* factorization of the bottom t_s rows of $\hat{J}^{(k)}$ (middle matrix in Figure 4). The algorithm proceeds, lines 10–17, by performing the *QR* factorization of the first block column of $J^{(k+1)}$ (Figure 5a). The remaining computation is carried out by `QRTiled` (Algorithm 1), through the call shown in line 18, to reduce submatrix $J_{1:M-1,1:N-1}$ to upper triangular (Figure 5b).

Algorithm 2 uses two more different type of tasks, apart from those included in Algorithm 1:

`TD_QR_T`: This task type is, somehow, kindred to task type **`TD_QR`** for the case in which the matrix to reduce to upper triangular form is $\begin{pmatrix} A_{k,k} \\ J_{i,k} \end{pmatrix}$, i.e. when the "lower" factor, $J_{i,k}$, is triangular.

`TD_QT_T`: This task type is the counterpart of **`TD_QT`** for the case in which factor Q^T was generated with the previous task, i.e. **`TD_QR_T`**.

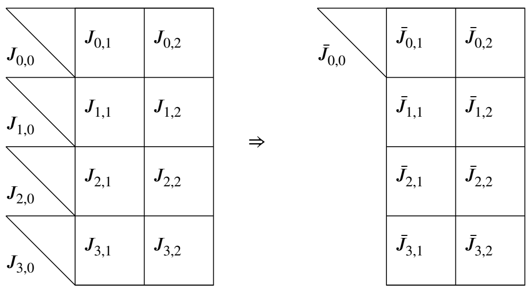
The execution of Algorithm 2 in parallel, i.e. with OpenMP activated, results in a DAG like the one shown in Figure 6 for the case of *jagged* matrices of size 4×3. Note that the DAG in Figure 6 does not represent the step carried out to obtain $J^{(k+1)}$ from $\hat{J}^{(k)}$ (middle matrix in Figure 4), i.e. steps 4–9. The number of tasks is the same as in the reduction to upper triangular form of rectangular matrices (Figure 3). However, the computational cost has been reduced thanks to the zeros introduced at the first column of *tiles* in the *jagged* matrix. As it was demonstrated in [11], the cost of reducing a *jagged* matrix to upper triangular form using a *QR* factorization can be approximated by

$$T_J = T_A - T_S = 2n^2\left(m - \frac{n}{3}\right) - \left(\frac{4}{3}t_s^3 - \left(\frac{4}{3}m + 2n\right)t_s^2 + 2mnt_s\right).$$

The cost (in flops) of performing the *QR* factorization of a rectangular matrix A of size $m \times n$ (Figure 2) is denoted by T_A here [12]. Member T_S, otherwise, denotes the savings obtained thanks to the use of *jagged* matrices.

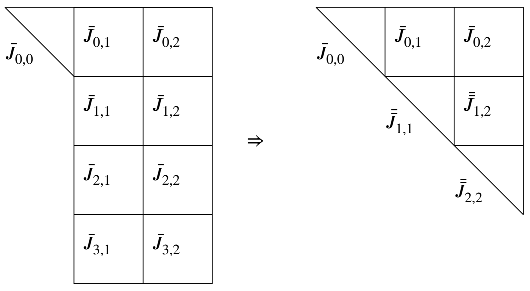
Algorithm 2 `QRTiledJagged(J)`: performs the QR factorization of a matrix of the form $\hat{J}^{(k)}$ (middle matrix in Figure 4).

```
#pragma omp parallel
#pragma omp single private(i,j,k)
{
  #pragma omp task depend( inout: J(M-1,0) )
  D_QR( J(M-1,0) );
  for( j = 1; j < N; j++ ) {
    #pragma omp task depend( in: J(M-1,0) ) depend( inout: J(M-1,j) )
    D_QT( J(M-1,0), J(M-1,j) );
  }
  for( i = 1; i < M; i++ ) {
    #pragma omp task depend( inout: J(0,0), J(i,0) )
    TD_QR_T( J(0,0), J(i,0) );
    for( j = 1; j < N; j++ ) {
      #pragma omp task depend( in: J(i,0) ) depend( inout: J(0,j), J(i,j) )
      TD_QT_T( J(i,0), J(0,j), J(i,j) );
    }
  }
  QRTile(J(1:M-1,1:N-1)); /* Algorithm 1 */
}
```



