Dataflow Systolic Array Implementations of Exploring Dual-Triangular Structure in QR Decomposition Using High-Level Synthesis

Siyang Jiang, Hsi-Wen Chen, Ming-Syan Chen✉
Department of Electrical Engineering, National Taiwan University
{syjiang, hwchen}@arbor.ee.ntu.edu.tw, mschen@ntu.edu.tw

***Abstract*—Tall and skinny QR (TSQR) decomposition is an essential matrix operation with various applications in edge computing, including data compression, subspace projection, and dimension reduction. As a critical component in TSQR, Dual-Triangular QR (DTQR) decomposition is solved by the Normal QR method in most works without utilizing the dual-triangular structure. Therefore, we propose a novel DTQR accelerator by recursively exploring the DT structure and propose three acceleration strategies with the systolic array to achieve higher parallelism. Experimental results manifest that our algorithm achieves 21.55x on average speedup compared with the baselines.**

***Index Terms*—QR decomposition, Dual-triangular matrix, High-Level Synthesis**

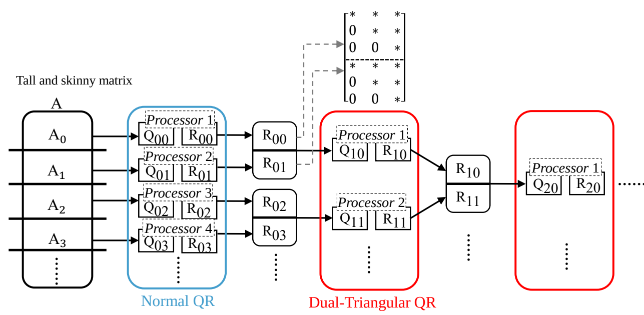


Fig. 1: **TSQR decomposition progress.** In TSQR decomposition, target matrix A is divided into several small parts (e.g., A_0, A_1 ...), and is decomposed by two operations, i.e., Normal QR and DTQR in the blue box and red box respectively [5].

I. Introduction

The QR decomposition aims to decompose a matrix into an orthogonal matrix Q and an upper triangular matrix R. For data statistics [1], [2], the matrices are commonly *tall and skinny (TS)* in streaming data services. For instance, a video system generates thousands of pixels in a frame (represented by rows) and tens of frames (represented by columns), which can be regarded as a TS matrix. To accelerate the *tall and skinny QR (TSQR)* decomposition, several works [3] have introduced different acceleration schemes, which typically take advantage of the powerful computation platforms, e.g., GPU. However, Zhang *et al.* [1] have shown that directly employing TSQR decomposition may lead to unacceptable performance on the edge devices, e.g., FPGAs, due to the limitation in storage and computation resources [1], [4].

TSQR decomposition consists of two primary operations, i.e., *Normal QR* and *Dual-triangular QR* (DTQR) (Fig. 1). While DTQR occupies half of the calculation, most existing works mainly focus on the Normal QR [6] and decompose the dualtriangular (DT) matrix by employing the Normal QR. Therefore, in this paper, we investigate the DT structure in submatrices [5] to accelerate the DTQR. First, we explore the recursive form of decomposing DT matrix and propose three acceleration schemes, i.e., *Parallel Rotation Parameter Generation*, and *Pipeline Rotation*, and *Dataflow*, to reduce latency. Our accelerator achieves $21.55\times$ speedup compared to the baselines.

