Acceleration of Parallel-Blocked QR Decomposition of Tall-and-Skinny Matrices on FPGAs

JOSE M. RODRIGUEZ BORBON, JUNJIE HUANG, BRYAN M. WONG, and
WALID NAJJAR, University of California, Riverside

QR decomposition is one of the most useful factorization kernels in modern numerical linear algebra algorithms. In particular, the decomposition of tall-and-skinny matrices (TSMs) has major applications in scientific computing, machine learning, image processing, wireless networks, and numerical methods. Traditionally, CPUs and GPUs have achieved better throughput on these applications by using large cache hierarchies and compute cores running at a high frequency, leading to high power consumption. With the advent of heterogeneous platforms, however, FPGAs are emerging as a promising viable alternative. In this work, we propose a high-throughput FPGA-based engine that has a very high computational efficiency (ratio of achieved to peak throughput) compared to similar QR solvers running on FPGAs. Although comparable QR solvers achieve an efficiency of 36%, our design exhibits an efficiency of 54%. For TSMs, our experimental results show that our design can outperform highly optimized QR solvers running on CPUs and GPUs. For TSMs with more than 50K rows, our design outperforms the Intel MKL solver running on an Intel quad-core processor by a factor of 1.5 $\times$. For TSMs containing 256 columns or less, our design outperforms the NVIDIA CUBLAS solver running on a K40 GPU by a factor of 3.0 $\times$. In addition to being fast, our design is energy efficient—competing platforms execute up to 0.6 GFLOPS/Joule, whereas our design executes more than 1.0 GFLOPS/Joule.

CCS Concepts: • **Hardware** $\rightarrow$ **Hardware accelerators**; *Arithmetic and datapath circuits*; • **Computer systems organization** $\rightarrow$ *Multicore architectures*;

Additional Key Words and Phrases: QR decomposition, accelerators, FPGA, reconfigurable computing

ACM Reference format:

Jose M. Rodriguez Borbon, Junjie Huang, Bryan M. Wong, and Walid Najjar. 2021. Acceleration of Parallel-Blocked QR Decomposition of Tall-and-Skinny Matrices on FPGAs. *ACM Trans. Arch. Code Optim.* 18, 3, Article 27 (April 2021), 25 pages.
<https://doi.org/10.1145/3447775>

1 INTRODUCTION

One of the fundamental problems in high performance computing is the fast decomposition of a matrix A into two or more factors. Notable algorithms include the Cholesky, LU , QR , SVD , Eigen,

This work was supported by the Office of Naval Research (grant N00014-18-1-2740).

Authors' addresses: J. M. R. Borbon, J. Huang, and W. Najjar, Department of Computer Science & Engineering, University of California, Riverside, Riverside, CA 92521; emails: {jrodr050, jhuan308}@ucr.edu, najjar@cs.ucr.edu; B. M. Wong, Department of Chemical & Environmental Engineering, University of California, Riverside, Riverside, CA 92521; email: bryan.wong@ucr.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2021 Association for Computing Machinery.

1544-3566/2021/04-ART27 \$15.00

<https://doi.org/10.1145/3447775>

and Schur decompositions [43]. In this work, we examine the design of energy-efficient high-throughput engines for the QR decomposition of **tall-and-skinny matrices (TSMs)** composed of a few hundred columns and hundreds of thousands of rows.

Due to its numerical stability, the QR decomposition of TSMs has pervasive applications, with the most well known being the solution of least squares problems [20, 46, 52]. For the most popular algorithms that use least squares in machine learning, the input matrices have a few hundred columns that correspond to the observed parameters, and thousands of rows, which represent the number of observations. The work of Dua and Graff [16] describes typical datasets for machine learning, with each instance containing a few to hundreds of parameters and thousands to hundreds of thousands of rows. In the field of video and image processing for stationary background subtraction [8], the input matrices have a few hundred columns that correspond to the given images and tens of thousands of rows representing the pixels. In particular, the work of Anderson et al. [2] describes the QR decomposition of TSMs for inputs containing more than 100,000 rows (the pixels) and a few hundred columns (the images). In the field of wireless communication [9], the input matrices have dozens of rows that correspond to the number of receiving antennas and a few columns that represent the number of transmitting antennas. By observing that the exchange of data in modern multiprocessors is more expensive than the execution of arithmetic operations, Demmel et al. [14] proposed communication-avoiding algorithms, among them, the QR factorization of TSMs. In particular, for inputs having tens of thousands of rows and a few hundred columns, these algorithms trade computations for communications while exploiting the locality of the data. Additional applications include the computation of eigenvalues [52] and the computation of Krylov subspaces [46]. In all of these applications, the fast QR decomposition of TSMs is paramount.

Multiple studies [3, 28, 31, 36, 37] have addressed the design of efficient cores for the QR decomposition of matrices using methods such as **Gram-Schmidt Orthogonalization (GS)** (and its modifications), the **Cholesky Decomposition (CH)**, and **Givens Rotations (GR)**. However, the QR decomposition via **Householder reflectors (HRs)** has, surprisingly, received less attention even though the algorithm is numerically stable, parallelizable, and has lower computational complexity [25, 46]. Indeed, the HR decomposition method has been named as one of the 10 most important algorithms of the past century [15].

In this article, we propose an engine capable of decomposing TSMs in parallel with *resource-aware* reduction circuits [18, 53], thereby achieving a higher computational efficiency. Moreover, we take advantage of additional optimizations including tiling, double buffering, wide and deep pipelines, and memory burst accesses (several of which were utilized in our previous work [39]). We have implemented an HR decomposition engine that targets the factorization of TSMs on a Micron SB-852 board [30]. The performance of the proposed core is compared against two configurations: (1) the QR solver within the Intel MKL routines [48] running on an Intel quad-core processor and (2) the QR solver in the **CUDA basic linear algebra subroutines (CUBLAS)** [32] running on a K40 GPU.

The contributions of our work are threefold:

- We develop a *flexible* and *scalable* QR solver that targets the decomposition of TSMs on FPGAs. By taking advantage of the numerical stability of the HR method, along with *resource-aware* reduction circuits, our design splits the input matrix into a series of blocks and executes the QR decomposition in parallel. Multiple parallel instances of this core are placed and routed on a Virtex UltraScale+ FPGA running at 266 MHz. Although the proposed architecture targets server-scale FPGAs, it can also be scaled down for designs targeting embedded systems. For the task at hand, our design achieves the highest efficiency (54%) compared to similar FPGA designs (36%).

- The performance of the proposed engine matches and surpasses that of the Intel MKL QR solver [48], a highly optimized library, running on a quad-core CPU. For matrices having a few thousands rows, our engine matches the performance of the MKL QR solver. As the number of rows in the input matrix increases, our design outperforms the MKL QR solver by a factor of $1.5\times$. Compared to the performance of the CUBLAS QR solver [32] running on a GPU, our design achieves a speedup ranging from $1.5\times$ to $3.0\times$ for matrices having up to 256 columns. When the input matrix has 512 columns, our design closely follows the performance of the library running on the GPU; however, it executes more **floating-point operations (FLOPS)** per Joule.
- Our experimental results show that while CPUs and GPUs execute at most 0.45 and 0.60 GFLOPS/Joule, respectively, our design executes 1.03 GFLOPS/Joule. These gains in energy efficiency are obtained because the proposed engine uses *wide* and *deep* pipelines: when the input matrix has a few hundred columns, the proposed engine executes $2.3 \times (12.4\times)$ more FLOPS per clock cycle than the GPU (CPU).

The remainder of this article is organized as follows. We present related work in Section 2. We introduce the QR decomposition of TSMs in Section 3 and describe the proposed FPGA engine in Section 4. We analyze the performance and the efficiency of our design by comparing its throughput and energy efficiency with those of a QR solver running on CPU and GPU platforms in Section 5. Finally, we present our conclusion in Section 6.

2 RELATED WORK

Techniques for improving the performance of QR decomposition on CPUs have been previously proposed [5, 14, 22]. In particular, methods for consolidating the application of HRs via matrix multiplications [5] have enabled the accelerated computation in platforms that have large caches. Communication-avoiding methods [14] for QR decomposition on multiple nodes have also been developed to minimize the amount of data exchanged between nodes.

