Document downloaded from:

http://hdl.handle.net/10251/148184

This paper must be cited as:

Alventosa, FJ.; Alonso-Jordá, P.; Vidal Maciá, AM.; Piñero, G.; Quintana-Ortí, ES. (2019). Fast block QR update in digital signal processing. The Journal of Supercomputing. 75(3):1051-1064. https://doi.org/10.1007/s11227-018-2298-5

The final publication is available at

https://doi.org/10.1007/s11227-018-2298-5

Copyright Springer-Verlag

Additional Information

Journal of Supercomputing manuscript No.
(will be inserted by the editor)

Fast Block QR Update in Digital Signal Processing

Fran J. Alventosa · Pedro Alonso · Antonio M. Vidal · Gema Piñero · Enrique S. Quintana-Ortí

Received: date / Accepted: date

Abstract The processing of digital sound signals often requires the computation of the QR factorization of a rectangular system matrix. However, sometimes, only a given (and probably small) part of the system matrix varies from the current sample to the next one. We exploit this fact to reuse some computations carried out to process the former sample in order to save execution time in the processing of the current sample. These savings can be critical for real time applications running on low power consumption devices with high mobility. In addition, we propose a simple out-of-order task-parallel algorithm for the QR factorization using OpenMP that exploits the multicore capability of modern processors. Furthermore, in the presence of a Graphics Processing Unit (GPU) in the system, our algorithm is able to off-load some tasks to the GPU to accelerate the computation on these hardware devices.

Keywords QR factorization, QR Update, *jagged* matrix, real-time, block QR.

1 Introduction

The QR factorization of an $m \times n$ matrix A is given by $A = QR$, where Q is an $m \times m$ orthogonal matrix and R is an $m \times n$ upper triangular matrix. This factorization is often used to solve overdetermined linear systems characterized by having more equations than unknowns ($m > n$). The computational cost is

Fran J. Alventosa · Pedro Alonso · Antonio M. Vidal
Depto. de Sistemas Informáticos y Computación, Universitat Politècnica de València, Spain
Fran J. Alventosa E-mail: fraalrue@upv.es
Pedro Alonso E-mail: palonso@upv.es
Antonio M. Vidal E-mail: avidal@dsic.upv.es
Gema Piñero
Instituto de Telecomunicaciones y Aplicaciones Multimedia (iTEAM), Universitat Politècnica de València
E-mail: gpinyero@iteam.upv.es
Enrique S. Quintana-Ortí
Dept. Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain
E-mail: quintana@uji.es

$2n^2(m - n/3)$ flops (floating point operations per second) using Householder reflections [9]. Many algorithms have been proposed in the past to improve the computation of the QR factorization. Any new hardware architecture or configuration has resulted in a new method to compute efficiently the QR factorization, yielding large number of versions for a wide range of sequential and multicore processors, including also accelerators like Graphics Processing Units (GPUs). Many of these proposals correspond to block algorithms which improve performance on processors with a hierarchical organization of the memory system [10,5]. In this work we commence with a sequential block algorithm that performs this factorization [13], that we then modify to incorporate our ideas.

The computation of the QR factorization in this work aims to extract the most approximate solution of a linear least squares problem arising as the main computational kernel involved in the evaluation of a digital sound signal. The digital sound signal is sampled in real time, which implies that there exists a short time slot to perform all the computations associated to process a sample before the next one is available. Furthermore, the time constrain becomes more critical because we are interested in the use of devices with low computational capabilities (mobile phones, tablets, etc.). Hence, we propose to speed-up the computation of the QR factorization by reusing some computations performed during the factorization of the previous sample.

In addition, we are interested in leveraging the multicore capability of modern processors. Parallel algorithms for the solution of basic linear algebra kernels like the LU, Cholesky or QR factorizations on multicore processors are particularly productive, and there exist libraries that include efficient implementations of these kernels. Furthermore, there exist several runtimes that dynamically schedule subproblems for parallel execution once the problem has been previously partitioned into subtasks. These runtime-assisted parallelization approach targets advanced shared-memory parallelism by borrowing out-of-order scheduling techniques from sequential superscalar architectures [13]. One effort in this direction is SuperMatrix [6–8] whose runtime system enqueues the "tasks" conforming the linear algebra operation on a queue, builds a Directed Acyclic Graph (DAG) to encode dependencies, and executes tasks as the dependencies are satisfied. SuperMatrix contains implementations, e.g. of the LU factorization with incremental pivoting, and a closely related algorithm-by-blocks for the QR factorization. Both algorithms are based in turn on an originally design for out-of-core computation [11]. This is also the idea behind PLASMA project [5,4] which, similarly to SuperMatrix, provides an implementation of many BLAS and LAPACK routines. The OmpSs [1] programming model equips the programmer with the ability to express tasks dependencies and a runtime to run a DAG. These proposals were motivated by the fact that the regular algorithmic notation used to write algorithms in which tasks and operations are arranged in loops limits the capability to express the out-of-order execution nature of some tasks. The scenario, however, has changed considerably with OpenMP 4.0 since now it is easier to describe the dependencies among suboperations/tasks.