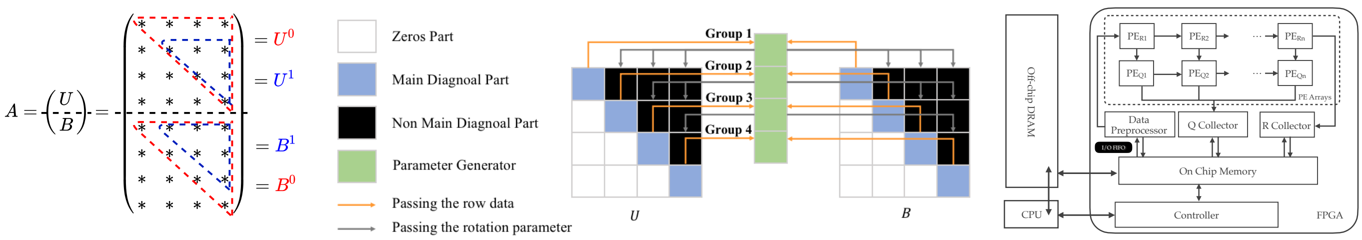
II. Related Works

Since TSQR decomposition is a fundamental primitive in several real applications [1], several works have studied the acceleration of the TSQR decomposition on edge devices, which is much more challenging due to limitations on both storage and computational resource [7]–[9]. Thus, Rafique *et al.* [8] schedule the data traffic time and the computing time using Household QR decomposition on edge devices. Wang *et al.* [9] propose a unified architecture with Householder reflection and Givens rotation. Another line of studies employs the divide-and-conquer method to transfer the TSQR problem into the Normal QR problem. Recent QR Decomposition work Desai *et al.* [10], Tan *et al.* [11], and Liu *et al.* [3], accelerate Normal QR via the Gram-Schmidt method, the modified Gram-Schmidt method and the Givens rotation method, but the accelerators aiming for accelerating DT matrix are missing.

III. Methodology

A. Preliminaries

In this paper, A denotes a matrix, a_i represents the i^{th} columns of matrix A, and $a_{i,j}$ represents the element in the i^{th} row, the j^{th} columns of A. DTQR decomposition aims to factor the target DT matrix $A \in \mathbb{R}^{2n\times n}$, into an orthogonal matrix $Q \in \mathbb{R}^{2n\times n}$ and upper triangular matrix $R \in \mathbb{R}^{n\times n}$, i.e., $A = QR$. To effectively solve DTQR,



(a) (b) (c)

Fig. 2: Illustration of (a) Operator Description, (b) Accelerating Strategies, and (c) Architecture Overview.

we employ Givens rotation QR decomposition [12] for our baseline method, because of the high parallelism and great numerical stability, especially suitable for sparse matrix (like DT matrix) [5], compared with other QR decomposition methods [12]. Specifically, the Givens rotation QR decomposition computes R from the DT matrix A by applying a sequence of rotations $G_i \in \mathbb{R}^{n\times n}$ to eliminate all the non-zero elements recursively, i.e., $R = G_k \cdots G_1 A$ and $Q = G_1^T \cdots G_k^T$, where G_i^T is the transpose matrix of G_i.

B. Recursive Form of Solving DTQR

Here, we present the recursive form to solve DTQR. Given a dual-triangular matrix A, which consist a upper matrix $U = A[0:n]$ and a bottom matrix $B = A[n:2n]$, our goal is to recursively compute the upper matrix $u_{i,j}^k \in U^k$ and the upper matrix $b_{i,j}^k \in B^k$ after k^{th} round elimination. Let $U^0 = U$ and $B^0 = B$. First, the rotation parameter c_i^k and s_i^k for polar coordinate transformation in k^{th} round is defined as follows.

$$c_i^k = \frac{u_{i,i}^k}{\sqrt{(u_{i,i}^k)^2 + (b_{i,i}^k)^2}}, \text{ and } s_i^k = \frac{b_{i,i}^k}{\sqrt{(u_{i,i}^k)^2 + (b_{i,i}^k)^2}} \quad (1)$$

By the definition of Givens rotation, we rotate the upper matrix U^k to $u_{i,j}^{k+1} \in U^{k+1}$ as follows.

$$u_{i,j}^{k+1} = \begin{cases} u_{i,j}^k c_i^k + b_{i,j}^k s_i^k, & \text{if } i \in [0,n], j \in [i,n] \\ u_{i,j}^k, & \text{otherwise} \end{cases}. \quad (2)$$