For GPUs, high-performance methods for QR decomposition have been presented in other works [1, 2, 26]. Most notably, a high-performance method has been previously used to execute the entire QR decomposition on GPUs [26]. This method takes advantage of highly optimized GPU-specific matrix multiplication routines and outperforms existing libraries, such as the MKL [48] and MAGMA [45], for square matrices containing thousands of columns. A communication-avoiding QR decomposition method for GPUs was also proposed in the work of Anderson et al. [2]. Static scheduling methods have also been shown to perform well on heterogeneous platforms (GPUs and CPUs), whereas dynamic scheduling methods perform better in platforms with multiple GPUs for each CPU [1].

A map-reduce architecture to solve the least-squares problem via the QR factorization of TSMs was described in the work of Constantine and Gleich [12]. This approach takes advantage of communication-avoiding algorithms: each node executes a local QR decomposition and computes the factor $\mathbf{R}$, which subsequently exchanges factors with other nodes until the final decomposition is executed. By taking advantage of communication-avoiding algorithms, it has been shown that decomposition methods based on Householder reflections are faster than CH approaches [4].

Implementations of QR decomposition of matrices on FPGAs, including GSs, CHs, and GRs, have been proposed in other works [6, 18, 40, 50, 51]. QR decomposition via the HR method has not received as much attention despite its compelling features, including a lower computational complexity, higher stability, and a larger degree of parallelism [20, 46]. Table 1 compares our work with existing HR approaches.

Table 1. Comparison of QR Decomposition via HR Designs in FPGAs

Work	Tai et al. [44]	Rafique et al. [38]	This Work
Year	2011	2012	2020
Synthesis Tool	ISE 10.0	ISE 10.0	Vivado 17.2
FPGA	Virtex-5	Virtex-6	Virtex (Ultrascale+)
Frequency MHz	150	315	266
Peak GFLOPS	10.2	129	68
Max. FLOPS/Cycle	64	409	256
Efficiency for TSM (%)	7.0–11.0	36.0	28.7–54.2
Target Matrix Shape	Square	TSM	TSM
Matrix Shape (Rows, Columns)	(10K, 10K)	(10K, 51)	(100K, 64–512)
Reductions (Dot Products)	Resource aware	Resource intensive	Resource aware
Parallel Engines (Blocks Per Engine)	1 (2)	1 (2)	16 (2)
Pipelined Reflectors Per Engine	16	1	4

A QR decomposition engine targeting the factorization of large *square* matrices has been previously proposed [44]. In this design, the input matrix A is divided into multiple *square* blocks (along the horizontal and vertical axis), and the HR decomposition is subsequently executed in multiple steps:

- (1) HR decomposition is applied to the top left-most block, and the computed reflectors are saved into the off-chip memory.
- (2) The engine reads the remaining blocks in the top row (one at a time) and applies the saved reflectors.
- (3) The HR method takes the top left-most block (an upper-triangular block resulting from the previous step) and the block below it as input, and a decomposition is carried out. As before, the reflectors computed in this step are saved into the off-chip memory.
- (4) The engine reads the remaining blocks in the first and second row and applies the saved reflectors.

By combining these four steps, the QR factorization of the input matrix is achieved.

The differences between our work and previous studies are summarized in Table 1. First, the work of Tai et al. [44] targets the decomposition of large *square* matrices, whereas our work targets the decomposition of *TSMs*. Because *TSMs* have a few hundred columns, dividing the input matrix in blocks along the horizontal axis is not required, and as a result, the preceding steps (2) and (4) can be removed altogether. The resources allocated to those steps can be allocated elsewhere. Second, we make use of recent developments including the QR decomposition of matrices via *binary trees* and the fast decomposition of *upper triangular matrices* [14]. Third, our work targets the decomposition of a large number of blocks in parallel via multiple engines (as permitted by the platform), whereas the work of Tai et al. [44] allocates all of the resources to the processing of at most two blocks via a single engine. Fourth, due to the shape of the input matrices, our work targets *tall-and-skinny* tiles¹ instead of *square* tiles since the former favors a higher performance for the current problem as shown in Section 5. However, both works share common techniques such as the use of *resource-aware* reduction circuits [18], the application of *HRs* via deep pipelines, and the decomposition of on-chip tiles.

¹To take advantage of on-chip memory in our work, large blocks are tiled.

Table 2. Resources used by Raque et al. [38] as a Function of the Columns of the Input Matrix

Cols	Mults per Reduction	Adds per Reduction	DSPs per Reduction (10*Mult + 3*Add)	Total DSPs per Engine (3 Reductions)
51	25	25	325	975
64	32	31	413	1239
128	64	63	829	2487
256	128	127	1661	4983
512	256	255	3325	9975

In the work of Rafique et al. [38], an FPGA engine targeting the decomposition of *skinny* matrices with up to 51 columns is presented. In this work, the input matrix is first divided in blocks with twice as many rows as columns. Next, these blocks are brought to the on-chip memory. They are then decomposed via an HR decomposition engine. The output of the first step is a series of upper triangular blocks. In the next step, two triangular blocks are brought to the on-chip memory and decomposed. The results are written back to the off-chip memory. The process of reading, merging, and writing upper triangular blocks continues until the final decomposition is computed.

As shown in Table 1, our work has a number of differences from that of Rafique et al. [38]. First, our work targets the simultaneous decomposition of multiple blocks via many engines (with resources on the target platform being the limiting factor), whereas the approach used in the work of Rafique et al. [38] targets the decomposition of two blocks in parallel via a single engine. Second, large blocks are *tiled* before they are processed in our work, whereas individual blocks were placed in on-chip memory in the work of Rafique et al. [38], and as a result, only very skinny matrices can be decomposed. Third, the approach of Rafique et al. [38] relies on *resource-intensive* reduction circuits, whereas our approach uses *resource-aware* reduction circuits [18]. Resource-intensive reduction circuits are very fast but use *prohibitive* amounts of hardware; hence, they drastically limit the number of parallel engines, as shown in Table 2.

While the work in of Rafique et al. [38] focused on input matrices with 51 columns, we added the other four rows for illustrative purposes (notwithstanding possible changes to the original design to make it work for these four cases). Our benchmarks indicate that implementing a fairly small double-precision floating-point resource-intensive reduction tree with 128 inputs requires approximately $2,487 = 3(64 \times 10 + 63 \times 3)$ DSPs per engine.² Furthermore, when the input blocks are upper triangular (for TSMs, half of the matrices are upper triangular), the resources allocated to tree-based reduction circuits are severely underutilized. In particular, these trees achieve only 25% of their peak performance, as they have to operate on irrelevant values (i.e., the lower part of the input matrix). In contrast, our design uses *resource-aware* reduction circuits that require one multiplier and one adder (i.e., 13 DSPs) per reduction. Because *resource-aware* reduction circuits use minimal resources, they allow for the processing of many blocks in parallel via multiple engines and are highly efficient since they always operate on relevant values (i.e., the upper values of the input matrix).

3 QR DECOMPOSITION

QR decomposition factors a real valued matrix $\mathbf{A}_{n \times n}$ into two matrices, $\mathbf{Q}_{n \times n}$ and $\mathbf{R}_{n \times n}$, such that $\mathbf{A} = \mathbf{QR}$ with $\mathbf{Q}$ an orthogonal matrix ($\mathbf{QQ}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I}$) and $\mathbf{R}$ an upper triangular matrix

²The Vivado suite reports that each multiplication takes 10 DSPs, whereas each addition takes 3 DSPs.

Fig. 1. QR Decomposition for TSMs. This binary tree represents the QR decomposition of $\mathbf{A}$ such that $\mathbf{A}_i = \mathbf{Q}_i \mathbf{R}_i$ and $\begin{pmatrix} \mathbf{R}_j \\ \mathbf{R}_{j+1} \end{pmatrix} = \mathbf{Q}_a \mathbf{R}_a$.