$J_{0,0}$ $J_{0,1}$ $J_{0,2}$
$J_{1,0}$ $J_{1,1}$ $J_{1,2}$
$J_{2,0}$ $J_{2,1}$ $J_{2,2}$
$J_{3,0}$ $J_{3,1}$ $J_{3,2}$
$\Rightarrow$
$\bar{J}_{0,0}$ $\bar{J}_{0,1}$ $\bar{J}_{0,2}$
$\bar{J}_{1,1}$ $\bar{J}_{1,2}$
$\bar{J}_{2,1}$ $\bar{J}_{2,2}$
$\bar{J}_{3,1}$ $\bar{J}_{3,2}$

(a) Reduction of the first block column of matrix $J^{(k+1)}$ to upper triangular form.



$\bar{J}_{0,0}$ $\bar{J}_{0,1}$ $\bar{J}_{0,2}$
$\bar{J}_{1,1}$ $\bar{J}_{1,2}$
$\bar{J}_{2,1}$ $\bar{J}_{2,2}$
$\bar{J}_{3,1}$ $\bar{J}_{3,2}$
$\Rightarrow$
$\bar{J}_{0,0}$ $\bar{J}_{0,1}$ $\bar{J}_{0,2}$
$\bar{\bar{J}}_{1,1}$ $\bar{\bar{J}}_{1,2}$
$\bar{\bar{J}}_{2,2}$

(b) Reduction of submatrix $\bar{J}^{(k+1)}_{1:M-1,1:N-1}$ to upper triangular form.

FIGURE 5 Reduction of a *jagged* matrix to upper triangular form.

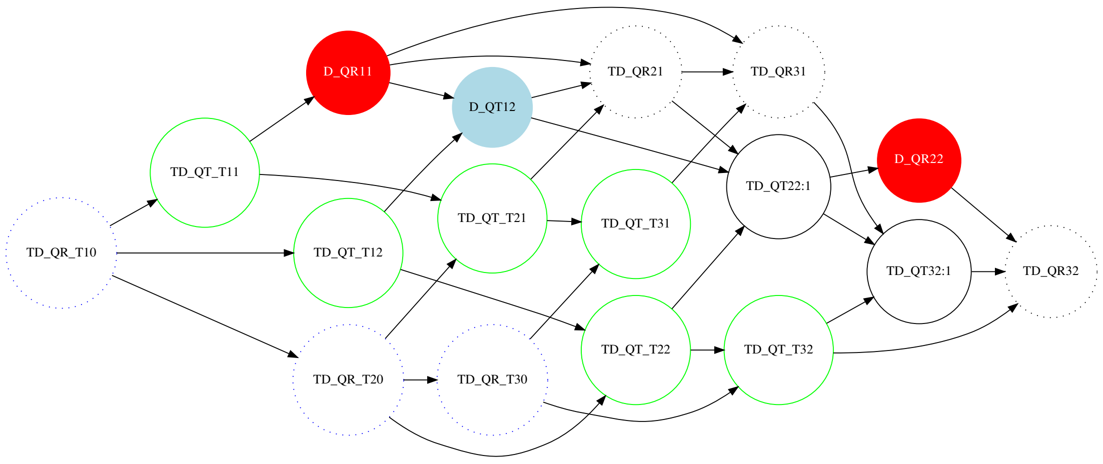


FIGURE 6 Directed Acyclic Graph of tasks featuring the QR factorization algorithm of a *jagged* matrix of 4×3 *tiles*.

