Document downloaded from:

http://hdl.handle.net/10251/123860

This paper must be cited as:

Dolz, MF.; Alventosa, FJ.; Alonso-Jordá, P.; Vidal Maciá, AM. (2019). A pipeline structure for the block QR update in digital signal processing. The Journal of Supercomputing. 75(3):1470-1482. https://doi.org/10.1007/s11227-018-2666-1

The final publication is available at

https://doi.org/10.1007/s11227-018-2666-1

Copyright Springer-Verlag

Additional Information

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

A pipeline structure for the block QR update in digital signal processing

Manuel F. Dolz · Fran J. Alventosa · Pedro Alonso-Jordá · Antonio M. Vidal

Received: date / Accepted: date

Abstract There exist problems in the field of digital signal processing, such as filtering of acoustic signals, that require processing a large amount of data in real-time. The Beamforming algorithm, for instance, is a process that can be modeled by a rectangular matrix built on the input signals of an acoustic system and, thus, changes in real-time. To obtain the output signals, it is required to compute its QR factorization. In this paper, we propose to organize the concurrent computational resources of a given multicore computer in a pipeline structure to perform this factorization as fast as possible. The pipeline has been implemented using both the application programming interface OpenMP, and GrPPI, a library interface to design parallel applications based on parallel patterns. We tackle, not only the performance challenge but also the programmability of our idea using parallel programming frameworks.

Keywords QR factorization, QR Update, pipeline QR update, *jagged* matrix, GrPPI, Beamforming Algorithm.

1 Introduction

It is quite common that for digital signal processing applications, given an input signal, the resulting output signal is required in real-time. One of these applications, for instance, consists of recovering an acoustic signal that has been altered by noise, reverberations or other possible alterations. This system, which takes place into a room, is represented by several microphones in one side of this room that are recording a mix of signals being emitted by several independent loudspeakers placed at the other side of the room. This system is modeled as a Multiple Input Multiple Output (MIMO) system [1, 2].

Manuel F. Dolz
Departament d'Enginyeria i Ciència dels Computadors, Universitat Jaume I de Castelló, Spain
E-mail: dolzm@icc.uji.es

Fran J. Alventosa, Pedro Alonso-Jordá, Antonio M. Vidal
Depto. de Sistemas Informáticos y Computación, Universitat Politècnica de València, Spain
E-mail: {fraalrue,palonso,amvidal}@upv.es

In this work, we deal with the solution to this problem in one of its forms, that in which the digital system is represented by a system matrix that relates inputs and outputs. The system matrix must be factorized (QR factorization) to solve this problem, being this part the most time-consuming kernel of the whole application. The system matrix is partitioned in blocks of rows and columns whose number and size all depend on the number of inputs, outputs, and other factors. In [3,4], we initiated this work by proposing the use of a special matrix called *jagged*. The QR factorization of this special matrix can be quickly updated real-time in parallel using OpenMP tasks with dependencies. The number and size of the OpenMP tasks are closely related to the number and size of matrix blocks. This fact may limit the scalability of the parallel factorization since the size and the number of tasks can not be freely adapted to the processor features, i.e. the performance of the parallel algorithm depends on the *shape* imposed by the physical conditions of the MIMO system. In this contribution, we arrange the available parallel computational resources to build a *pipeline* structure or pattern. With this structure, we manage not only to reduce the execution time to get this QR factorization but also to improve the throughput of the application, i.e. to increase the number of QR factorizations calculated per time unit.

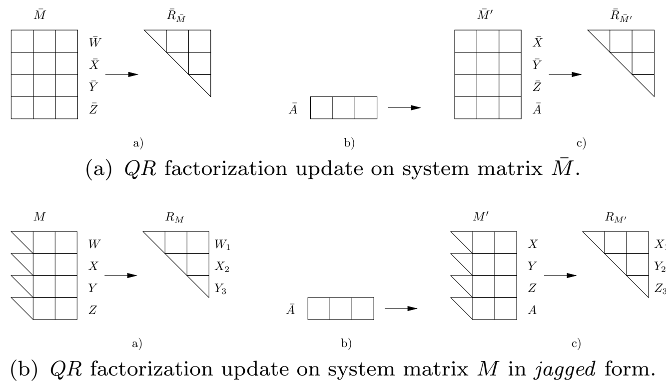
For the implementation of the pipeline, we have used two different tools. The first one is OpenMP, a very known programming interface to design parallel applications based on pre-processor macros. The stages of the pipeline are OpenMP `tasks` interconnected through queues. The second tool is GrPPI (Generic Reusable Parallel Pattern Interface) [5]. GrPPI is an open source generic and reusable parallel pattern programming interface that simplifies the developer efforts for parallel programming. The interface leverages modern C++ features, meta-programming concepts, and generic programming to achieve this goal.