($\mathbf{R}_{i,j} = 0$ for $i > j$) [20, 46, 52]. When the input matrix $\mathbf{A}$ is nonsingular, the decomposition is unique, given that the diagonal elements of $\mathbf{R}$ are positive. More generally, when $\mathbf{A}$ is $m \times n$, with $m \geq n$, the decomposition is still possible with $\mathbf{Q}$ being an $m \times m$ matrix and $\mathbf{R}$ being an $m \times n$ matrix. Specifically,

$$\mathbf{A} = \mathbf{QR} = [\mathbf{Q}_1 \ \mathbf{Q}_2] \begin{bmatrix} \mathbf{R}_1 \\ \mathbf{0} \end{bmatrix} = \mathbf{Q}_1 \mathbf{R}_1, \quad (1)$$

where $\mathbf{R}_1$ is an $n \times n$ matrix, $\mathbf{Q}_1$ is an $m \times n$ matrix, $\mathbf{Q}_2$ is an $m \times (m - n)$ matrix, and the matrix $\mathbf{0}$ has dimensions of $(m - n) \times n$.

3.1 QR Decomposition for TSMs

Although the QR decomposition can be applied to square matrices, our focus is on decomposing TSMs, $\mathbf{A}_{m \times n}$, such that $m \gg n$ [14]. As shown in Figure 1, the input matrix $\mathbf{A}$ of size $8n \times n$ is partitioned into four blocks $\mathbf{A}_1, \dots, \mathbf{A}_4$, where $\mathbf{A}_i$ has dimensions of $2n \times n$. The QR decomposition is implemented in three steps:

- (a) Four processors decompose $\mathbf{A}_1, \dots, \mathbf{A}_4$ such that $\mathbf{A}_1 = \mathbf{Q}_1 \mathbf{R}_1$, $\mathbf{A}_2 = \mathbf{Q}_2 \mathbf{R}_2$, $\mathbf{A}_3 = \mathbf{Q}_3 \mathbf{R}_3$, and $\mathbf{A}_4 = \mathbf{Q}_4 \mathbf{R}_4$ are computed in parallel.
- (b) Two processors decompose $\mathbf{R}_1, \dots, \mathbf{R}_4$ such that $\begin{pmatrix} \mathbf{R}_1 \\ \mathbf{R}_2 \end{pmatrix} = \mathbf{Q}_5 \mathbf{R}_5$ and $\begin{pmatrix} \mathbf{R}_3 \\ \mathbf{R}_4 \end{pmatrix} = \mathbf{Q}_6 \mathbf{R}_6$ are computed in parallel.
- (c) One processor decomposes $\mathbf{R}_5$ and $\mathbf{R}_6$ such that $\begin{pmatrix} \mathbf{R}_5 \\ \mathbf{R}_6 \end{pmatrix} = \mathbf{Q}_7 \mathbf{R}_7$.

In this figure, each $\mathbf{Q}_i$ has dimensions of $2n \times n$ and $\mathbf{R}_i$ is $n \times n$. Notice that step (b) takes the factors $\mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3$, and $\mathbf{R}_4$ from stage (a) as inputs. Likewise, step (c) uses the factors $\mathbf{R}_5$ and $\mathbf{R}_6$ from stage (b). As a result, one only requires the computation of matrices $\mathbf{R}_i$ in this approach. Once $\mathbf{R}_7$ is found, finding matrix $\mathbf{Q}$ is immediate because $\mathbf{Q} = \mathbf{A} \mathbf{R}_7^{-1}$.

As described, the QR decomposition of matrices can be achieved using four methods. The classical GS method is known to be numerically unstable due to rounding errors in finite precision arithmetic, although its instabilities can be removed via a modified approach [46]. Likewise, CH is known to be numerically unstable because the condition number of the matrix $\mathbf{A} \mathbf{A}^T$ is the square

of the condition number of $\mathbf{A}$ [20]. However, GR and HR methods are known to be numerically stable given that certain practices are observed [20]. In this work, we investigate the performance of HR for decomposing TSMs on FPGAs. Our decision is mainly based on the fact that this decomposition is numerically stable, and as a result, no additional hardware is dedicated to maintain its stability. Moreover, it has a lower computational complexity compared to the CH method [20, 52].

3.2 QR Decomposition Using Householder Reflections

Let $\mathbf{x} = (x_0, x_1, \dots, x_{n-1})^T$ be a vector. The HR method [46, 52] transforms $\mathbf{x}$ into $\mathbf{y} = (y_0, 0, \dots, 0)^T$ by constructing a matrix $\mathbf{Q}_{n \times n}$, usually called a *Householder reflector*, such that

$$\mathbf{Q} \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ 0 \end{pmatrix}. \quad (2)$$

The method transforms vector $\mathbf{x}$ into vector $\mathbf{y} = (-\sigma, 0, \dots, 0)^T$ where $\sigma = \pm \|\mathbf{x}\|_2$ is the 2-norm of the vector $\mathbf{x}$. The matrix $\mathbf{Q}$ that achieves such a transformation is built as follows. A vector $\mathbf{u}$ is defined as

$$\mathbf{u} = \mathbf{x} - \mathbf{y} = (x_0 + \sigma, x_1, \dots, x_{n-1})^T. \quad (3)$$

The matrix $\mathbf{Q}$ is given by

$$\mathbf{Q} = \mathbf{I} - \gamma \mathbf{u} \mathbf{u}^T, \quad (4)$$

where $\gamma = \frac{2}{\|\mathbf{u}\|_2^2}$, $\|\mathbf{u}\|_2^2 = (u_0^2 + u_1^2 + \dots + u_{n-1}^2)$, and $\mathbf{I}_{n \times n}$ is the identity matrix. In addition to annihilating multiple elements in a vector, the HR decomposition can be used to calculate the $\mathbf{R}$ component in the matrix decomposition $\mathbf{A} = \mathbf{QR}$. In this approach, the application of a set of HRs $\mathbf{Q}_1, \mathbf{Q}_2, \dots, \mathbf{Q}_n$ [46, 52] to matrix $\mathbf{A}$ leads to the computation of matrix $\mathbf{R}$.

ALGORITHM I: Canonical QR Decomposition of $\mathbf{A}$ Size $m \times n$

```

1  for  $k = 1$  to  $n - 1$  do
    // Step one—Generate HR
2     $\mathbf{x}_k = \mathbf{A}(k : m, k)$ 
3     $d_1 = \text{ddot}(\mathbf{x}_k, \mathbf{x}_k)$ 
4     $d_2 = \sqrt{d_1}$ 
5     $\mathbf{u}_k = \mathbf{x}_k$ 
6     $u_k(1) = x_k(1) + \text{sign}(x_k(1))d_2$ 
7     $d_3 = d_1 u_k(1)$ 
8     $\gamma_k = \frac{-2}{d_3}$ 
    // Step two—Update trailing columns of  $\mathbf{A}$ 
9    for  $j = k$  to  $n$  do
10      $\mathbf{a}_j = \mathbf{A}(k : m, j)$ 
11      $d_4 = (\gamma_k) \text{ddot}(\mathbf{a}_j, \mathbf{u}_k)$ 
12      $\mathbf{t}_j = \text{axpy}(\mathbf{a}_j, d_4, \mathbf{u}_k)$ 
13      $\mathbf{A}(j : m, j) = \mathbf{t}_j$ 
14   end for
15 end for
```

Algorithm I shows the canonical implementation of this decomposition. In this routine, the operation $\text{ddot}(\mathbf{x}, \mathbf{y})$ executes the dot product between the arguments. Likewise, the operation $\text{axpy}(\mathbf{x}, d, \mathbf{y})$ executes the vector subtraction $\mathbf{x} - (d)\mathbf{y}$. As shown in this routine, the canonical QR decomposition has two major steps: (1) the computation of the HR reflector (see Equation (4)), and (2) the updating of the trailing columns

Table 3. Computational Complexity Analysis

Task	Complexity
Householder Vector (A_i)	$3n^2 + n$
QA Mults + Adds (A_i)	$4n((5/6)n^2 + n + 1/6)$
Householder Vector (R_i)	$n^2 + 3n$
QR Mults + Adds (R_i)	$4n(n+1)(n/6 + 5/6)$

$$\mathbf{Q}_j \mathbf{a}_i = \mathbf{a}_i - \gamma_j (\mathbf{u}_j^T \mathbf{a}_i) \mathbf{u}_j, \quad (5)$$

where $\mathbf{a}_i$ is a column of $\mathbf{A}$.