The next expression represents the speed-up (S) achieved with the use of *jagged* matrices, that is the time to obtain the QR factorization of matrix *jagged* $\hat{J}^{(k)}$ (middle matrix in Figure 4) compared to the time needed to obtain the QR factorization of a

TABLE 1 Theoretical speed-up obtained when working on *jagged* matrices compared with the algorithm that works on rectangular matrices.

$m \times n$ / t_s	1	2	8	32	80	160	320	480	640	960
1280×960	1.00	1.00	1.01	1.04	1.11	1.21	1.35	–	–	–
2560×1920	1.00	1.00	1.01	1.02	1.06	1.11	1.21	–	1.35	–
3840×2880	1.00	1.00	1.00	1.01	1.04	1.07	1.14	1.21	–	1.35

rectangular matrix $A^{(k+1)}$ (Figure 2),

$$S = \frac{T_A}{T_J} = \frac{T_A}{T_A - T_S} = \frac{2n^2(m - \frac{n}{3})}{2n^2\left(m - \frac{n}{3}\right) - \left(\frac{4}{3}t_s^3 - \left(\frac{4}{3}m + 2n\right)t_s^2 + 2mnt_s\right)} \tag{3}$$
$$= \frac{1}{1 - \left(\frac{2}{3}t_s^3 - \left(\frac{2}{3}m + n\right)t_s^2 + mnt_s\right) / \left(n^2(m - \frac{n}{3})\right)} .$$

We draw, here, the attention of the reader on an interesting aspect. The point is that the savings obtained thanks to the use of *jagged* matrices are larger than the proportion of zeros introduced in the system matrix to form that *jagged* shape with respect to the matrix size. The speed-up S (3) can help to understand this point but, since the speed-up does expression is complicated due to it does not vary linearly with t_s, we provide numerical examples to better understand the savings achieved. Given some values of the block size t_s for two different matrix sizes, Table 1 shows the speed-up obtained with (3). Note that, for a 4×3 *tiles* matrices of size 1280×960, 2560×1920, and 3840×2880 the *tile* size is $t_s = 320$, $t_s = 640$, and $t_s = 960$, respectively. In all cases is achieved the largest speed-up of 1.35.

5 | THE PIPELINE TO UPDATE THE *QR* FACTORIZATION

In this paper we propose reducing the computational cost of updating the *QR* factorization of matrices of 4×3 *tiles* which appear in the Bemformer problem (Figure 1). One of the main limitations of the previous algorithms is the low number of processors/cores that can be used concurrently to solve the problem. The *QR* factorization in the Beamformer Algorithm is a repeated operation carried out recurrently over modified matrices with input data. We use this fact to design a pipeline structure of tasks that allows to exploit more processors. Furthermore, reusing data calculated in previous stages of the pipeline, the total time needed to compute the upper triangular form (R) of the current matrix is reduced.

To understand the idea, let's firstly assume that we have a *jagged* matrix that represents the system matrix (A) and the upper triangular factor (R_A) of its *QR* factorization (Figure 7a). Given a set of new rows represented by a 1×3 *tiles* matrix ($\bar{W}$) (Figure 7b), the first step of the algorithm is to compute its *QR* factorization to obtain W. Then, the 1×3 *tiles* submatrix on the top (S) is discarded, the new *jagged* submatrix (W) is appended at the bottom, and the *QR* factorization of A' is computed to obtain $R_{A'}$ (Figure 7c). As we saw in Section 4, using *jagged* matrices instead of the original rectangular allows to speed up computations up to 1.35 (Table 1).