The next section presents the digital system and outlines the kernel of the Beamforming Algorithm. The main idea of this contribution is described in Section 3. Section 4 deeps into the implementation details of the pipeline while Section 5 is devoted to the results obtained with our pipeline compared with previous efforts. The paper closes with a conclusions section.

2 The Beamforming Algorithm and the sound digital system

One approach to solve the Beamformer problem for sound signals is based on a correlation matrix that represents the relationship between input and output signals [6]. Provided all the room channel responses are known, the known as LCMV (Linearly Constrained Minimum Variance) Beamformer Algorithm [7] can be used to design the broadband "beamformers" (digital filters) that allows obtaining the desired clean output signal.

Let $\bar{M} \in \mathbb{R}^{m \times n}$ be the correlation matrix of the system over time [7], then this matrix changes periodically by losing a bunch of t_s top rows while an equal bunch of rows (built with new data) is appended to the bottom. Each time the system matrix is updated, the algorithm performs a QR factorization of $\bar{M}$ from which only the upper triangular factor $\bar{R}_{\bar{M}}$ is required in order to solve the least squares problem (Fig. 1(a), a)). This factorization is by far the most time-consuming part of the algorithm [3]. Performing this factorization is critical if the result is required



(a) QR factorization update on system matrix $\bar{M}$.

(b) QR factorization update on system matrix M in *jagged* form.

Fig. 1 QR factorization of system matrices.

in real-time, as it is the case with other signal processing applications such as 3D audio [8,9], which also use this correlation matrix between inputs and outputs.

The digital system under test imposes a special structure to the correlation matrix $\bar{M}$ (Fig. 1(a)) so that the proportion between the number of rows and columns is 4×3. The matrix is partitioned in square *tiles* of order t_s, so that $m = 4t_s$ and $n = 3t_s$. At each step of the process, matrix $\bar{M}$ changes by losing 25% of the rows at the top (row of *tiles* $\bar{W}$). Given a set of new rows represented by a 1×3 *tiles* matrix $\bar{A}$ (Fig. 1(a), b)), this block is appended at the bottom to form matrix $\bar{M}' \cdot$. Then, a new upper triangular factor of the QR factorization of the new "updated" system matrix $\bar{M}'$, i.e. $\bar{R}_{\bar{M}'}$, must be calculated. The objective is to recompute (or update) this factor as fast as possible at each processing time.

The computational cost of performing a QR factorization is $2n^2(m-n/3)$ flops (floating point operations per second) using Householder reflections [10]. There exist proposals to improve this computational kernel on different hardware architectures. Many of them are based on block algorithms which improve performance on processors with a hierarchical organization of the memory system [11,12]. In [13], for instance, we built a parallel version of a sequential block algorithm proposed in [14] that performs this factorization with the aim at exploit the multicore capacity available in current personal computers. This version builds a DAG (Directed Acyclic Graph) to encode dependencies and executes tasks as the dependencies are satisfied. OpenMP `task` together the `depend` clause are enough tools to articulate this DAG. The OpenMP runtime schedules the operations according to that DAG, which represents the dependency relationships among operations on *tiles*. Using *jagged* matrices instead of the original rectangular ones allows to speed up computations by a factor of $\times 1.35$ approximately [4].

3 The pipeline structure to update the QR factorization

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 2 shows the

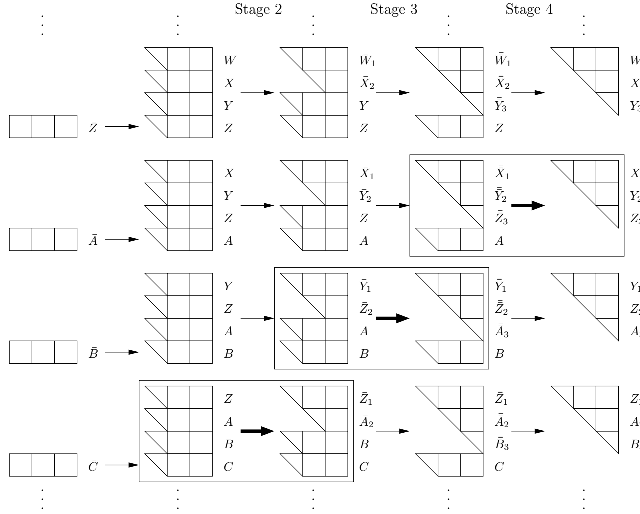


Fig. 2 Reduction to upper triangular form through QR factorization of the system matrix M, in *jagged* form, in different steps of the algorithm.