3.3 HRs: Complexity Analysis

Now we analyze the computational complexity of applying the HR method to the decomposition shown in Figure 1. Table 3 summarizes our analysis. In this analysis, the matrices $\mathbf{A}_i$ have dimensions of $2n \times n$ while the matrices $\mathbf{R}_i$ are $n \times n$. The computation of the first vector $\mathbf{u}_1$ requires at least $2n$ multiplications with $2n$ additions plus the computation of the square-root operation. Next, the vector $\mathbf{u}_2$ has to be computed for a $(2n-1) \times (n-1)$ sub-matrix. Thus, the computation of all the vectors $\mathbf{u}_i$ requires at least $\sum_{i=0}^{n-1} (2n-i) = (3/2)n^2 + n/2$ multiplications and an equal number of additions. The complexity of the application of matrices $\mathbf{Q}_n, \dots, \mathbf{Q}_2, \mathbf{Q}_1$ can be computed in a similar fashion. In steps (b) and (c) of Figure 1, the computation of the vectors $\mathbf{u}_i$ is executed over columns of size 2 up to $n+1$ to take advantage of the upper triangular matrices $\mathbf{R}$, and otherwise the calculations are the same as before.

3.4 Canonical QR Decomposition in CPUs and GPUs

The HR decomposition in CPUs and GPUs is typically implemented via blocks. In this approach, the input matrix $\mathbf{A}_{m \times n}$ is divided into tiles [5, 26] such that $\mathbf{A} = [\mathbf{A}_1 \mathbf{A}_2 \dots \mathbf{A}_k]$ with $\mathbf{A}_i$ having dimensions of $m \times r$ and $k = n/r$ an integer. The method is shown in Algorithm II.

ALGORITHM II: Canonical QR Decomposition in CPUs and GPUs

Line	Step	Operation
1		for $j = 1$ to r do
2	S1	$[\mathbf{u}, \gamma] = \text{house}(\mathbf{A}_1(j : m, j))$
3	S2	$\mathbf{A}_1(j : m, j : r) = \mathbf{A}_1(j : m, j : r) - \gamma \mathbf{u}(\mathbf{u}^T \mathbf{A}_1(j : m, j : r))$
4	S2	$\mathbf{V}(:, j) = [\text{zeros}(j-1, 1); \mathbf{u}]$
5	S2	$\mathbf{B}(j) = \gamma$
6		end for
7	S3	$\mathbf{Y} = \mathbf{V}(:, 1) ; \mathbf{W} = -\mathbf{B}(1) \cdot \mathbf{V}(:, 1)$
8		for $j = 2$ to r do
9	S3	$\mathbf{u} = \mathbf{V}(:, j)$
10	S3	$\mathbf{z} = -\mathbf{B}(j) \cdot \mathbf{u} - \mathbf{B}(j) \cdot \mathbf{W}(\mathbf{Y}^T \mathbf{u})$
11	S3	$\mathbf{W} = [\mathbf{W} \mathbf{z}] ; \mathbf{Y} = [\mathbf{Y} \mathbf{u}]$
12		end for
13	S4	$\mathbf{A}(:, r+1 : n) = \mathbf{A}(:, r+1 : n) + \mathbf{Y} \mathbf{W}^T \mathbf{A}(:, r+1 : n)$
14	S5	execute step one

This algorithm takes five steps. In step one (S1), the reflectors $\mathbf{Q}_1, \mathbf{Q}_2, \dots, \mathbf{Q}_r$ for the tile $\mathbf{A}_1$ are computed. In step two (S2), the reflectors are applied to tile $\mathbf{A}_1$. In step three (S3), the reflectors $\mathbf{Q}_1, \mathbf{Q}_2, \dots, \mathbf{Q}_r$ are transformed. In step four (S4), the reflectors are applied to the remaining tiles of $\mathbf{A}$. In the last step (S5), the previous steps repeat starting from $\mathbf{A}_2^{(1)}$, where $\mathbf{A}_2^{(1)} = \mathbf{Q}_1, \mathbf{Q}_2, \dots, \mathbf{Q}_r, \mathbf{A}_2$.

Fig. 2. Tiling the QR decomposition. Instead of applying the QR decomposition on blocks of size $(2n - j) \times (n - j)$ as shown in (a), we partition the decomposition in tiles of size $(2n - j) \times t$. Next, we apply the QR decomposition to the left-most tile and save the reflectors as shown in (b). Finally, we apply these reflectors to the remaining tiles as shown in (c) and (d).

As far as the computational complexity of this routine, we note that S1 has a complexity proportional to $O(mr)$ since the execution time is dominated by the calculation of dot products involving vectors of size m . The complexity of S2 is proportional to $O(r(mr))$ since each iteration operates over matrices of size $m \times r$. The complexity of S3 is proportional to $O(r(rm))$ due to the presence of the matrix-vector product $\mathbf{Y}^T \mathbf{u}$ in addition to the product $\mathbf{W}(\mathbf{Y}^T \mathbf{u})$. Finally, the complexity of S4 is proportional to that of matrix multiplication.

4 PROPOSED ACCELERATOR ARCHITECTURE

In this section, we describe the design and implementation of the HR accelerator engine. The engine makes use of techniques to increase the performance in FPGAs, namely tiling, deep pipelines, double buffering, replication of pipelines, and memory bursting [10, 11].

4.1 Proposed Optimizations

Our parallel-blocked approach optimizes the QR decomposition of TSMs via a set of optimization techniques including (a) parallel-blocked decomposition, (b) tile QR decomposition, (c) efficient processing of the tiles via deep pipelines, (d) efficient processing of the tiles in the $\mathbf{R}_i$ blocks, (e) efficient computation of the dot products, (f) efficient access to the off-chip memory, (g) and efficient use of FPGA resources. In the following, we describe each optimization.

(a) Parallel-blocked QR decomposition. As shown in Figure 1, the QR decomposition of TSMs can be executed in parallel by multiple processing engines. In this regard, the decomposition of multiple blocks $\mathbf{A}_i$ in parallel is advantageous due to the large number of rows in the input matrix. Likewise, once the blocks $\mathbf{A}_i$ are decomposed, the decomposition of the blocks $\mathbf{R}_i$ in parallel is highly beneficial since the large number of rows in the input matrix implies the presence of multiple levels in the decomposition tree. In our work, we decompose multiple blocks in parallel as permitted by the availability of resources in the target device.

(b) Tiling the QR decomposition. At iteration j , as shown in Figure 2(a), the QR decomposition has to be applied to a $(2n - j) \times (n - j)$ block. Due to the iterative nature of the decomposition (see Algorithm 1), it is useful to store a large portion of this block in on-chip memory since storing the entire block is not feasible due to the limited memory resources on the FPGA. Thus, we tile the QR decomposition as shown in Figure 2(b) through (d). In this figure, the maximum size of the tile is $2n \times t$, where $2n$ is the maximum number of rows in $\mathbf{A}_i$ and t is the number of columns

Fig. 3. QR decomposition using HRs. At the top, the operation $\mathbf{a}_1^{(1)} = \mathbf{Q}_1 \mathbf{a}_1$ is executed via a shallow pipeline. At the bottom, the operation $\mathbf{a}_1^{(4)} = \mathbf{Q}_4 \mathbf{Q}_3 \mathbf{Q}_2 \mathbf{Q}_1 \mathbf{a}_1$ is executed via a deep pipeline.

in the tile. The QR decomposition using tiles involves two steps. In the first step, the QR decomposition is applied to the left-most tile as shown in Figure 2(b). This step involves the computation of t reflectors and the application of these reflectors to the columns in the tile. The first reflector $\mathbf{Q}_1$ is applied t times, and the last reflector $\mathbf{Q}_t$ is applied once. In addition, these reflectors are saved into the on-chip memory. In the second step, the saved reflectors are applied to the remaining tiles as shown in Figure 2(c) and (d). Here, each reflector is applied t times per tile. Notice that by adjusting t , we can tailor the decomposition in environments with copious, as well as scarce, on-chip memory resources.