When we identify the suboperations necessary to factorize each updated matrix, it can be seen that some of these suboperations can be reused and/or executed in advance. This fact, together with the availability to use concurrent threads, motivates the construction of the proposed pipeline structure. Figure 8 shows the process followed when a new bunch of t_s rows is coming (matrices $\bar{V}$, $\bar{W}$, $\bar{X}$, and $\bar{Y}$). Each row of matrices represents the stages followed to reduce a *jagged* matrix to upper triangular form using the *QR* factorization. For instance, the second matrix at the first row represents system matrix A in Figure 7a. Stage 2 consists of reducing the submatrix formed by blocks S and T to the one formed by blocks $\bar{S}_1$ and $\bar{T}_2$. At the next stage, the submatrix formed by blocks $\bar{S}_1$, $\bar{T}_2$, and U is reduced to the one formed by blocks $\bar{\bar{S}}_1$, $\bar{\bar{T}}_2$, and $\bar{U}_3$. Analogously, the last stage produces the sought-after upper triangular factor $R_A = \left(S_1^T \; T_2^T \; U_3^T \right)^T$ of Figure 7a. In the same way, the second row represents the new system matrix A' (Figure 7c) resulted by discarding T from A and appending W, which is the result of reducing the new bunch of rows $\bar{W}$ (Figure 7b) to a trapezoid by its *QR* factorization. Hence, at the last stage we get $R_{A'}$ (Figure 7c). The rest of the rows of Figure 8 represent the following system matrices in *jagged* form, i.e. matrices formed when the first block is discarded and a new one is appended.

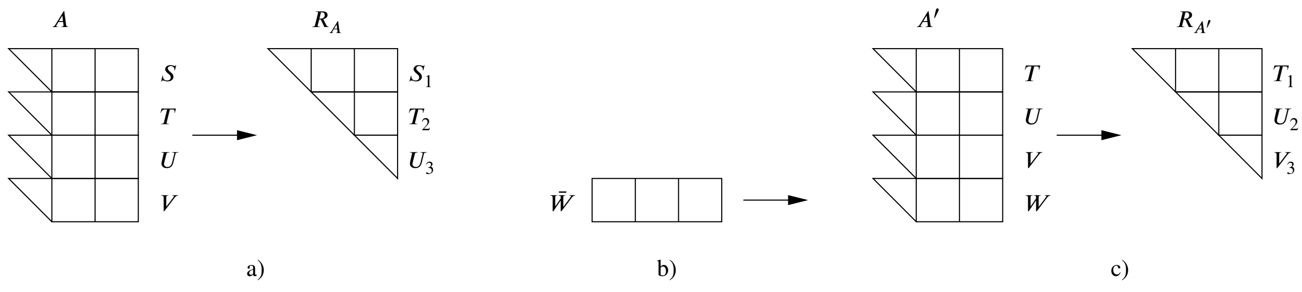


FIGURE 7 *QR* factorization update on *jagged* matrices.