In order to achieve the level of performance required to fully exploit current processor architectures, the programmer has to meet several requirements at the same time, such as data locality, load balancing, etc. This challenge becomes more dramatic with the addition of an accelerator to the team of computational resources. Several strategies offer support for heterogeneous CPU-GPU systems in

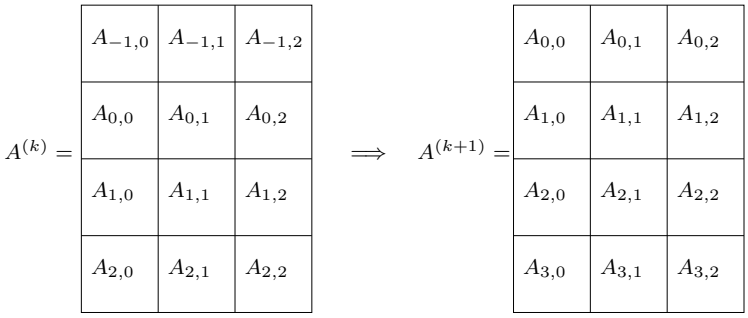

Fig. 1 Updating of system matrix A from iteration k to $k+1$.

order to map work to the CPU or the GPU. Some strategies are in the conceptual basis of the task programming library StarPU [3], or have been incorporated to the OmpSs [1] programming model. However, these tools must be previously installed in the system and require the use of a proprietary library or macro language. In this paper, we have tested that a simple strategy of scheduling tasks to the CPU or GPU subsystem based on task type is enough to exploit a SoC like the NVIDIA Jetson, a processor consisting of 4 ARM A57 cores and a GPU NVIDIA Pascal with 256 CUDA cores. In particular, the main contribution of this paper is a Hybrid-Tiled QR Factorization Algorithm (FQRFA) annotated with OpenMP 4.0 directives that computes a reduced QR factorization taking benefit of previously computed QR factorizations.

The rest of the paper is organized as follows. Section 2 explains the problem we tackle: the QR update in the context of Digital Signal Processing. In the following section we explain the implementation of a QR tiled algorithm and our proposal to improve performance on this factorization. In Section 4 we analyze the computational cost of the algorithm and the savings achieved. Section 5 elaborates on the parallelization of the algorithm in a hybrid architecture. The results are presented in Section 6 and we close the paper with some conclusions.

2 The QR Update Problem

Let $k = 0, 1, \ldots$ denote the iteration space for the sampling problem where the integer k represents the order of the sample at instant k. Processing a sample involves the QR factorization of the system matrix $A^{(k)}$. Assume this matrix is partitioned into square blocks, referred to as *tiles*, of size $t_s \times t_s$ (Fig. 1). Matrix $A^{(k)}$ is then "updated" with new data to form the system matrix of the following iteration $A^{(k+1)}$ as follows: 1) the "oldest" rows, which are the first t_s rows, are deleted; and 2) a set of t_s rows which come from the sampled signal[1] are appended to the bottom of the system matrix. The orthogonal factor Q is not explicitly formed since it is not needed.

At each iteration k we have to compute the QR factorization of the system matrix, i.e. $A^{(k)} = Q_k R_k$. In order to save operations in each factorization we propose to work on a different type of matrix that we denote as *jagged* (matrix J

[1] For the sake of simplicity we omit here the exact procedure to build the new rows of the system matrix from the sample.

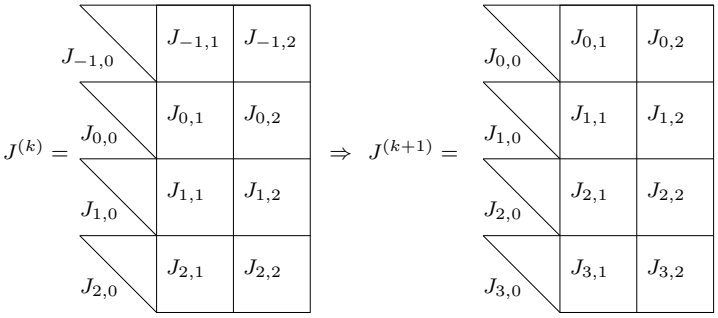

Fig. 2 Updating the *jagged* system matrix J from iteration k to $k+1$.