(c) *Efficient processing of the tiles via deep pipelines.* Although the canonical approach presented in Algorithm 1 assumes that we apply one reflector $\mathbf{Q}$ to each incoming vector $\mathbf{a}_j$ per iteration, we apply multiple reflectors³ via deep pipelines in our work. Figure 3 illustrates our approach when four reflectors are applied in a pipeline fashion.

At the top of this figure, we apply reflector $\mathbf{Q}_1$ to all columns of the current tile, one column at a time via a shallow pipeline. The result of the operation is $\mathbf{A}^{(1)} = \mathbf{Q}_1 \mathbf{A}$. In the bottom part, we apply the reflectors $\mathbf{Q}_1, \dots, \mathbf{Q}_4$ to each column of the tile. This operation is illustrated in Equation (6):

$$\mathbf{A}^{(4)} = \mathbf{Q}_4 \mathbf{Q}_3 \mathbf{Q}_2 \mathbf{Q}_1 \mathbf{A} = (\mathbf{I} - \gamma_4 \mathbf{u}_4 \mathbf{u}_4^T)(\mathbf{I} - \gamma_3 \mathbf{u}_3 \mathbf{u}_3^T)(\mathbf{I} - \gamma_2 \mathbf{u}_2 \mathbf{u}_2^T)(\mathbf{I} - \gamma_1 \mathbf{u}_1 \mathbf{u}_1^T) \mathbf{A}. \quad (6)$$

Notice that by applying multiple reflectors for each incoming vector, we can take advantage of the copious resources available on the FPGA, namely **block RAMs (BRAMs)**, **DSPs**, and **lookup tables (LUTs)**. Furthermore, we also use this approach in the processing of the $\mathbf{R}_i$ blocks.

(d) *Efficient processing of the tiles in the $\mathbf{R}_i$ blocks.* As described in the canonical QR decomposition (see Algorithm 1), the QR decomposition has two main steps: the generation of the reflectors, $\mathbf{Q}_1$, and their application. When the inputs to the QR decomposition are the upper triangular matrices $\mathbf{R}_i$, further optimizations [14] for both steps are possible as shown in Figure 4.

At iteration j (see Figure 4(a)), the non-optimized HR decomposition works over tiles of size $((n - j) + n) \times t$. Because the elements below the diagonal in matrices $\mathbf{R}_1$ and $\mathbf{R}_2$ are zero, the computation of the reflector $\mathbf{Q}_j$ can be optimized as shown in the left-most tile in Figure 4(b). The optimized HR decomposition works over tiles of size $(t + (j + t)) \times t$.

³Although technically the term *Householder reflector* refers to the matrix $\mathbf{Q} = \mathbf{I} - \gamma \mathbf{u} \mathbf{u}^T$, in this work we use the word *reflector* to refer to the vector $\mathbf{u}$ as well. The context of the discussion makes it clear whether we are talking about $\mathbf{Q}$ or $\mathbf{u}$.

Fig. 4. Iteration j of the QR decomposition for upper triangular matrices R_1 and R_2 . (a) Non-optimized QR decomposition. (b) Optimized QR decomposition.

Fig. 5. Efficient computation of dot products via a *resource-aware* reduction circuit.

(e) **Efficient computation of dot products.** As shown in Algorithm I, the computation and application of the reflectors, Q_i , requires the efficient computation of dot products. As the decomposition of the tiles in A_i advances, the size of these dot products decreases. The size of the vectors involved in these dot products ranges from $2n$ to $t + n$. Likewise, as the decomposition of the tiles in R_i advances, the size of the vectors ranges from $2t$ to $t + n$. As a result, it is important to implement a flexible circuit that easily adapts to these requirements. To meet these needs, we have implemented a *resource-aware* reduction circuit similar to the one described in the work of Gerards et al. [18]. As shown in Figure 5, this circuit uses two FIFOs, two multiplexers, one adder, one register, and one controller. In addition to being resource aware, this circuit has a latency proportional to the size of the input. Because modern FPGAs have plenty of built-in FIFOs (as well as soft FIFOs), floating-point units, and registers, many of these reduction circuits can be easily implemented in one application.

(f) **Efficient access to the off-chip memory.** By inspection of Algorithm I, we observe that accessing the matrices A_i and R_i is in a column major fashion. Moreover, our benchmarks indicate that in the FPGA development environment [30], accessing off-chip arrays via columns (when the arrays are stored in row major fashion) drastically reduces the performance of the I/O memory

Fig. 7. PE responsible for applying the HRs $\mathbf{Q}_k$ to an incoming vector $\mathbf{a}_j$ such that $\mathbf{t}_j = \mathbf{Q}_k \mathbf{a}_j$. All computations are executed in double-precision floating-point arithmetic using Xilinx DSP cores.

Fig. 8. HR Decomposition engine, which computes the reflectors $\mathbf{Q}_k$ and applies these reflectors to the incoming vectors $\mathbf{a}_j$.

the current set of reflectors while the data memory block stores the tile under decomposition. The *Scheduler* controls the execution of the aforementioned modules and PEs. At the beginning, the memory blocks and the caches are initialized with zero values.

As shown in Figure 2, the tile QR decomposition involves two steps: (1) the computation of the reflectors for the current tile and (2) the application of these reflectors to the remaining tiles. To simplify our discussion, we assume that each tile has $2n \times 8$ -sized elements such that each tile contains eight columns. In the following, we explain the operation of this engine when it executes step one and, subsequently, step two. In the first step, the computation of the reflector $\mathbf{Q}_1$ proceeds as follows:

- (a) The *Scheduler* begins the execution by signaling to the *Reader* to read the first column of the current tile. The off-chip memory responses arrive in an orderly fashion to a FIFO within the *Reader* module.
- (b) The *Reader* makes two copies of the incoming vector: one copy goes to the *Compute-Reflector* PE and the other goes to the *Apply-Reflector I* PE as shown in Figure 8.
- (c) The *Compute-Reflector* PE computes the reflector $\mathbf{Q}_1$ as shown in Figure 6. Moreover, this PE writes $\mathbf{Q}_1$ to the reflector memory block. This memory block stores the vectors $\mathbf{u}_k$ in a dual-port memory one after another. Moreover, it saves the parameters γ_k on registers.
- (d) The *Cache* module is responsible for storing four active reflectors. This module is implemented using four dual-port BRAMS (i.e., one BRAM for each reflector). When the reflector $\mathbf{u}_1$ arrives, this module writes it into a dual-port BRAM. Likewise, the parameter γ_1 is written into a register.
- (e) The *Apply-Reflector I* PE applies the reflector $\mathbf{Q}_1$, arriving from the *Cache*, to the incoming vector $\mathbf{a}_1$, arriving from the *Reader*, as depicted in Figure 7.
- (f) The *Apply-Reflector II, III, and IV* PEs execute the identity operation to the incoming vector. The output of the *Apply-Reflector IV* PE is the vector $\mathbf{Q}_1\mathbf{a}_1$. Moreover, this module writes its output to the data memory block and to a FIFO within the *Writer*.
- (g) The *Writer* writes $\mathbf{Q}_1\mathbf{a}_1$ to the off-chip memory.

At this point, the reflector $\mathbf{Q}_1$ is on cache. Moreover, once the reflector $\mathbf{Q}_1$ is computed, the engine proceeds to compute the vector $\mathbf{Q}_1\mathbf{a}_2$:

- (a) This step is similar to step (a) described earlier, but this time the engine reads the second column of the current tile.
- (b) The *Reader* copies the incoming vector to a FIFO within the *Apply-Reflector I* PE.
- (c) The *Apply-Reflector I* PE computes the vector $\mathbf{Q}_1\mathbf{a}_2$ by using, in addition, the reflector $\mathbf{Q}_1$ in the *Cache* module.
- (d) The *Apply-Reflector II, III, and IV* PEs apply the identity operation over the incoming vector. Next, the output of the *Apply-Reflector IV* PE (i.e., the vector $\mathbf{Q}_1\mathbf{a}_2$) is written to the data memory block as well as the off-chip memory.