process followed when a new bunch of t_s rows is coming (first column of matrices). 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 M in Fig. 1(b), a). Stage 2 consists of reducing the submatrix formed by blocks W and X to the one formed by blocks $\bar{W}_1$ and $\bar{X}_2$. At the next stage, the submatrix formed by blocks $\bar{W}_1$, $\bar{X}_2$, and Y is reduced to the one formed by blocks $\bar{\bar{W}}_1$, $\bar{\bar{X}}_2$, and $\bar{\bar{Y}}_3$. Analogously, the last stage produces the sought-after upper triangular factor $R_M = \left(W_1^T \; X_2^T \; Y_3^T \right)^T$ of Fig. 1(b), a). In the same way, the second row represents the new system matrix M' (Fig. 1(b), c)) resulted by discarding X from M and appending A, which is the result of reducing the new bunch of rows $\bar{A}$ (Fig. 1(b), b)) to a trapezoid by its QR factorization. Hence, at the last stage we get $R_{M'}$ (Fig. 1(b), c)). The rest of the rows represent the following system matrices in *jagged* form, i.e. matrices formed when the first block is discarded and a new one is appended.

The objective of our proposal is to obtain the QR factorization of matrix M' (Fig. 1(b)), i.e. $R_{M'}$, as fast as possible. This can be accomplished if the upper triangular factor $\left(\bar{\bar{X}}_1{}^T \; \bar{\bar{Y}}_2{}^T \; \bar{\bar{Z}}_3{}^T \right)^T$ of the QR factorization of the upper square submatrix of M', i.e. $\left(X^T \; Y^T \; Z^T \right)^T$, has already been computed when matrix A is available. A key factor that can be observed in Fig. 2 is that those framed stages operate on the same submatrix A and the three of them can be executed concurrently. We have used this fact to design a pipeline structure like the one shown in Fig. 3. The figure shows four steps of execution at each horizontal line. It can be observed that, at Step 1, the output of the pipeline is matrix R_M (Fig. 1(b)). Assuming that the matrices represented into the stages of the pipeline at Step 1 have been computed previously, we can see that Step 2 represents the operation in which Stage 1 broadcasts factor A to the following three stages. The computations are carried out at the third step of the pipeline, where those matrices built at

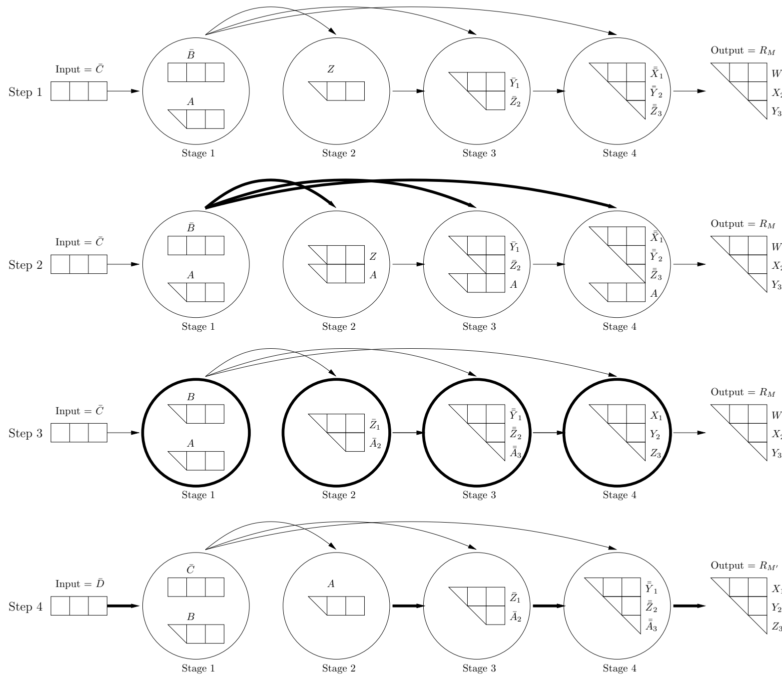


Fig. 3 Pipeline for the QR factorization update.