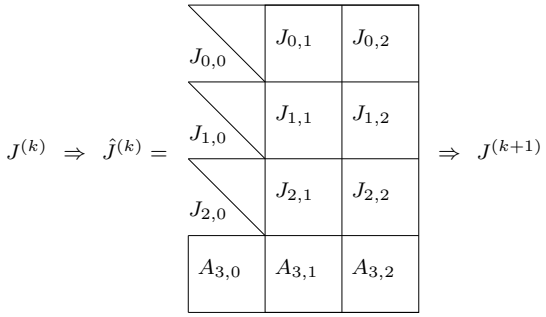

Fig. 3 Intermediate matrix (denoted as $\hat{J}^{(k)}$) between *jagged* matrices $J^{(k)}$ and $J^{(k+1)}$.

in Fig. 2). Each block row of $J^{(k)}$ is the triangular factor obtained from the QR factorization of the corresponding row in $A^{(k)}$.

In order to form $J^{(k+1)}$ from $J^{(k)}$ there exists an intermediate step that consists of performing the QR factorization of the new t_s rows, i.e. the last rows of $A^{(k+1)}$ and which are represented by $\begin{pmatrix} A_{3,0} & A_{3,1} & A_{3,2} \end{pmatrix}$, to obtain the last t_s rows of $J^{(k+1)}$, i.e. $\begin{pmatrix} J_{3,0} & J_{3,1} & J_{3,2} \end{pmatrix}$. Clearly, computing the QR factorization of the *jagged* matrix $J^{(k+1)}$ is cheaper than computing the QR factorization of the full matrix $A^{(k+1)}$.

3 The tiled QR factorization algorithm

The idea to obtain an algorithm-by-blocks for the QR factorization can be obtained following the out-of-core algorithm in [10]. Our approach is also based on the techniques introduced in [13].

As stated before, we consider the matrix as a collection of square *tiles* of size $t_s \times t_s$. Each *tile* is an array of elements stored in contiguous positions into memory. This implies that, if the matrix is stored in the usual column-major order required by BLAS/LAPACK routines, a previous transformation is needed to reorganize the matrix as a collection of *tiles*. However, this step is only necessary before processing the first sample. For now on in all our discussion and experimental results we use a matrix consisting of 4×3 *tiles* as that shown in Fig. 1. This is not a limitation since our proposal can be extended to any other grid arrangement of *tiles*. Also, the problem size is always a multiple of the *tile* size. This assumption

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

```
1  #pragma omp parallel
2  #pragma omp single private(i,j,k)
3  for( k = 0; k < N; k++ ) {
4    #pragma omp task depend( inout: A(k,k) )
5    D_QR( A(k,k) )
6    for( j = k+1; j < N; j++ ) {
7      #pragma omp task depend( in: A(k,k) ) depend( inout: A(k,j) )
8      D_QT( A(k,k), A(k,j) )
9    }
10   for( i = k+1; i < M; i++ ) {
11     #pragma omp task depend( inout: A(k,k), A(i,k) )
12     TD_QR( A(k,k), A(i,k) )
13     for( j = k+1; j < N; j++ ) {
14       #pragma omp task depend( in: A(i,k) ) depend( inout: A(k,j), A(i,j) )
15       TD_QT( A(i,k), A(k,j), A(i,j) )
16     }
17   }
18 }
```

is not a hard constraint since the underlying physical problem addressed in our proposal allows some degree of freedom in the choice of the *tile* size.

Algorithm 1[2] (`QRTiled(A)`) shows the *tile* algorithm for the QR factorization of an `M`×`N` *tiled* matrix of the form shown in Fig. 1. This algorithm uses four different types of tasks (those already identified and used in [13]). Their description is the following (we only specify the *tiles* of A involved in the computation as input parameters, omitting other factors):

`D_QR`: Computes the QR factorization of *tile* $A_{k,k}$. This operation (colored in red) produces a triangular factor on the pivot *tile* (subfigures a), i), and o)).

`D_QT`: Applies the factor Q^T obtained from the QR factorization of the diagonal block $A_{k,k}$ to *tile* $A_{k,j}$. (The Householder reflectors of Q^T are stored in the lower triangular part of $A_{k,k}$.) This operation (colored in cyan) can be found in subfigures b) and j), which comprise the execution of the outermost loop indexed by `j`.