Similarly, the bottom matrix $b_{i,j}^{k+1} \in B^{k+1}$ becomes,

$$b_{i,j}^{k+1} = \begin{cases} -u_{i,j}^k s_i^k + b_{i,j}^k c_i^k, & \text{if } i \in [0,n-k], j \in [i,n] \\ b_{i,j}^k, & \text{otherwise} \end{cases}. \quad (3)$$

The major difference is cross column-wise elimination. As illustrated in Fig. 2(a). In the first round (U^0, B^0), we calculate the element in the two red triangles, and in the second round (U^1, B^1), we compute the two blue triangles. Compared to the convectional column-wise elimination, i.e., Gaussian elimination, we recursively eliminate the elements across different columns for higher parallelism.

C. Accelerating Strategies

Equipped with the recursive rotation, we introduce three strategies to accelerate the decomposition in different levels.

First, we propose the *Parallel Rotation Parameter Generation (PG)*. As shown in Fig. 2(b), the target matrix A is divided into matrix $U_{4\times4}$ and matrix $B_{4\times4}$. The orange lines send the raw data to generate rotation parameters, and the gray lines send back the parameters for rotating. As memory limitation on edge devices, the zeros part would not be considered and stored in implementation. The diagonal elements of the corresponding row become groups and are sent to the parameter generator (the green boxes). It is worth noting that each group of rotation parameters is generated independently and simultaneously. After obtaining the rotation parameters, i.e., c_i^k and s_i^k, in Eq. (1), rotation parameters in different groups are employed to rotate the non-main-diagonal part according to the group index, and thus reduce the time complexity of generation from $O(n)$ to $O(1)$.

At the same time, the non-main-diagonal part is rotated during the generation of the rotation parameter. As generation usually requires more computation resources than rotation, we introduce *Pipeline Rotation (PR)* to utilize resources better. The advantage of pipeline rotation overlaps different processing stages, i.e., read, compute, write, during rotating in Eq. (2) and Eq. (3), and reduce latency in a constant-level.

To further accelerate the DTQR process, we also use *Dataflow (DF)*, a task-level pipeline. Specifically, the parameters flow into the subsequent elimination round to rotate the data in the non-diagonal parts and matrix Q after rotation. Therefore, it initiates the subsequent elimination rounds before the previous operation is finished, thereby overlapping computing time and leading to a constant improvement through stacking parts of the computing costs over each elimination.

D. Implementation.

In addition to the aforementioned strategies, we present, in the beginning, the *Diagonal-Major Based Data Rearrangement (DR)* to rearrange both the upper matrix U and the bottom matrix B into two parts, i.e., the diagonal part and the non-diagonal part (Fig. 2(b)). Since the rotation parameter generation only occurs in the diagonal part, we can reduce the access time by allocating the diagonal part adjacently in the memory. Besides, we can also accelerate the whole rotation progress by rearranging the non-diagonal part as we sequentially process the data across different PEs via *DF*.

We illustrate the overall architecture of our DTQR accelerator in Fig. 2(c). Firstly, data is transferred from off-chip memory to on-chip memory (BRAM) using direct memory

Algorithm 1: Parallel Recursive Rotation DTQR

Input: DT matrix $A \in \mathbb{R}^{2n \times n}$

Output: Unitary matrix $Q \in \mathbb{R}^{2n \times n}$; Upper Triangular matrix $R \in \mathbb{R}^{n \times n}$;