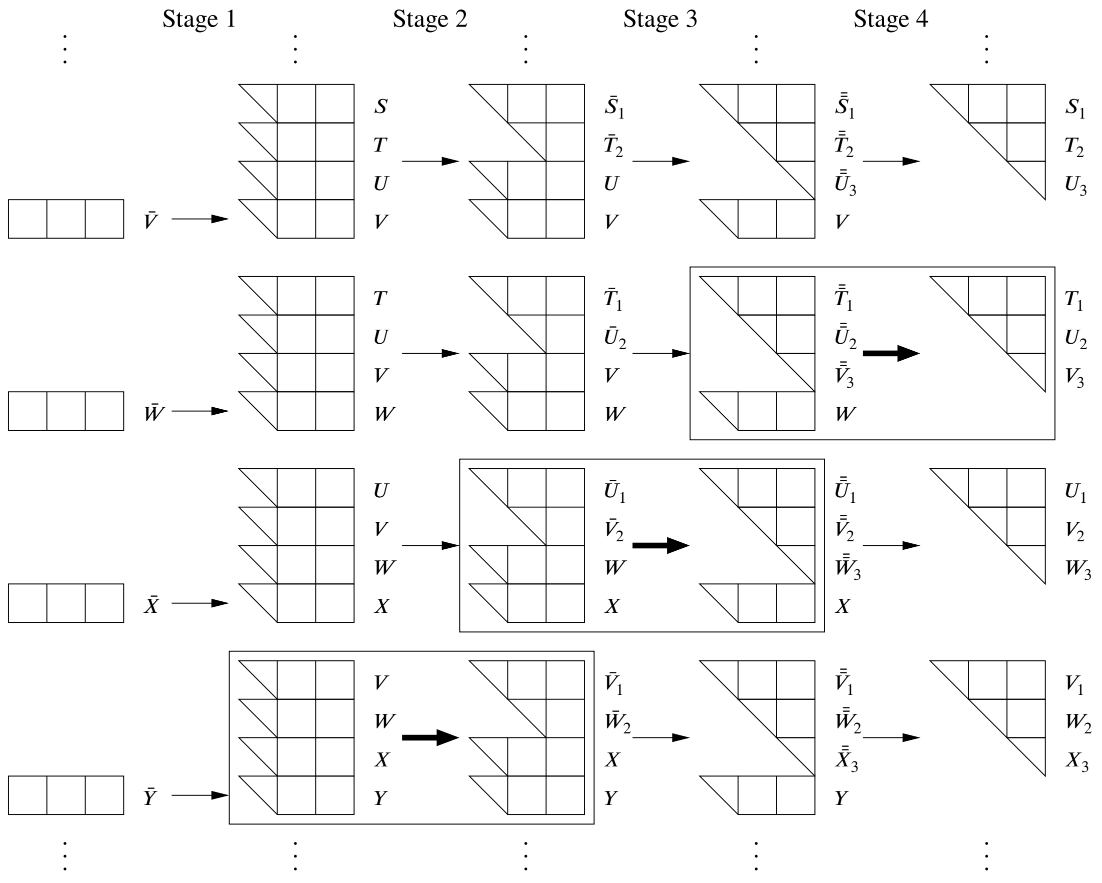


FIGURE 8 Reduction to upper triangular form through *QR* factorization of the system matrix A, in *jagged* form, in different steps of the algorithm.

The objective is to obtain the *QR* factorization of matrix A' (Figure 7), i.e. $R_{A'}$, as fast as possible. This can be accomplished if the upper triangular factor $\left(\bar{\bar{T}}_1^T \ \bar{\bar{U}}_2^T \ \bar{\bar{V}}_3^T \right)^T$ of the *QR* factorization of the upper square submatrix of A', i.e. $\left(T^T \ U^T \ V^T \right)^T$, has already been computed when matrix $\bar{W}$ is available. A key factor that can be observed in Figure 8, and it is useful for the pipeline design, is that those framed stages operate on the same submatrix W and the three of them can be executed concurrently. We have used this fact to design a pipeline structure like the one shown in Figure 9. The figure shows four steps of execution of the pipeline at each horizontal line. At Step 1, the output of the pipeline is matrix R_A (Figure 7). Let's assume that matrices represented into the stages of the pipeline at Step 1 all have been computed previously, then we can see that Step 2 represents the operation in which Stage 1 broadcasts factor W to the following three stages. The computations are carried out at the third step of the pipeline, where those matrices built at Step 2 are reduced through a *QR* factorization to the triangular (or trapezoidal)