`TD_QR`: Computes the QR factorization of matrix $\begin{pmatrix} A_{k,k} \\ A_{i,k} \end{pmatrix}$, where $A_{k,k}$ is upper triangular, so that *tile* $A_{i,k}$ is completely annihilated. This is represented by a dotted square *tile* in subfigures c), e), g), k), m), and p). Notice that, as *tile* $A_{k,k}$ is modified by this operation, its color becomes darker.

`TD_QT`: Applies the factor Q^T obtained by operation `TD_QR` within the same block row i. This operation modifies *tiles* $A_{k,j}$ and $A_{i,j}$, with the last one labeled with the name of the operation. The operation also reads *tile* $A_{i,k}$ because it keeps the Householder reflectors implicitly representing factor Q^T.

In order to factorize matrices of the *jagged* form (Fig. 2) we have added two types of tasks more:

`TD_QR_T`: This operation is equivalent to `TD_QR` when the matrix to reduce is $\begin{pmatrix} A_{k,k} \\ J_{i,k} \end{pmatrix}$ implying that the "lower" factor is triangular.

[2] In order to save space in the document we have annotated the algorithms with OpenMP tags that will be explained later. The sequential version arises from simply deleting these OpenMP directives.

Algorithm 2 `QRTiledJagged(J)`: performs the QR factorization of a matrix of the form $\hat{J}^{(k)}$ (Fig. 3).

```
1  #pragma omp parallel
2  #pragma omp single private(i,j,k)
3  {
4    #pragma omp task depend( inout: J(M-1,0) )
5    D_QR( J(M-1,0) );
6    for( j = 1; j < N; j++ ) {
7      #pragma omp task depend( in: J(M-1,0) ) depend( inout: J(M-1,j) )
8      D_QT( J(M-1,j), J(M-1) );
9    }
10   for( i = 1; i < M; i++ ) {
11     #pragma omp task depend( inout: J(0,0), J(i,0) )
12     D_QR_T( J(0,0), J(i,0) );
13     for( j = 1; j < N; j++ ) {
14       #pragma omp task depend( in: J(i,0) ) depend( inout: J(0,j), J(i,j) )
15       D_QT_T( J(i,0), J(0,j), J(i,j) );
16     }
17   }
18   QRTile(J(1:M,1:N)); /* Algorithm 1 */
19 }
```

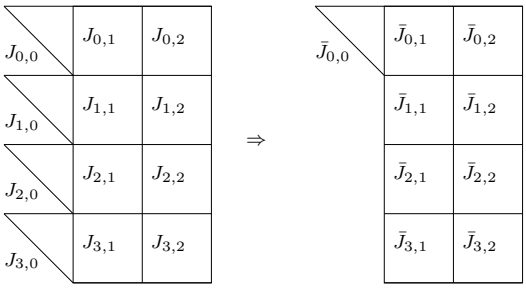

Fig. 4 Reduction of the first block column of matrix $J^{(k+1)}$ to upper triangular form.

`TD_QT_T`: As in the previous case, this operation is equivalent to `TD_QT` when the factor Q^T was generated with `TD_QR_T`.

The algorithm for the QR factorization of a *jagged* matrix has the form shown in Algorithm 2. This algorithm includes the QR factorization of the bottom t_s rows of matrix $\hat{J}^{(k)}$ (Fig. 3) in lines 4–9. Instructions in lines 10–17 perform the QR factorization of the first block column of matrix $J^{(k+1)}$. This operation is represented in Fig. 4. Finally, submatrix $J_{1:M,1:N}$ is reduced to triangular form by using routine `QRTiled` (Algorithm 1), as it is illustrated in Fig. 5.

4 Model for the tiled QR factorization algorithm

This section analyzes the cost in flops of the QR factorization of a *jagged* matrix, i.e. the cost of routine `QRTiledJagged` (Algorithm 2). An interesting point that shows the cost model is that the savings when operating with a *jagged* matrix compared with the cost of the original one are larger than the proportion of zeros in a *jagged* matrix with respect to the matrix size.

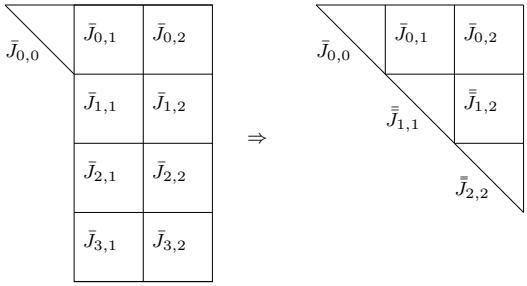

Fig. 5 Reduction of submatrix $\bar{J}_{1:M,1:N}$ to upper triangular form.