Step 2 are reduced through a QR factorization to the triangular (or trapezoidal) matrix represented inside the bold-line circles. Indeed, these computational steps are those represented in the framed stages of Fig. 2. 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. $R_{M'}$ (Fig. 1(b)). Notice that Step 1 and Step 4 are really the same but with different data.

The computational cost of performing Stage 4 of the pipeline (Fig. 3) can be approximated by considering that 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 (see [10] for details). Provided the triangularizations are carried out by means of Householder Reflections the complete cost consists of two terms. 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 the remaining rectangular $t_s \times (n - t_s)$ bottom block. Thus, the total number of flops is

$$\sum_{i=1}^{t_s} 6(i+1) + 4(n-i)(i+1) + \sum_{i=1}^{m} 6(t_s+1) + 4(t_s+1)(m-i) \; , \qquad (1)$$

and, after some arithmetic operations, this expression 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 (1) the cost of triangulating factor $\bar{A}$, which is $2t_s^2 n$ flops (this cost can be deduced in the same way as (1)). 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 the rectangular matrix $\bar{M}$

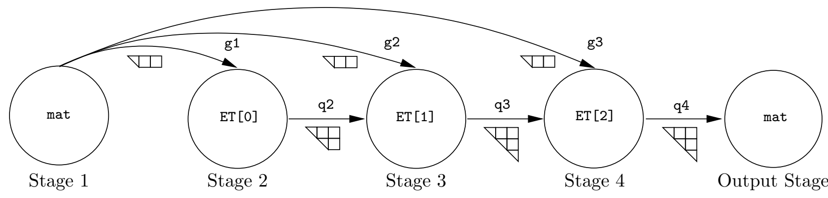


Fig. 4 OpenMP pipeline implementation.

to $\bar{R}_{\bar{M}}$ (Fig. 1(a)), $40t_s^3$ flops to reduce the *jagged* matrix M to R_M (Fig. 1(b)), and $20t_s^3$ flops to reduce a matrix of the form shown in Stage 4 of Fig. 2 to upper triangular form, including the computation of A as well.

4 Implementation of the pipeline

In this section, we describe the implementations of the pipeline described in the previous section. In particular, we propose *i)* an OpenMP-based implementation; and *ii)* a pattern-based implementation using GrPPI, a novel C++ high-level interface of parallel patterns. Further, we improved these implementations using nested parallelism at pipeline stage.

4.1 The OpenMP implementation

The OpenMP implementation leverages *tasks* to execute the pipeline stages and *Single-Producer/Single-Consumer* (SPSC) queues to communicate matrices among them. Fig. 4 depicts the main pipeline schema, while Listing 5(a) shows a simplified version of the OpenMP implementation. It is worth noting that the operations related to each of the stages are executed in a loop fashion so they emulate the stream nature of the pipeline pattern. Particularly, an iteration of Stage 1 reads the input set of t_s rows (e.g. $\bar{A}$ of Step 2 in Fig. 3), computes its QR factorization (A) and broadcasts it to Stages 2–4 (pushing A into queues `g1`, `g2` and `g3`). Alternatively, an iteration of Stages 2–4 pops A and appends it at the end of the stage *state matrix* (`ET`). Finally, Stages 2–4 reduce the state matrix via the QR factorization, push the resulting upper triangular factor to the corresponding output queue (`q2`, `q3` and `q4` for Stage 2, 3 and 4, respectively) and pop/copy the new incoming upper triangular/trapezoid factor from the input queue `q2`/`q3` for Stage 3/Stage 4 to the stage state matrix. This process is repeated once per input matrix and stage. We denote this pipeline version as Pipe-OMP-Seq.