$Q = I_{2n \times 2n}$; $U = upper(A)$; $B = bottom(A)$;
#pragma HLS dataflow
for k = *0 to n-1* **do**
    $U^{(k)} = \varphi^{(k)}(U)$; $B^{(k)} = \psi^{(k)}(B)$;
    for i = *0 to n - k - 1* **do**
        #pragma HLS unroll
        /* Rotation Parameter Generation*/
        $norm\_pre = dot(u_{i,i}^{(k)}, b_{i,i}^{(k)})$; $norm = sqrt(norm_{pre})$;
        $c_i^{(k)} = u_{i,i}^{(k)}/norm$; $s_i^{(k)} = b_{i,i}^{(k)}/norm$;
        $u_{i,i}^{(k)} = norm$; $b_{i,i}^{(k)} = 0$;
    for i = *0 to n-k-1* **do**
        for j = *i+1 to n-k-1* **do**
            #pragma HLS pipeline /* Rotation R */
            $u_{i,j}^{(k)} = c_i^{(k)} u_{i,j}^{(k)} + s_i^{(k)} b_{i,j}^{(k)}$;
            $b_{i,j}^{(k)} = -s_i^{(k)} u_{i,j}^{(k)} + c_i^{(k)} b_{i,j}^{(k)}$;
        for l = *0 to 2n-1* **do**
            #pragma HLS pipeline /* Rotation Q */
            $q_{l,k+i} = c_i^{(k)} q_{l,k+i} + s_i^{(k)} q_{l,i+n}$;
            $q_{l,i+n} = -s_i^{(k)} q_{l,k+i} + c_i^{(k)} q_{l,i+n}$

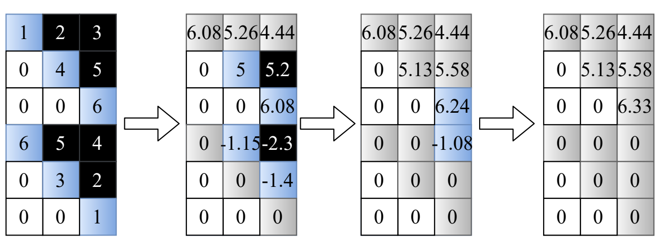


Fig. 3: Illustrative Example.

access. Next, Diagonal-Major Based Data Rearrangement is deployed on the data preprocessor. Then, the data stream is transferred into the sequence of processing engines PE_R, which are used for computing matrix R using Parallel Rotation Parameter Generation (Eq. 1) and Pipeline Rotation (Eq. 2 and Eq. 3). PE_{Qi} is only employed to calculate matrix Q with deploying Pipeline Rotation (Eq. 2 and Eq. 3). Lastly, Q Collector and R Collector is used for collecting the matrix Q and R from each PE_{Qi} and PE_{Ri}, respectively. Note that we adopt the systolic array (SA) [3] to chain the sequence of PE_{Ri} and PE_{Qi}. Dataflow is deployed among PE_{Ri} and PE_{Gi} for decreasing the latency, i.e., the rotation is launched, once parameters are generated in PE_{Ri}. The pseudo-code of our accelerator is presented in Algorithm 1.

E. Illustrative Example

Fig. 3 illustrates an example of our approach. In each elimination round, we first use the main diagonal part (in the blue boxes) to generate the rotation parameters in parallel, which is employed to rotate the non-main diagonal part (in the black boxes). Lastly, the final results (the grey boxes in the first four rows) are obtained. The dataflow is used as a task-level pipeline for reducing latency across the different elimination rounds.

IV. Experiments

A. Experiment Setup

We compare four methods as our baselines. i) *Original Method (OM)* [12] is a classical QR decomposition method on FPGAs and CPU. ii) *HLS LIB* is the standard linear algebra library from Xilinx. iii) *1D* SA [3] and iv) *2D SA* [3] are both acceleration methods for QR Decomposition based on systolic arrays. We deploy all methods on device Zynq-ZCU104 Ultrascale+ FPGA (xczu7ev-ffvc1156-2-e) with 100MHz clock frequency and adopt single-precision floating-point. We choose Intel i7-9700K@3.6GHz as our CPU platform. We implement it in custom C code and use Vivado HLS 2019.2 to compile to RTL. The test benches are randomly generated by MATLAB.