Consider Figure 2 and 3. The cost of computing $\begin{pmatrix} J_{3,0} & J_{3,1} & J_{3,2} \end{pmatrix}$ from the new rows $\begin{pmatrix} A_{3,0} & A_{3,1} & A_{3,2} \end{pmatrix}$, denoted as c_1, can be approximated as:

$$\begin{aligned} c_1 &= \sum_{i=1}^{t_s-1} (3(t_s - i + 1) + 4(t_s - i + 1)(n - i)) \\ &\approx 2nt_s^2 + \frac{3}{2}t_s^2 - \frac{2}{3}t_s^3 \approx 2t_s^2 \left(n - \frac{t_s}{3}\right) \text{ flops.} \end{aligned}$$

The reduction of a *jagged* matrix to upper triangular form is divided into two steps. The first one, which consists of the reduction of the first block column to upper triangular (Fig. 4), can be approximated as

$$\begin{aligned} c_2 &= \sum_{i=1}^{t_s} (3(i+1) + 4(i+1)(n-i)) \left(\frac{m}{t_s} - 1\right) \\ &\approx \left(2nt_s^2 + \frac{3}{2}t_s^2 - \frac{4}{3}t_s^3\right) \left(\frac{m}{t_s} - 1\right) \approx \left(2t_s^2 \left(n - \frac{2}{3}t_s\right)\right) \left(\frac{m}{t_s} - 1\right) \text{ flops.} \end{aligned}$$

The cost of the second step, plotted in Fig. 5, has the following form

$$\begin{aligned} c_3 &= 2(n - t_s)^2 \left((m - t_s) - \frac{n - t_s}{3}\right) \\ &= 2n^2 \left(m - \frac{n}{3}\right) - \left(4t_s mn + \frac{4}{3}t_s^3 - 2t_s^2 m - 2t_s^2 n\right) \text{ flops.} \end{aligned}$$

Merging the three costs and operating with the result we obtain an approximation T_J of the theoretical cost to execute routine `QRTiledJagged` (Algorithm 2) on matrix $\hat{J}^{(k)}$ in order to obtain the QR factorization of matrix $J^{(k+1)}$:

$$\begin{aligned} T_J &= c_1 + c_2 + c_3 = \left(2nt_s^2 - \frac{2}{3}t_s^3\right) + \left(2nt_s^2 - \frac{4}{3}t_s^3\right) \left(\frac{m}{t_s} - 1\right) \\ &\quad + 2n^2 \left(m - \frac{n}{3}\right) - \left(4t_s mn + \frac{4}{3}t_s^3 - 2t_s^2 m - 2t_s^2 n\right) \\ &\leq \left(2nt_s^2 - \frac{2}{3}t_s^3\right) \left(\frac{m}{t_s}\right) + 2n^2 \left(m - \frac{n}{3}\right) - \left(4t_s mn + \frac{4}{3}t_s^3 - 2t_s^2 m - 2t_s^2 n\right) \\ &= 2n^2 \left(m - \frac{n}{3}\right) - \left(\frac{4}{3}t_s^3 - \left(\frac{4}{3}m + 2n\right) t_s^2 + 2mnt_s\right) \\ &= T_A - T_S \ , \end{aligned}$$

Table 1 Theoretical speed-up obtained when working on *jagged* matrices compared with the "naive" algorithm.

$m \times n$ / t_s	1	2	8	32	80	160	320	640
1280×960	1.00	1.00	1.01	1.04	1.11	1.21	1.35	1.33
2560×1960	1.00	1.00	1.01	1.02	1.05	1.11	1.21	1.35

where T_A denotes the cost (in flops) of performing the QR factorization of an $m \times n$ matrix A and T_S denotes the savings.

The speed-up (S) of the QR factorization of matrix $\hat{J}^{(k)}$ (Fig. 3) compared to the QR factorization of matrix $A^{(k+1)}$ (Fig. 1) can be approximated as

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

As the speed-up S does not vary linearly with t_s we need to use numerical examples to see the impact of working on *jagged* matrices instead of the original ones. Table 1 shows numerical examples for several values of the block size t_s and two matrix sizes.