The computation of the reflector $\mathbf{Q}_2$ is as follows:

- (a) The *Reader* reads $\mathbf{Q}_1\mathbf{a}_2$ from the data memory block and copies this vector in a FIFO inside the *Compute-Reflector* PE and to a FIFO inside the *Apply-Reflector I* PE.
- (b) The *Compute-Reflector* PE computes the reflector $\mathbf{Q}_2$. Furthermore, it stores this reflector into the reflector memory block and into the *Cache* module.
- (c) In parallel, the *Apply-Reflector I* PE outputs the vector $\mathbf{Q}_1\mathbf{a}_2$ by executing the identity operation.
- (d) The *Apply-Reflector II* PE applies reflector $\mathbf{Q}_2$ to the incoming vector $\mathbf{Q}_1\mathbf{a}_2$. Afterward, the *Apply-Reflector III* and *Apply-Reflector IV* modules execute the identity operation over the incoming vector.
- (e) Finally, the resulting vector $\mathbf{Q}_2\mathbf{Q}_1\mathbf{a}_2$ is written into the data memory block and into the off-chip memory.

At this point, the reflectors $\mathbf{Q}_1$ and $\mathbf{Q}_2$ are available in the *Cache* module as well as the reflector memory block. The computation of the reflectors $\mathbf{Q}_3$ and $\mathbf{Q}_4$ is executed similarly. Once the reflectors $\mathbf{Q}_1$, $\mathbf{Q}_2$, $\mathbf{Q}_3$, and $\mathbf{Q}_4$ are computed, the engine applies these reflectors to the remaining four columns in the current tile. In this process, it reads one column at a time and subsequently

applies these four reflectors via a deep pipeline as shown in Figure 8. Since the current tile has eight columns, one requires the computation of another set of reflectors. This process is as described earlier with the difference that the computation of reflectors starts at column five in the tile. Once these reflectors are computed, the decomposition of the current tile finishes. At this point, the reflector memory block contains eight reflectors, and all results of the decomposition of the first tile are written to the off-chip memory. This concludes step one.

In the second step, the engine makes use of the eight reflectors stored in the reflector memory blocks. It applies those reflectors to the columns of the remaining tiles one tile at a time. In this step, the *Compute-Reflector* PE does not compute reflectors and only reads reflectors from the BRAM. First, the engine applies the saved reflectors to the columns of the second tile. As shown in Figure 8, in this step the engine reads the columns of the second tile one at a time, and for each column, it applies four reflectors via a deep pipeline (see Figure 3). This process is executed in two stages. In the first stage, the first four reflectors are applied to all columns in the second tile, and the results are saved into the data memory block. Next, the engine applies the remaining four reflectors to the saved tile and writes the results to the off-chip memory. At this point, the processing of the second tile is complete. The processing of the remaining tiles is similar to that of the processing of the second tile. Once step two finishes, the engine continues the factorization but this time computes the reflector of the second tile as described in the first step. Next, it applies these reflectors to the remaining tiles as described in step two. This process is executed until the reflectors for all tiles have been computed and applied.

As shown in Figure 1, it is possible to factorize multiple blocks at a time, such as a pair of the A_i blocks or a pair of the R_i blocks. Furthermore, the engine used in this work is able to process either of these pairs. Due to the availability of resources in the target FPGA, we have replicated the proposed engine multiple times as shown in Figure 9. For illustrative purposes, assume that the proposed engine is replicated eight times and the input matrix can be divided into 256 A_i blocks. In this scenario, each engine processes 32 adjacent A_i blocks first, and then each engine processes 16 R_i blocks. Next, each engine processes 8, 4, and finally 2 R_i blocks. At the end of these steps, each engine outputs 1 R_i block. At this point, each engine signals to the host the availability of the R_i block (i.e., 8 blocks in total). Next, the host signals the first four engines to decompose those 8 blocks. Once the working engines finish this step, they signal the availability of their outputs to the host. Later, the host signals the first two engines to decompose these 4 R_i blocks, and this process continues until the first engine finds the final R_i block.

5 EXPERIMENTAL RESULTS

Our experimental work was carried out on the Wolverine II [30] co-processor series. All of our experimental work (*placement*, *routing*, and *execution*) utilized the SB-852VU7P version of the Wolverine II board. We have also *placed* and *routed* our design on the SB-852VU9P version of that board. Table 4 compares these co-processors.

As shown in Table 4, these co-processors are similar, and the main difference is the amount of FPGA resources per board. The first board uses a Xilinx VU7P FPGA, and the second board uses a Xilinx VU9P FPGA. As stated earlier, our experimental testbed consists of an Intel CPU E5-2460 with a SB-852VU7P board [30]. All engines are described in Verilog and were *synthesized*, *placed*, and *routed* with Vivado 17.3 [17]. We addressed all timing errors until the design met the timing requirements of 266 MHz, which was imposed by the Micron Wolverine II board. Our engines took double-precision floating-point matrices as input, and all arithmetic operations were implemented on Xilinx DSP cores [23].

Fig. 9. Replication of the HR decomposition engines.

Table 4. Micron Wolverine II: Co-Processor Comparison

Feature	SB-852VU7P	SB-852VU9P
FPGA	VU7P	VU9P
- Registers	1,576K	2,364K
- LUTs	788K	1,182K
- BRAMs	1,440	2,160
- Block Ultra RAMs	640	960
- DSPs	4,560	6,840
Memory Channels	32	32
Off-chip Memory (DDR4)	64 GB	64 GB
Bandwidth	68 GB/s	68 GB/s
Frequency	266 MHz	266 MHz

Table 5. FPGA Resources per Co-Processor

Resource	SB-852VU7P (Total)	(%) Utilization (10 Engines)	SB-852VU9P (Total)	(%) Utilization (16 Engines)
Registers	1,576K	61.11	2,364K	54.4
LUTs	788K	71.09	1,182K	63.0
LUT RAMs	394K	29.0	591K	24.7
BRAMs	1.4K	65.4	2.2K	55.9
Ultra RAMs	640	18.75	960	20.0
DSPs	4.6K	21.8	6.8K	23.1
Memory Channels	32	62.5	32	100

5.1 FPGA Resources

The decomposition engines were replicated on the FPGA to process as many blocks in parallel as possible. Table 5 shows the resources required by these engines when they were *synthesized, placed, and routed* in each of the two co-processors.

Fig. 10. Execution times versus number of engines.

The SB-852VU7P and SB-852VU9P co-processors can accommodate 10 and 16 engines, respectively. In both cases, we placed the data and reflector memory blocks on the on-chip RAMs. We made extensive use of ultra BRAMs due to their large capacity. Each engine uses one channel for reads and another for writes. By doing so, we prevent stalls in the pipeline due to starvation of data (pending reads) or stalls due to the saturation of the output FIFOs (pending writes). The usage of LUT RAMs is mostly due to the presence of distributed FIFOs to coordinate the execution of operations.

5.2 Execution Times and Efficiency

We generated an $A_{m \times n}$ TSM with $m \gg n$ as shown in Figure 1. Each A_i of size $2n \times n$ (with $m/2n$ being an integer) is a non-singular uniformly distributed matrix.

Figure 10 shows the execution times for the QR decomposition of matrices with 256 columns and 4,096 to 13,1072 rows, as computed on 4 to 16 engines. In this figure, the execution times for 4 and 8 engines were measured on the SB-852VU7P board.⁴ The execution times for 16 engines were a projection based on the specifications of the SB-852VU9P board after *placing* and *routing*.⁵ As shown in this figure, the execution times are inversely proportional to the number of engines. This result is expected because the engines are able to decompose individual blocks independently and the amount of work per block is the same as shown in Table 3. Because the performance of the proposed design is a *linear function* of the number of engines, we only report the performance of the design for 16 engines from this point forward.

We note that the times reported in Figure 10 do not include the time it takes to transfer data to and from the FPGA. Our experiments indicated that for matrices having sizes of $13,1072 \times 256$, $65,536 \times 256$, $32,768 \times 256$, $16,384 \times 256$, and $8,192 \times 256$, the times were 114.0, 57.3, 27.9, 14.49, and 8.60 ms, respectively. In other words, an additional 23% of the execution time was spent moving data from the host to the FPGA and vice versa, and similar time percentages were observed for other experiments. As noted earlier, we did not include this time in our time calculations, and we only report it here for completeness.

⁴Indeed, we furnished our design with an accumulator that counted the number of cycles in the entire factorization.