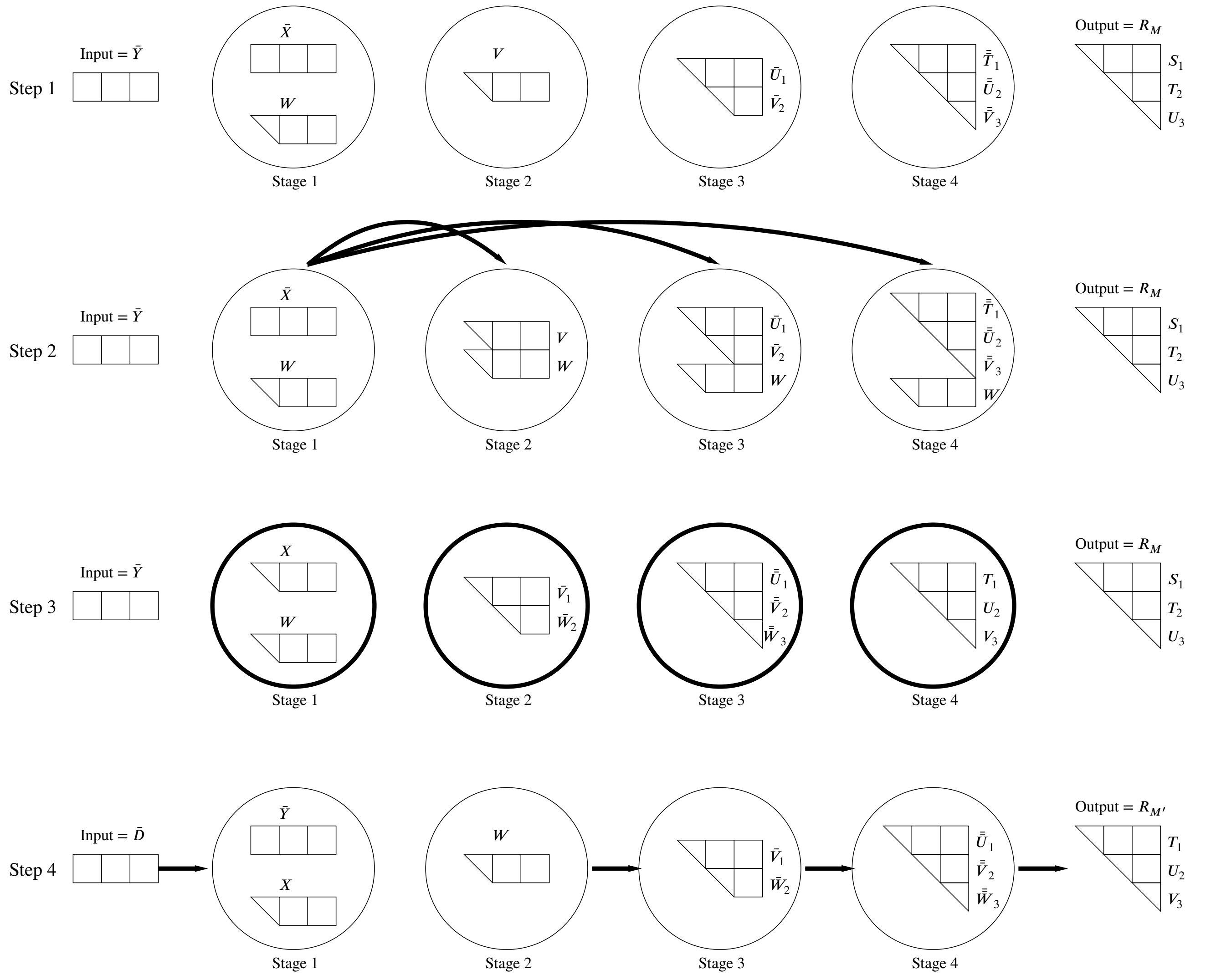


FIGURE 9 Pipeline for the *QR* factorization update.

matrix represented inside the bold-line circles. Note that these computational steps are those represented in the framed stages of Figure 8. At the last step, all stages pass their factor to the right-hand side stage. At this step, the last stage issues the sought-after triangular factor at the output, i.e. $R_{A'}$ (Figure 7). Notice that Step 1 and Step 4 are really the same but with different data.

According to [12], applying a Householder Reflection to nullify the $\mu - 1$ last components of a column of a rectangular $\mu \times \nu$ matrix has a cost of $6\mu + 4\mu\nu$ flops. Provided the triangularizations are carried out by means of Householder Reflections the computational cost of performing Stage 4 of the pipeline (Figure 9) can be approximated by two terms,

$$\sum_{i=1}^{t_s} 6(i+1) + 4(n-i)(i+1) + \sum_{i=1}^{m} 6(t_s+1) + 4(t_s+1)(m-i) \ . \tag{4}$$

The first one represents the cost of zeroing the bottom $t_s \times t_s$ triangular block and, the second one, represents the cost of zeroing the remaining rectangular $t_s \times (n - t_s)$ bottom block. After some arithmetic operations, the total number of flops can be approximated as $2t_s n^2 + 2t_s^3 - 2nt_s^2$ flops.