5 The Hybrid-Tiled QR Factorization Algorithm

A multithreaded version of the tiled QR factorization algorithm based of OpenMP is quite straightforward here for the two loops indexed by `j`. There exists, however, a more efficient approach that unleash an out-of-order execution of the operations that can be run concurrently, regardless of whether they belong to the same loop or not. To this end the algorithm can be formulated as a collection of operations on tiles that are related by a set of dependencies. In our formulation we use OpenMP 4.0. A task-oriented design for a Parallel Tiled QR Factorization Algorithm can be written by annotating the sequential version with the OpenMP directive `task` (Algorithm 1). Furthermore, the clause `depend` can be leveraged to specify the dependencies among tasks. The OpenMP runtime schedules the operations according to a Direct Acyclic Graph (DAG) that represents the dependency relationships among operations on tiles. This DAG is illustrated in Fig. 6, where we distingish each type of task with a diferent color. The same strategy is used to implement the multithreaded version of the Tiled QR Factorization Algorithm that works on matrices of the form $\hat{J}^{(k)}$ (Fig. 3) as shown in Algorithm 2. The DAG of routine `QRTiledJagged` is very similar to the one shown in Fig. 6.

In addition, we have enhanced our parallel algorithms with the ability to use a NVIDIA GPU in the system yielding the Hybrid-Tiled QR Factorization Algorithm. There exist different options at the time of scheduling a task either to the CPU or the GPU. We have chosen a natural choice that makes the decision based on the task consisting in that `D_QR` and `TD_QR` are always executed in CPU, while `D_QT` and `TD_QT` are executed in GPU. The operations carried by the GPU are differentiated from those carried out by the CPU in the DAG of Fig. 6 because the node shape is a doubled circle. Analogously, in the case of Algorithm 2, any

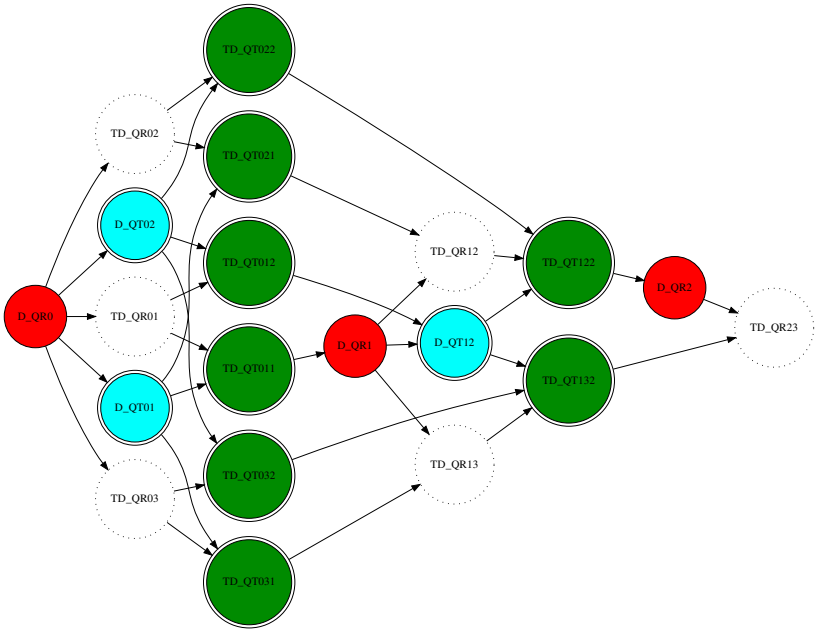

Fig. 6 Direct Acyclic Graph (DAG) for the execution of the Hybrid-Tiled QR Factorization Algorithm (Algorithm 1) on a 4×3 *tiles* matrix $A^{(k+1)}$ (Fig. 1).

task of type `TD_QR_T` is executed by the CPU while tasks of type `TD_QT_T` are all uploaded to the GPU.

On the CPU side, we have used OpenBLAS [2] for the essential BLAS/LAPACK kernels invoked from each task. The implementation of the tasks executed by the GPU is subdivided into smaller BLAS routines so they can be easily performed via CUBLAS [12] routines. When the algorithm encounters a task of type `D_QT` or `TD_QT`, the *tiles* involved in the computation are off-loaded to the GPU, the GPU performs the computation in GPU and, finally, the results are returned back to the CPU. This is simple to implement but comes at the cost of increasing the amount of data CPU-GPU transferences. However, if the GPU is fed with many tasks, communications and computations can be overlapped hiding thus the communication cost. This implementation can obtain some benefits from Hyper-Q on those devices that support this technology. This feature was integrated in the Kepler architecture to exploit the throughput of a device that receives operations from multiple threads concurrently. These requests, that are enqueued in the same default stream, are not yet "falsely" serialized [14].