⁵At the time of this writing, our lab had access to the development tools for both the SB-852VU7P and SB-852VU9P boards [29, 30]. As a result, we can *synthesize*, *place*, and *route* our design in both cards. Moreover, our lab had access to the SB-852VU7P board, and we are actively pursuing the purchase of the newer SB-852VU9P board.

Fig. 11. Execution times (a) and efficiency (b) of the QR decomposition via HRs for 16 engines.

Figure 11(a) shows the execution times when the number of engines is set to 16 while the number of columns increases from 64 to 512 and the number of rows from 4,096 to 13,1072. In this figure, when the number of rows is fixed and the number of columns is increased, the execution time increases quadratically. Conversely, when the number of columns is fixed and the number of rows is increased, the execution time increases linearly.

In Figure 11(b), we plot the efficiency of the engine (i.e., the ratio of the executed FLOPS to the nominal peak performance per clock cycle). In steady state, each reflector executes four simultaneous FLOPS, as shown in Figure 7. As a result, the 16 engines can execute a maximum of $256 = 16(4 \times 4)$ FLOPS per clock cycle. As shown in Figure 11(b), the efficiency of our design is a function of the numbers of columns of the input matrices if the HR engine is in steady state.⁶ Our engine has a maximum efficiency of 54.2% when the input matrices have 512 columns and a minimum efficiency of 28.6% when these matrices have 64 columns. In these cases, the $\mathbf{A}_i$ matrices have sizes of $1,024 \times 512$, whereas the $\mathbf{R}_i$ matrices have sizes of 64×64 , respectively.

5.3 Comparison with CPUs and GPUs

The CPU testbed consists of a workstation equipped with an Intel i7-3370 processor and 8 GB of RAM running the Intel MKL double-precision QR solver [48]. The code was compiled with version 7.4.0 of the gcc compiler. In all of our CPU experiments, four threads were used since the inclusion of additional threads did not improve performance. The GPU testbed consists of a workstation with an Intel E5-520 processor, 24 GB of RAM, and an NVIDIA K40 GPU.⁷ The code was compiled with the CUDA compiler (release 9.0) and the double-precision QR solver from the CUBLAS linear algebra library. The frequency of the GPU was set to 562 MHz, and the auto-boost feature as well as the error correction capabilities (ECC) were disabled.

Table 6 compares the features of the accelerators.

For the FPGA and the GPU, the data was first copied from the host to the accelerator local memory. Next, the QR solver was invoked, either in hardware or software. Finally, the resulting matrix was moved from the accelerator to the host for verification. The time to move the data to and from the accelerator local memory is not included in the execution time. Figure 12 shows

⁶Our internal benchmarks indicate that steady state is achieved when the input matrices are composed of at least a few thousand rows.

⁷We used one of the K40 devices available within the K80 GPU.

Table 6. Comparison of Parameters for the Three Accelerators

Accelerator	Frequency	Peak GFLOPS/s	Cores
Intel i7-3370 CPU	3.4 GHz	108.8	4
NVIDIA K40 GPU	562 MHz	935.0	2496
Micron SB-852VU9P	266 MHz	68.0	16

Fig. 12. Execution times as a function of the number of rows for QR decomposition on FPGAs, GPUs, and CPUs.

the execution times for the target platforms for matrices with 4,096 to 13,1072 rows and 64 to 512 columns.

From this figure, we notice that the FPGA and the library running on the CPU are very fast when the input matrix has 256 columns and less, although the CPU library loses its edge for matrices with 256 columns and more than 64K columns. For the cases of 64, 128, and 256 columns, the FPGA engine has a speedup of 2.0 $\times$, 3.0 $\times$, and 1.3 $\times$ compared with the library running on the GPU. For the case of 512 columns, the routine running on the GPU edges the performance of our proposed FPGA engine. Moreover, the proposed engine and the CPU solver have an equivalent performance for most of the cases, although for very tall matrices with 64K rows and more, the FPGA is faster by a factor of at least 1.5 $\times$.

The performance of the FPGA and GPU can be elucidated by analyzing the pipelines running in the FPGA as well as the QR solver running on the GPU and the CPU. In the FPGA, the QR decomposition of the tiles is divided into two steps as shown in Algorithm I. Because the computation of reflectors is serial (i.e., reflector Q_{i+1} has to be computed after reflector Q_i is available), the performance of the first step is limited by its sequential nature. In addition, the computation of the reflectors does not favor a high performance because the cost of this calculation is dominated by the dot products as shown in Equation (3). Second, the application of the reflectors favors deep (the number of reflectors applied) and wide (the number of running engines)

pipelines; as a result, higher performance is possible. Thus, when the input matrix has a low number of columns, the loss in performance of the first step limits the overall performance of our design. Otherwise, as the number of columns increases, the performance of our design increases as well. Moreover, as shown in Figure 11(b), the efficiency of our design plateaus for inputs with 512 columns.

To analyze the performance of the QR solver running on the GPU, we first performed a detailed complexity analysis of the QR solver described in Algorithm II. Next, we profiled the FLOPS reported by the NVIDIA development kit. Our results indicate that for matrices with 64, 128, 256, and 512 columns, our complexity analysis was very close to the number of FLOPS reported by the NVIDIA API with an accuracy ranging from 90% to 95%. In other words, for matrices with a large number of columns (16,384 rows and more), our complexity analysis is very accurate. However, for matrices with 8192 rows and less, we theorize that the algorithm dynamically changes some of the execution parameters, and as a result, our complexity analysis drops in accuracy by a few percentage points but still captures 90% of the FLOPS reported by the NVIDIA API.

We profiled the number of FLOPS on the CPU as well. Our results indicate that for matrices with 128, 256, and 512 columns, our complexity analysis matches the operations reported by the profiling tool [47] with an accuracy of 97%, 91%, and 90%, respectively. For matrices with 64 columns, our results overestimate the number of FLOPS executed by the CPU by about 7%. By inspecting Algorithm II, we theorize that the differences observed are mostly due to the adjustment of parameters, particularly the block size r , during execution time. As discussed in the following, changing this parameter has broad consequences on the performance of the QR solver. In brief, our experimental work indicates that the QR solver running on the GPU and the CPU most likely uses the well-known routine shown in Algorithm II [20].

For the GPU and CPU, the QR decomposition can also be divided roughly into two major steps, namely the computation of the reflectors (S1, S2, and S3) and their application (S4) as shown in Algorithm II. Since the computation of the reflectors (S1), the application of the reflectors to the current tile (S2), and their transformation (S3) are serial in nature, the first step has limited performance. Moreover, the second step is dominated by matrix products of the form $(\mathbf{I} + \mathbf{Y}\mathbf{W}^T)\mathbf{A}^{(i)}$, and, as a result, greater performance is achieved due to the highly optimized matrix multiplication routines available on the GPU [13, 49]. In addition, for very skinny TSMs, we attribute the rather low performance of the QR solver on GPUs and CPUs to the existence of trade-offs in the implementation of Algorithm II. Regarding this routine, setting the parameter r has broad consequences. If r is small, little progress per iteration is achieved by the algorithm since the resulting number of tiles is large. The large number of tiles implies that steps S1, S2, and S3 have to be executed multiple times. Moreover, step S4 suffers because the multiplication of matrices has to be executed over small matrices [13, 27], namely $\mathbf{W}$ and $\mathbf{Y}$. If r is larger, the performance of step S4 is enhanced at the cost of increasing the execution times of the other steps. On GPUs, larger values of r are not practical due to the limited capacity of shared memory [27].

In brief, the computation of the reflectors is serial based for all of the platforms, and, as a result, the accelerators are not optimal for the execution of this task. However, once the reflectors are available, the application of these reflectors has a better performance on all of the platforms. On the FPGA, the application of these reflectors is dominated by deep pipelines that execute vector operations of the type $\mathbf{a}_j - \gamma(\mathbf{u}^T \mathbf{a}_j)\mathbf{u}$. For the GPU and CPU, the application of the reflectors is dominated by the execution of matrix multiplications and additions of the form $\mathbf{A} + \mathbf{Y}\mathbf{W}^T \mathbf{A}$. For TSMs with 256 columns and less, our design exhibits a high performance since it can efficiently execute these vector operations via wide (16 engines) and deep (four reflectors per engine applied in a pipeline fashion) pipelines. Furthermore, as the number of columns in the input matrix increases, our design maintains its performance, and larger inputs do not result in further gains in

Fig. 13. Double-precision FLOPS per clock cycle for CPUs, GPUs, and FPGAs and their efficiency.

performance. In the case of QR solvers running on the GPU for TSMs with 256 columns and less, the routines are not able to exploit the full potential of the hardware due to the presence of very skinny matrices (i.e., matrices Y and W). In contrast, as the number of columns in the input matrix increases, these libraries achieve a much higher performance since larger inputs favor larger gains in performance.