In order to obtain the total cost, we must add to (4) the cost of triangulating factor $\bar{W}$, which is $2t_s^2 n$ flops (this cost can be deduced in the same way as (4)). If we account for the 4×3 relationship between the number of row and column *tiles*, we have the following costs: $54t_s^3$ flops to reduce a rectangular matrix of the form shown in Figure 2 to upper triangular, $40t_s^3$ flops to

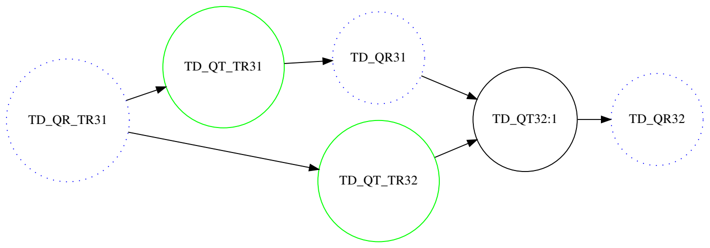


FIGURE 10 Directed Acyclic Graph of tasks featuring the *QR* factorization algorithm of the matrix represented in Stage 4 of the pipeline in Figure 9.

TABLE 2 Time in seconds to perform the *QR* factorization of a rectangular matrix, a *jagged* matrix, and using the pipeline for Figure 9 for different matrix sizes.

		Execution time			Speed up		
$m \times n$	Tile size	*Rectangular*	*Jagged*	*Pipeline*	*Rec./Jag.*	*Jag./Pip.*	*Rec./Pip.*
1280×960	320	0.137 s.	0.102 s.	0.046 s.	1.34	2.22	2.98
2560×1920	640	0.863 s.	0.653 s.	0.297 s.	1.32	2.20	2.91
3840×2880	960	2.754 s.	2.022 s.	0.988 s.	1.36	2.05	2.79

reduce the *jagged* matrix A to R_A (Figure 7), and $20t_s^3$ flops to reduce a matrix of the form shown in Stage 4 of Figure 9 to upper triangular form, including the computation of W as well.

A parallel algorithm to compute the factorizations carried out in each of the stages of the pipeline can be derived easily from Algorithm 2. In particular, the algorithm to perform the reduction in Stage 4, the largest one, results in a DAG like the one shown in Figure 10. As in the DAG of Figure 6, the steps carried out to obtain the trapezoid matrix W from $\hat{W}$ has not been represented. Clearly, the number of tasks is very low compared with the previous DAGs, showing a large reduction in computational cost. The number of tasks that can be executed concurrently have been reduced to only two, however, this loss of parallelism is compensated by the increase in the number of concurrent tasks that are needed to build the pipeline.

The pipeline has been implemented in C++17 with OpenMP tasks and *Single-Producer/Single-Consumer* queues connecting the stages. However, the structure can be implemented using other software tools like GRPPI (Generic Reusable Parallel Pattern Interface) [13]. GRPPI [14] is an open source generic and reusable parallel pattern programming interface that simplifies the developer efforts for parallel programming.

Table 2 shows a comparison in terms of execution time for three different problem sizes with its respective *tile* size. We have used a node with two Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70 processors with 12 cores each. The column labeled as *Rectangular* shows the execution time for the *QR* factorization with a tiled algorithm [9]. Column *Jagged* shows the execution time required to obtain the *QR* factorization of a matrix of the type shown in Figure 7 with the algorithm presented in [11]. The fifth column shows the time obtained with the pipeline proposed in this work, i.e. this is the time to reduce a matrix of the form shown in Stage 4 at Step 2 in Figure 9 to upper triangular, in other words, the time needed to obtain each factorization. It can be shown that all the times obtained are coherent with the computations carried out. As it can be seen in the last three columns of the table, the use of the pipeline allows to speed up the computation more than ×2 the time for *jagged* matrices, and close to ×3 the time for the *QR* factorization of rectangular matrices.