6 Experimental Results

All the experiments have been realized in a NVIDIA Jetson TX2 development kit. This platform features a Quad ARM® A57 and a NVIDIA Pascal™ with 256 CUDA cores. We compile the CPU code with the GNU compiler `gcc 6.2`, a version which is compliant with OpenMP 4.5. We note here that a version $\geq$ 4.9

Table 2 Time in seconds of QR factorization of matrix A (Fig. 1) vs. matrix $\hat{J}$ (Fig. 3) for problem sizes 1280×960 and 2560×1920 varying the tile size and the block size.

		$m \times n = 1280 \times 960$				$m \times n = 2560 \times 1920$		
b_s	t_s	QR($A^{(k+1)}$)	QR($\hat{J}^{(k+1)}$)	S	t_s	QR($A^{(k+1)}$)	QR($\hat{J}^{(k+1)}$)	S
20	160	0.094 s.	0.102 s.	1.43	160	0.691 s.	0.695 s.	1.41
32		0.100 s.	0.113 s.	1.32		0.714 s.	0.734 s.	1.35
40		0.108 s.	0.124 s.	1.21		0.740 s.	0.772 s.	1.30
80		0.142 s.	0.176 s.	0.88		0.887 s.	0.992 s.	1.04
20	320	0.089 s.	0.097 s.	1.51	320	0.692 s.	0.698 s.	1.41
32		0.095 s.	0.108 s.	1.38		0.693 s.	0.708 s.	1.40
40		0.102 s.	0.119 s.	1.27		0.710 s.	0.741 s.	1.35
80		0.131 s.	0.166 s.	0.95		0.811 s.	0.920 s.	1.13

is needed in order to use the `depend` clause introduced in OpenMP 4.0. For the GPU code we used CUDA 8.0 and the CUBLAS library provided for BLAS.

The results for our first experiment were obtained in a single ARM Cortex-A57 core of our target machine. The purpose of experiment is to demonstrate the benefits of using a *jagged* matrix compared with the naive regular rectangular matrix. Table 2 shows the time in seconds to perform the QR factorization of each matrix $A^{(k)}$ (column QR($A^{(k)}$)), and the factorization of the *jagged* matrix $J^{(k)}$ (column QR($J^{(k)}$)). The speed-up S achieved is also compared. We show the results on matrices of two problem sizes and two different *tile* sizes (t_s) for each problem size. The implementation of the tasks is parametrized by an algorithmic block size (b_s) so that different block sizes result in different execution times. Both the block size b_s and the *tile* size t_s affect the performance of the algorithm. There exist some freedom to select the block size b_s provided it is always integer divisor of t_s. In the exposition so far and for clarity, the *tile* size has been assumed to be equal to the number of rows to be updated, but in the actual implementation the *tile* size can be an integer divisor of the number of rows to be updated. Table 2 shows that the best performance of the QR factorization of the two matrices, i.e. QR($A^{(k)}$) and QR($J^{(k)}$), is obtained always for the same combination of values (t_s, b_s). In these cases, the speed-up of QR($J^{(k)}$) with respect to QR($A^{(k)}$) is in the range $\approx [1.41, 1.45]$. These numbers are slightly higher than those obtained by our theoretical model in Table 1.

We have evaluated matrices with $\mathtt{M} = 4$ and $\mathtt{N} = 3$ *tiles*. The two rectangular problems thus differ in the *tile* size, which is $t_s \in \{320, 640\}$, yielding matrix sizes $m \times n = 1280 \times 960$ and $m \times n = 2560 \times 1920$.

The plots in Fig. 7 report the execution time for the two problem sizes tackled (1280×960 and 2560×1920, respectively). The purple line shows the execution time obtained with the algorithm in Fig. 2, which uses the four ARM cores. The green line shows the hybrid version, i.e. which adds the GPU to the computation. The two plots are both compared with the OpenBLAS implementation of the QR factorization that also uses the four ARM cores. (Note that the *tile* size does not affect to the OpenBLAS version.) The block size is the best determined in the previous experiment.