While the pipeline stages of Pipe-OMP-Seq execute the tiled QR algorithm in series, we have developed an improved version that executes the same algorithm but in parallel. Specifically, this implementation builds a DAG using the OpenMP `task` directive and the `depend` clause, which encapsulates the dependencies of each QR factorization performed at the pipeline stages. Given the nested parallel nature of this implementation, the OpenMP tasks executing the pipeline stages require the directive `taskgroup` wrapping the QR *subtasks*. By doing so, the stage task waits until the completion of all QR-related subtasks before processing the next

```
1  #pragma omp parallel num_threads(5)
2  {
3   #pragma omp single nowait
4   { // Stage 1
5    #pragma omp task shared(gen,g1,g2,g3)
6    { for (;;) {
7        auto mat = gen();
8        g1.push(mat);
9        ...
10       if (!mat) break;
11     }
12   }
13   // Stage 2
14   #pragma omp task shared(ET,qr,g1,q2)
15   { for (;;) {
16       auto mat = g1.pop();
17       if (!mat) break;
18       ET[0].copy_rows(*mat, 1, 0, 1);
19       q2.push( qr(ET[0], 0) );
20       ET[0].copy_rows(*mat, 0, 0, 1);
21     }
22   }
23   // Stage 3
24   #pragma omp task shared(ET,qr,g2,q3)
25   { ... }
26   // Stage 4
27   #pragma omp task shared(ET,qr,g3,q4)
28   { ... }
29   // Output stage
30   #pragma omp task shared(q4,cons)
31   { ... }
32   #pragma omp taskwait
33  }
34 }
```

(a) OpenMP implementation.

```
int state[5] = {0,0,0,0,0};
parallel_execution_omp exec_omp{},
grppi::pipeline(
 exec_omp, // Execution mode
 // Stage 1
 [&]() -> optional<Matrix> {
  if (state[0]++ < max_it * 2) {
     return gen(state[0]);
  else
     return {};
 },
 // Stage 2
 [&] (auto mat) {
  switch (state[1]++ % 2) {
   case 0:
    ET[0].copy_rows(mat, 1, 0, 1);
    return mat;
   case 1:
    auto mat_aux= ET[0];
    auto mat_qr= qr(ET[0], 0);
    ET[0].copy_rows(mat_aux, 0, 1, 1);
    return mat_qr;
  }
 },
 // Stage 3
 [&] (auto mat)
 { ... },
 // Stage 4
 [&] (auto mat)
 { ... },
 // Output stage
 [&] (auto mat)
 { ... }
}
```

(b) GrPPI implementation.

Fig. 5 OpenMP and GrPPI implementations of the pipeline in Beamformer algorithm.

iteration with a new incoming matrix. We denote this pipeline implementation as Pipe-OMP-Par.

4.2 The pattern-based implementation

The pattern-based implementation of the pipeline makes use of the GrPPI interface, a parallel pattern interface for C++ applications [15]. Specifically, GrPPI takes full advantage of modern C++ features, metaprogramming concepts, and generic programming to act as a unified interface between the OpenMP, C++ threads and Intel TBB parallel programming models, hiding away the complexity behind the use of native concurrency mechanisms. To propose an alternative version of the algorithm presented in this paper, we take advantage of the GrPPI pipeline pattern.

Listing 1 shows the GrPPI pipeline interface. As it can be seen, the interface receives the execution policy (`exec_policy`), which can be any of the supported programming interfaces (OpenMP, ISO C++ threads, or Intel TBB), and the functions (`stages`) related to the pipeline stages by universal reference (`&&`). The C++ interface uses templates, making it more flexible and reusable for any data type. Note also the use of C++11 variadic templates (`...`), allowing a pipeline to have an arbitrary number of stages by receiving a collection of functions passed as arguments. Depending on the chosen execution policy the pipeline performs different actions. For the sake of simplicity, however, we limit the execution of the

Listing 1 GrPPI pipeline interface.

```
template <typename E, typename ... S>
void pipeline(E exec_policy, S && ... stages);
```

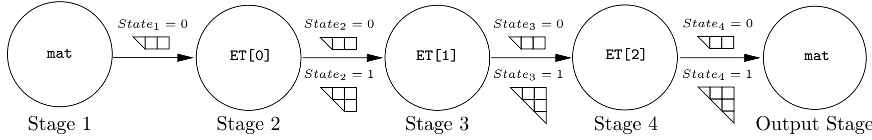


Fig. 6 GrPPI pipeline implementation.

GrPPI pipeline to only OpenMP, where individual tasks are used to articulate the pipeline stages.