6 | CONCLUSIONS

This paper presents an idea to accelerate the update of the *QR* factorization of the matrix appeared in the Beamformer Algorithm. The idea consists of a pipeline in which some computations are advanced in time using data that has fed the pipeline in previous steps. The result, i.e. the upper triangular factor of the sought-after *QR* factorization, is produced by the last stage of the pipeline in the minimum possible time according to the data available to perform the update. With this parallel pattern, we managed to improve the speed 2 and 3 times, respectively, with regard to other two former algorithms.

This work has addressed the particular problem of a system matrix is made of 4×3 *tiles*. This shape arises when the system matrix changes 25% of its rows at each step. However, the proposal can be extended to a more general case in which the percentage of rows is different. Furthermore, as it what shown in previous works where the *QR* factorization tasks were accelerated with GPUs, the stages of the pipeline could be easily accelerated using these devices.

References

[1] Belloch J, Ferrer M, González A, Martínez-Zaldívar F, Vidal A. Headphone-Based Virtual Spatialization of Sound with a GPU Accelerator. *Journal of the Audio Engineering Society* 2013; 61: 546-561.

[2] Belloch JA, González A, Martínez-Zaldívar FJ, Vidal AM. Real-time massive convolution for audio applications on GPU. *The Journal of Supercomputing* 2011; 58(3): 449–457. doi: 10.1007/s11227-011-0610-8

[3] Ramiro C, Vidal A, González A. MIMOPack: A High Performance Computing Library for MIMO communication systems. *The Journal of Supercomputing* 2015; 71: 751-760.

[4] Alventosa F, Alonso P, Piñero G, Vidal A. Implementation of the Beamformer Algorithm for the NVIDIA Jetson. In: Springer; 2016; Granada, Spain: 201-211.

[5] Hammarling S, Lucas C. Updating the QR factorization and the least squares problem. tech. rep., The University of Manchester; 2008.

[6] Alventosa FJ, Alonso P, Piñero G, Vidal AM. QR update applied to Beamforming filter. In: . II(I). NESUS cost IC1305. UC3M; 2016; Timisoara, Romania: 33-36.

[7] Lorente J, Piñero G, Vidal A, Belloch J, González A. Parallel Implementations of Beamforming Design and Filtering for Microphone Array Applications. In: European Association for Signal Processing (EURASIP). ; 2011; Barcelona, Spain: 501-505.

[8] Gunter BC, Geijn v. dRA. Parallel Out-of-Core Computation and Updating the QR Factorization. *ACM Transactions on Mathematical Software* 2005; 31(1): 60–78.

[9] Quintana-Ortí G, Quintana-Ortí ES, Geijn RAVD, Zee FGV, Chan E. Programming Matrix Algorithms-by-blocks for Thread-level Parallelism. *ACM Trans. Math. Softw.* 2009; 36(3): 14:1–14:26. doi: 10.1145/1527286.1527288

[10] The OpenMP API specification for parallel programming. www.openmp.org; . Accessed on October 2018.

[11] Alventosa FJ, Alonso P, Vidal AM, Piñero G, Quintana-Ortí ES. Fast block QR update in digital signal processing. *The Journal of Supercomputing* 2018. doi: 10.1007/s11227-018-2298-5

[12] Golub G, Van Loan C. *Matrix Computations*. Johns Hopkins Studies in the Mathematical SciencesJohns Hopkins University Press . 2013.

[13] Dolz MF, Alventosa FJ, Alonso P, Vidal AM. A pipeline structure for the block QR update in digital signal processing. *The Journal of Supercomputing* To appear in 2019. doi: 10.1007/s11227-018-2666-1

[14] Rio Astorga dD, Dolz MF, Fernández J, García JD. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience* 2017; 29(24). doi: 10.1002/cpe.4175

□