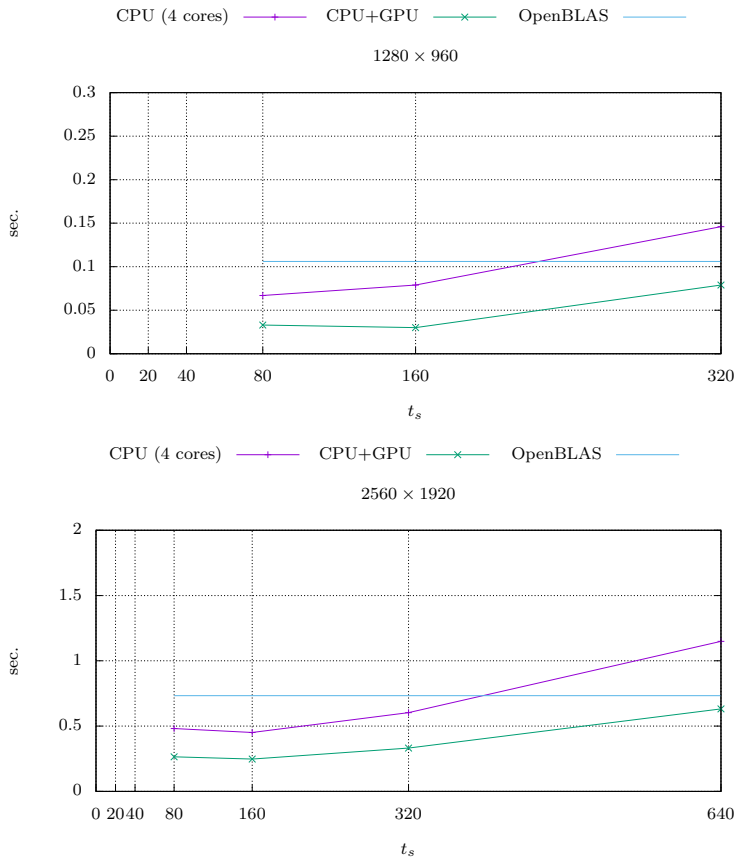

Fig. 7 Execution time of the Hybrid-Tiled QR Factorization Algorithm.

7 Conclusions

Real time applications are characterized by performing the same computation repeatedly over new coming data. Sometimes, data, which frequently is represented as a system matrix, changes from one iteration to the next one only in a small part. If the computation on this matrix is expensive, as it is the QR factorization, we can use this fact to save processing time. We propose to work on a modified matrix, called *jagged*, instead of on the original system matrix. With this simple idea, it is possible to increase performance by a factor close to 1.45 in the particular case we tackled in this paper, i.e. when data is represented by a 4×3 *tiles* matrix and 50% of data (25% discarded rows and 25% new rows) changes from one iteration to the next one.

The parallel algorithm proposed partitions the computation in a set of tasks with a dependency relationship. This dependency is easily expressed without having to modify the sequential algorithm, just annotating the code with OpenMP directives introduced in standards 3.0 and 4.0. We have exploited all the capabilities of our target machine which consists of 4 CPU cores and 1 GPU by driving a

task either to a CPU core or to the GPU depending on its type. The results show that our solution can be used for applications with real-time if the problem size is not very large.

Acknowledgments

This work was supported by the Spanish Ministry of Economy and Competitiveness under MINECO and FEDER projects TEC2015-67387-C4-1-R and TIN2014-53495-R; and the Generalitat Valenciana PROMETEOII/2014/003.

References

1. The OmpSs Programming Model. https://pm.bsc.es/ompss. Accessed on May 2017.
2. Openblas. http://www.openblas.net. Accessed on May 2017.
3. Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Research Report RR-7240, INRIA, March 2010.
4. Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.
5. Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38 – 53, 2009.
6. Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 116–125, New York, NY, USA, 2007. ACM.
7. Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert De Van Geijn. Satisfying your dependencies with supermatrix. In *Proceedings - 2007 IEEE International Conference on Cluster Computing, CLUSTER 2007*, pages 91–99, 2007.
8. Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Supermatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In Siddhartha Chatterjee and Michael L. Scott, editors, *PPOPP*, pages 123–132. ACM, 2008.
9. G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.
10. Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, March 2005.
11. Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. In *Applied Parallel Computing, State of the Art in Scientific Computing, 7th International Workshop, PARA 2004, Lyngby, Denmark, June 20-23, 2004, Revised Selected Papers*, pages 413–422, 2004.
12. NVIDIA. The cuBLAS library. http://docs.nvidia.com/cuda/cublas.
13. Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, July 2009.
14. Florian Wende, Thomas Steinke, and Frank Cordes. Multi-threaded kernel offloading to gpgpu using hyper-q on kepler architecture. Technical Report 14-19, ZIB, Takustr.7, 14195 Berlin, 2014.