B. Quantitative Evaluation

Table. I illustrates the result of our method compared to the other four baselines. Specifically, our method outperforms OM with 57.23$\times$ and 2.67$\times$ on FPGA and CPU, respectively. In addition, our method speeds up HLS LIB 21.55$\times$ on average. Besides, our approach requires a similar resource with 11.85$\times$ speedups compared to 1D SA. Furthermore, we significantly decrease resource utilization compared to 2D SA with 4.39$\times$ speedups, demonstrating the effectiveness of our frameworks. Note that several cases in 1D SA (i.e., 24) and 2D SA (i.e., 12, 16, and 24) are out of memory (OOM) due to the limited resources on FPGA, which also manifest the efficiency of our frameworks.

Table II presents the ablation studies of six variances of our model i) PG, ii) PR, iii) DF, iv) PG+PR, v) PG+DF, and vi) PG+PR+DF. All accelerating strategies improve the efficiency when the matrix dimension increases, demonstrating the scalability of our method. While only using PG scheme has relatively weak performance due to the overhead in data traffic and blocking data flowing to the next round without other methods, i.e., DF or PR significantly boosts efficiency.

C. Bottleneck Analysis

In this section, we consider the computing time bottleneck, it is transferred from computing R (represented by the blue bar) into computing Q (represented by the yellow bar) when the size of the matrix increases, according to the result from Fig. 4, shown as computing time percentage. The pipeline rotation and dataflow accelerating strategy could not decrease the computational time of Q directly but only more effectively overlap the computation time.

TABLE I: Comparison with previous work in same platform and frequency (100MHz)

Method		Metrics	Dual 4×4	Dual 8×8	Dual 12×12	Dual 16×16	Dual 24×24	Average Ratio
OM [12]	i7-9700K	Speed Ratio	3.11x	2.72x	3.19x	2.23x	2.11x	2.67x
	ZCU104	Cycle	3705	28337	735505	204558	1023559	57.23x
		Speed Ratio	18.90x	45.26x	50.10x	53.90x	70.60x	
		LUTs	1%	1%	1%	1%	1%	
		FF	∼0%	∼0%	∼0%	2%	∼0%	
		DSP48E	∼0%	∼0%	∼0%	∼0%	∼0%	
		BRAM	0%	0%	0%	0%	0%	
HLS LIB [13]		Cycle	1852	12006	40007	83208	347227	21.55x
		Speed Ratio	9.45x	19.18x	19.76x	20.37x	23.95x	
		LUTs	3%	3%	3%	3%	3%	
		FF	2%	2%	2%	2%	2%	
		DSP48E	1%	1%	1%	1%	1%	
		BRAM	0̃%	0̃%	1%	2%	4%	
1D SA [3]		Cycle	1723	8264	28247	55590	OOM	11.85x
		Speed Ratio	8.79x	13.20x	11.83x	13.61x	OOM	
		LUTs	9%	21%	36%	43%	OOM	
		FF	2%	6%	11%	13%	OOM	
		DSP48E	4%	7%	11%	15%	OOM	
		BRAM	5%	10%	16%	21%	OOM	
2D SA [3]		Cycle	709	3239	OOM	OOM	OOM	4.39x
		Speed Ratio	3.62x	5.17x	OOM	OOM	OOM	
		LUTs	11%	33%	OOM	OOM	OOM	
		FF	4%	13%	OOM	OOM	OOM	
		DSP48E	8%	30%	OOM	OOM	OOM	
		BRAM	14%	49%	OOM	OOM	OOM	
Ours		Cycle	196	626	2387	4083	14498	1.0x
		Speed Ratio	1.0x	1.0x	1.0x	1.0x	1.0x	
		LUTs	9%	21%	33%	45%	74%	
		FF	3%	8%	13%	19%	31%	
		DSP48E	6%	13%	21%	29%	43%	
		BRAM	11%	27%	41%	56%	95%	