For the development of this version, several modifications on the original pipeline were adopted. Given that the DAG presented for Pipe-OMP-Seq in Fig. 4 does not exactly match the pipeline pattern, the original DAG was flattened into a GrPPI pipeline. This flattening process led to bistate stages to mimic the behavior of the OpenMP pipeline. The bistate stages of the pipeline proceed as depicted in Fig 6 and in Listing 5(b). During the state 0, the set of t_s rows (A) is propagated along all pipeline stages. That is, Stage 1 reads the input set of t_s rows, computes its QR factorization and emits the upper triangular factor to Stage 2; simultaneously, Stage 2 receives, copies (into its private matrix), and forwards the upper triangular factor to Stage 3; the same occurs for Stages 3 and 4. During state 1 the rest of operations are performed in Stages 2–4. These stages perform the QR computation from its internal matrix state, copy the upper triangular/-trapezoid factor received by argument from the previous stage, and return the resulting factor to the next stage. For further analyses, we denote this implementation as Pipe-GrPPI-Seq.

Similar to the OpenMP implementation, which computes the QR tiled algorithm at stage level using a task-parallel approach, we developed the analogous for the GrPPI version. This design exploits parallelism using natively OpenMP and the same mechanism presented in the previous subsection. We refer to this implementation as Pipe-GrPPI-Par.

All in all, though this version required the transformation of the single-state stages into bistate, the high-level of patterns interface provided by GrPPI simplified the design of the application. For instance, different from the OpenMP implementation, GrPPI interface hides away the communication queues used to transfer items (matrices) between stages, and the main loops used in the pipeline stages. Also, its compact design is capable of reducing the number of lines of code, making it more robust due to the encapsulation of concurrency mechanisms.

5 Experimental results

In this section, we perform an experimental evaluation of the QR factorization algorithm implemented utilizing the pipeline design, by using both the OpenMP

and the GrPPI programming models. To carry out this evaluation, we used the following hardware and software components:

- *Target platform*: The evaluation has been carried out on a server platform equipped with 2× Intel Xeon Ivy Bridge E5-2695 v2 with a total of 24 cores running at 2.70 GHz, 30 MB of L3 cache and 128 GB of DDR3 RAM. The OS is a Linux Ubuntu 16.04.4 LTS with the kernel 4.4.0-109.
- *Software*: To evaluate the performance of the proposed OpenMP pipeline implementations, we used the `icc` compiler v17.0.4 (Intel Parallel Studio XE 2018). Differently, for the pattern-based implementation, we used GrPPI v0.4s along with the `gcc` GNU compiler v6.3.0, which supports C++17. In both versions, the `-O3` was specified. The BLAS and LAPACK (sequential) kernels used to compute the task-parallel tiled QR algorithm were those provided by Math Kernel Library in the Intel Parallel Studio XE 2018.

In the following sections, we analyze the performance of the QR factorizations and the pipeline throughput of both OpenMP and GrPPI versions computing the QR factorizations in sequential and in parallel.

5.1 Evaluation of the pipeline with sequential stages

In a first step, we evaluate the pipeline implementations using the sequential QR algorithm (Pipe-OMP-Seq and Pipe-GrPPI-Seq) with respect to a tiled algorithm and the same with a *jagged* matrix. Table 1 shows a comparison in terms of execution time for three different problem sizes with its respective *tile* size. The basic computational kernels that work on any *tile* are blocking routines for which, in all cases, the block size b_s was fixed to 40 [4]. The column labeled as *Rectangular* shows the execution time for the QR factorization with a tiled algorithm [14]. The column labeled as *Jagged* shows the execution time required to obtain the QR factorization of a matrix of the type shown in Fig. 1(b) with the algorithm presented in [4]. The last 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 in Stage 4 at Step 2 in Fig. 3 to upper triangular. Note that we evaluate the QR factorization performed at Stage 4, as it is the largest one and acts as the pipeline bottleneck. It can be seen that all the times obtained are coherent with the computations carried out. The use of the pipeline allows to speed up the computation more than $\times 2$ the time for *jagged* matrices, and close to $\times 3$ the time for the QR factorization of rectangular matrices.

To extend this analysis, Fig. 7 shows the QR factorization execution time using the tiled algorithm on *rectangular* and *jagged* matrices with different problem sizes and number of threads. As it can be seen, the best time-to-solution is obtained using 4 threads. Going beyond 4 threads does not bring any improvement given the fixed matrix layout of 4×3 tiles, where, according the DAG, hardly more than three tiles can be processed in parallel by individual threads.