TABLE II: Ablation study (*OM as the baseline*)

Matrix Size	Dual 4×4	Dual 8×8	Dual 16×16	Dual 24×24
PG	0.95x	0.94x	0.96x	0.97x
PR	2.14x	3.25x	4.09x	4.14x
DF	**4.79x**	**10.79x**	**22.35x**	**33.81x**
PG+PR	5.80x	12.14x	19.72x	22.92x
PR+DF	10.83x	24.29x	35.34x	41.32x
PG+DF	12.86x	31.01x	43.42x	50.35x
PG+PR+DF	**18.90x**	**45.26x**	**53.90x**	**70.60x**

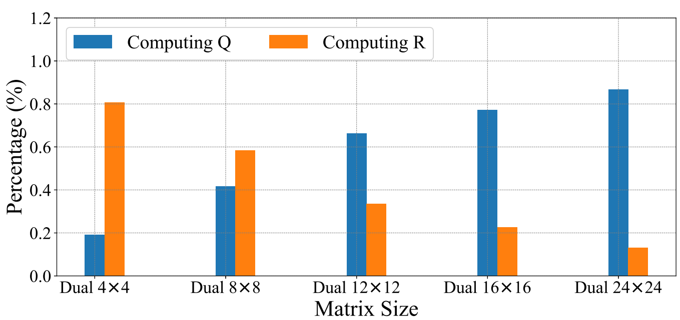


Fig. 4: Bottleneck of Computing Time.

V. Conclusion

We propose a DTQR accelerator on FPGAs via the DT structure. Experimental results show that our method achieves 21.55× speedups. Future work includes exploring the acceleration of DTQR in extremely resource-limited situations.

Acknowledgement

The work is in part supported by MOST Project No. 108-2221-E-002 -022 -MY3, Taiwan.

References

[1] S. Zhang, E. Baharlouei, and P. Wu, "High accuracy matrix computations on neural engines: A study of qr factorization and its applications," in *ACM HPDC*, 2020.

[2] Y. Liang, L. Lu, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on fpgas," *IEEE TCAD*, 2019.

[3] J. Liu and J. Cong, "Dataflow systolic array implementations of matrix decomposition using high level synthesis," in *ACM/SIGDA FPGA*, 2019.

[4] X. Hao, S. Yang, B. Deng, J. Wang, X. Wei, and Y. Che, "A cordic based real-time implementation and analysis of a respiratory central pattern generator," *Neurocomputing*, 2021.

[5] N.-Y. Cheng and M.-S. Chen, "Exploring dual-triangular structure for efficient r-initiated tall-skinny qr on gpgpu," in *PAKDD*, 2019.

[6] M. Langhammer and B. Pasca, "High-performance qr decomposition for fpgas," in *ACM/SIGDA FPGA*, 2018.

[7] S. I. Venieris, G. Mingas, and C.-S. Bouganis, "Towards heterogeneous solvers for large-scale linear systems," in *IEEE FPL*, 2015.

[8] A. Rafique, N. Kapre, and G. A. Constantinides, "Enhancing performance of tall-skinny qr factorization using fpgas," in *IEEE FPL*, 2012.

[9] X. Wang, P. Jones, and J. Zambreno, "A reconfigurable architecture for qr decomposition using a hybrid approach," in *2014 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2014.

[10] P. Desai, S. Aslan, and J. Saniie, "Fpga implementation of gram-schmidt qr decomposition using high level synthesis," in *IEEE EIT*, 2017.

[11] C. Y. Tan, C. Y. Ooi, and N. Ismail, "Loop optimizations of mgs-qrd algorithm for fpga high-level synthesis," in *IEEE SOCC*, 2019.

[12] J. W. Demmel, *Applied numerical linear algebra*. SIAM, 1997.

[13] V. Xilinx, "Vivado design suite user guide-high-level synthesis," 2014.