5.2 Evaluation of the pipeline with parallel stages

In this section, we evaluate the execution time of the QR factorization performed at Stage 4 and the throughput (matrices/s) of the Pipe-OMP-Par and Pipe-

Table 1 Time in seconds to perform the QR factorization of a rectangular matrix, a *jagged* matrix, and using the pipeline (at Stage 4) for different matrix sizes.

$m \times n$	Tile size	*Rectangular*	*Jagged*	Pipe-OMP-Seq	Pipe-GrPPI-Seq
1280 × 960	320	0.074 s.	0.064 s.	0.023 s.	0.023 s.
2560 × 1920	640	0.379 s.	0.307 s.	0.129 s.	0.117 s.
3840 × 2880	960	1.184 s.	0.934 s.	0.363 s.	0.368 s.

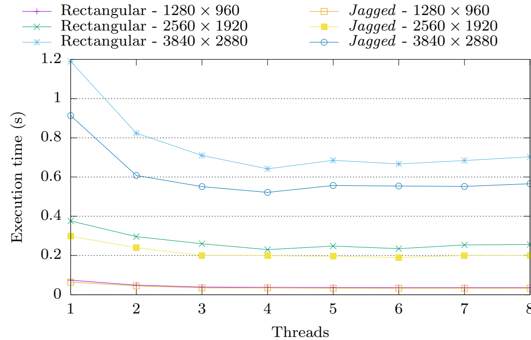
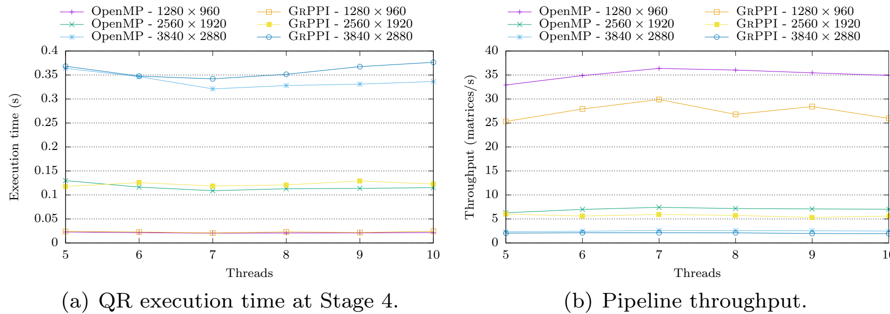


Fig. 7 *QR* factorization execution time using a tiled algorithm on *rectangular* and *jagged* matrices with varying number of threads.

GrPPI-Par implementations with increasing number of threads, from 5 to 10. Note that the use of 5 threads corresponds to the sequential version, as the pipeline is composed of 5 stages and no additional threads are left for the parallel execution of the *QR* factorization. Note as well the selected upper boundary of 10 threads; this is given due to the fixed matrix layout of 4×3 tiles, allowing the *QR* tiled algorithm to process only up to three tiles in parallel. As it can be seen in Fig. 8(a), the *QR* execution time at Stage 4 remains almost constant regardless the number of additional threads left for processing the *QR* subtasks. It is a bit noticeable the execution time decrease for 7 threads in all problem sizes. This is because the *QR* tiled algorithm at this stage can not leverage more than 2 threads; thus, if 5 threads are always dedicated to run the pipeline stages and more than 2 threads bring no performance benefits, the best execution time is obtained with 7 threads. In any case, if we compare these results with those shown in Figure 7, for the *QR* factorization execution time using the tiled algorithm on *rectangular* and *jagged* matrices, we can easily observe that pipeline-based algorithms always deliver better speedups. Concretely, it reduces, at least, 50 % the execution time of both *QR* factorizations on *rectangular* and *jagged* matrices for their best configuration using 4 threads.

On the other hand, Fig. 8(b) shows the pipeline throughput at the consumer stage (Stage 5), i.e. the ratio of upper triangular factors (matrices) delivered per second. As it can be seen, the figures are also almost constant w.r.t. the number of threads. As it occurs for the *QR* execution time and the reasons mentioned before, the best throughput is obtained when 7 threads are used. Furthermore, we also notice a throughput slowdown for the GrPPI pipeline in all problem sizes. We relate this slowdown to the modifications introduced into the application to fit the DAG into a pipeline with bistate stages to match the GrPPI interface, making slightly less efficient w.r.t. the OpenMP version. Although the slowdown

is considerable, the GrPPI benefits of using high-level patterns, i.e. less lines-of-code, better portability, and productivity, may pay off the use of the interface in some cases.



(a) QR execution time at Stage 4. (b) Pipeline throughput.

Fig. 8 QR factorization execution time at Stage 4 and total throughput of the Pipe-OMP-Par and Pipe-GrPPI-Par implementations with varying number of threads.

6 Conclusions

The QR factorization of a rectangular matrix is a basic computational kernel very used in many scientific applications. The application tackled in this paper, the Beamformer problem, is defined by a rectangular matrix that changes quickly (real-time) with a given set of input signals (of digital sound). The factorization of this matrix, which is required to build the corresponding output signals, is the most time-consuming task of those that are part of the Beamforming Algorithm, and must be computed rapidly to keep the track of inputs. We have shown that arranging the computational resources available in a computer system in a pipeline structure allows reducing the execution time of this factorization.

The basic idea was defined as a special DAG, very close to a pipeline (quasi-pipeline), that can be implemented with the OpenMP programming tool. In order to use higher level programming tools, we flattened the former DAG or quasi-pipeline to derive a real pipeline that can be easily implemented using GrPPI. Comparing the performance obtained with the two tools we can consider that both are almost at the same performance level.

The features of the physical system under test imposes a shape to the system matrix unsuitable to be accelerated with parallelism, i.e. the degree of concurrency is very low. We have shown that, with our pipeline, the number of threads that can be exploited to improve performance and throughput can be higher.

Acknowledgments

This work was supported by the Spanish Ministry of Economy and Competitiveness under MINECO and FEDER projects TIN2014-53495-R and TEC2015-67387-C4-1-R.

References

1. Yiteng Huang, Jacob Benesty, and Jingdong Chen. *Acoustic MIMO Signal Processing (Signals and Communication Technology)*. Springer-Verlag, Berlin, Heidelberg, 2006.
2. C. Ramiro, A.M. Vidal, and A. González. MIMOPack: A High Performance Computing Library for MIMO communication systems. *The Journal of Supercomputing*, 71:751–760, 2015.
3. F.J Alventosa, P Alonso, G Piñero, and A.M. Vidal. Implementation of the beamformer algorithm for the nvidia jetson. pages 201–211, Granada, Spain, 2016. Actas de la Conferencia (ISBN 978-3-319-49955-0).
4. Fran J. Alventosa, Pedro Alonso, Antonio M. Vidal, Gema Piñero, and Enrique S. Quintana-Ortí. Fast block qr update in digital signal processing. *The Journal of Supercomputing*, Mar 2018.
5. David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and José Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24), 2017.
6. Jacob Benesty, Jingdong Chen, Yiteng Huang, and Jacek Dmochowski. On microphone-array beamforming from a MIMO acoustic signal processing perspective. *IEEE Trans. Audio, Speech & Language Processing*, 15(3):1053–1065, 2007.
7. J Lorente, G Piñero, A.M. Vidal, J.A. Belloch, and A González. Parallel implementations of Beamforming design and Filtering for Microphone Array Applications. In *19th European Signal Processing Conference (EUSIPCO)*, pages 501–505, Barcelona, Spain, 2011.
8. J.A. Belloch, M. Ferrer, A. González, F.J. Martínez-Zaldívar, and A.M. Vidal. Headphone-based Virtual Spatialization of Sound with a GPU Accelerator. *Journal of the Audio Engineering Society*, 61:546–561, 2013.
9. Jose A. Belloch, Alberto González, F. J. Martínez-Zaldívar, and Antonio M. Vidal. Real-time massive convolution for audio applications on GPU. *The Journal of Supercomputing*, 58(3):449–457, Dec 2011.
10. G.H. Golub and C.F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 2013.
11. 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.
12. 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.
13. Manuel F. Dolz, Fran J. Alventosa, Pedro Alonso-Jordá, and Antonio M. Vidal. A pipeline for the QR update in digital signal processing. In *Proceedings of the 18th International Conference on Computational and Mathematical Methods in Science and Engineering (CMMSE 2018)*, pages 1–5, Rota, Cádiz, Spain, July 2018.
14. 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.
15. David del Rio Astorga, Manuel F. Dolz, Javier Fernández, and J. Daniel García. A generic parallel pattern interface for stream and data processing. *Concurrency and Computation: Practice and Experience*, 29(24):e4175, 2017. e4175 cpe.4175.