5.4 Operations per Clock Cycle and Efficiency

We measure the number of FLOPS executed per clock cycle as well as the efficiency of the platforms as shown in Figure 13. In this figure, we only report the FLOPS per clock cycle and efficiency for 65536 rows; these metrics are nearly the same for all of the other experiments. We observe that in both our design and on the GPU, there is a sustained increase in the number of FLOPS per clock cycle as the number of columns in the input matrix increases from 64 to 512. In short, for 512, 256, 128, and 64 columns, the proposed engine executes $1.4 \times$ (13.5), $2.3 \times$ (17.91), $5.0 \times$ (18.0), and $4.4 \times$ (12.44) more FLOPS per cycle compared to the GPU (CPU). In terms of efficiency (i.e., the achieved performance divided by the nominal peak performance), our engine is the most efficient since it achieves 54.2% of the nominal peak performance. The libraries running on the CPU (GPU) achieve 32.1% (5.8%) of the peak performance of the hosting platform.

We note that the proposed design is placed and routed at 266 MHz since the interface to the off-chip memory in the development board is fixed to this frequency. Because we use standard Xilinx cores, namely floating point cores, FIFOS, and BRAMs, we rationalize that our design can be placed and routed at higher frequencies with minimal effort. In particular, memory interfaces running at higher frequencies have been available on the market for some time [21, 42].

5.5 Energy Efficiency

Last, we compare the energy efficiency in FLOPS/Joule for each platform. For the CPU, GPU, and FPGA, we measured the raw power by taking advantage of the LIKWID monitoring tools [47], the NVIDIA management library [34], and the Convey development kit [30], respectively.⁸ For the CPU and GPU, we measured the FLOPS for each task by taking advantage of hardware counters [33, 35]. For the FPGA, we analytically derived the operations executed by the engine. On the GPU,

⁸In the case of the FPGA, we measured the power consumption of 10 engines (the SB-852VU7P), and we then extrapolated the power consumption to 16 engines (the SB-852VU9P).

Fig. 14. Energy efficiency for CPUs, GPUs, and FPGAs.

once the power data is obtained, corrections were made to have an accurate power estimation [7]. Figure 14 shows the energy efficiency per platform, and we only report the energy efficiency when the number of rows is fixed at 65,536. In all cases, the energy efficiency of the FPGA is higher compared to the other platforms. Compared to the GPU, the engine running on the FPGA is 5.4×, 7.7×, 2.8×, and 2.3× more energy efficient when the matrices have 64, 128, 256, and 512 columns, respectively. Compared to the CPU, the proposed engine is 2.0×, 3.0×, 3.0×, and 2.3× more energy efficient for the same task. It should be noted that FPGA technological capabilities continue to increase [41] in both clock frequency and available on-chip resources (memory, DSP, etc.). As such, the expected performance of FPGA-based accelerators is expected to increase even further with the added benefit of energy efficiency [24].

6 CONCLUSION

In this work, we propose a high-throughput FPGA engine capable of executing the QR decomposition of TSMs. Our design is based on the highly stable, parallelizable, and low complexity Householder (HR) decomposition method. The HR engine takes advantage of a series of performance optimizations including tiling, wide and deep pipelines, resource-aware reductions circuits, as well as fast access to off-chip memory. Due to these optimizations, our engine achieves the *highest computational efficiency* compared to previous studies: whereas previous approaches have achieved up to 36% efficiency, our design achieves an efficiency of 54%. Because our design uses *resource-aware* circuits, it can be used to tackle the QR decomposition of the full spectrum of TSMs, including those with hundreds of columns and matrices with hundreds of thousands of rows. Moreover, by tailoring the number of engines in execution, as well as the number of HRs applied, the proposed engine can be implemented in server-grade as well as embedded FPGAs.

Our experimental evaluation shows that the proposed engine outperforms the MKL solver on an Intel quad-core processor by a factor of 1.5× when the input matrices have more than 64K rows. For matrices with up to 256 columns, our engine outperforms the QR solver running on the GPU by a factor of 3.0×. An evaluation of the energy efficiency of these three platforms shows that CPUs and GPUs execute up to 0.45 and 0.60 GFLOPS/Joule, respectively, whereas our design executes up to 1.03 GFLOPS/Joule. This energy efficiency is due to the use of highly efficient *deep* and *wide* pipelines executing more than 100 FLOPS per clock cycle.

REFERENCES

- [1] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. 2011. QR factorization on a multicore node enhanced with multiple GPU accelerators. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 932–943.
- [2] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. 2011. Communication-avoiding QR decomposition for GPUs. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Los Alamitos, CA, 48–58.
- [3] Semih Aslan, Sufeng Niu, and Jafar Saniie. 2012. FPGA implementation of fast QR decomposition based on Givens rotation. In *Proceedings of the 55th IEEE International Midwest Symposium on Circuits and Systems (MWSCAS'12)*. IEEE, Los Alamitos, CA, 470–473.
- [4] Austin R. Benson, David F. Gleich, and James Demmel. 2013. Direct QR factorizations for tall-and-skinny matrices in MapReduce architectures. In *Proceedings of the IEEE International Conference on Big Data*. IEEE, Los Alamitos, CA, 264–272.
- [5] Christian Bischof and Charles Van Loan. 1987. The WY representation for products of Householder matrices. *SIAM Journal on Scientific & Statistical Computing* 8, 1 (1987), s2–s13.
- [6] Akkarat Boonpoonga, Sompop Janyavilas, Phaophak Sirisuk, and Monai Krairiksh. 2010. FPGA implementation of QR decomposition using MGS algorithm. In *Proceedings of the International Symposium on Applied Reconfigurable Computing*. 394–399.
- [7] Martin Burtcher, Ivan Zecena, and Ziliang Zong. 2014. Measuring GPU power with the K20 built-in sensor. In *Proceedings of the ACM Workshop on General Purpose Processing Using GPUs*. ACM, New York, NY, 28–36.
- [8] Emmanuel J Candès, Xiaodong Li, Yi Ma, and John Wright. 2011. Robust principal component analysis? *Journal of the ACM* 58, 3 (2011), 11.
- [9] Abha Chauhan and Rajesh Mehra. 2014. Analysis of QR decomposition for MIMO systems. In *Proceedings of the IEEE International Conference on Electronic Systems, Signal Processing, and Computing Technologies*. IEEE, Los Alamitos, CA, 69–73.
- [10] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. 2008. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proceedings of the IEEE Symposium on Application Specific Processors*. IEEE, Los Alamitos, CA, 101–107.
- [11] Jason Cong, Zhenman Fang, Michael Lo, Hanrui Wang, Jingxian Xu, and Shaochong Zhang. 2018. Understanding performance differences of FPGAs and GPUs. In *Proceedings of the 26th Annual IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*. IEEE, Los Alamitos, CA, 93–96.
- [12] Paul G. Constantine and David F. Gleich. 2011. Tall and skinny QR factorizations in MapReduce architectures. In *Proceedings of the 2nd ACM International Workshop on MapReduce and Its Applications*. ACM, New York, NY, 43–50.
- [13] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. 2010. The Scalable Heterogeneous Computing (SHOC) benchmark suite. In *Proceedings of the 3rd ACM Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, New York, NY, 63–74.
- [14] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 34, 1 (2012), A206–A239.
- [15] Jack Dongarra and Francis Sullivan. 2000. Guest editors' introduction: The top 10 algorithms. *IEEE Computing in Science & Engineering* 2, 1 (2000), 22.
- [16] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. Retrieved February 23, 2021 from <http://archive.ics.uci.edu/ml>.
- [17] Tom Feist. 2012. *Vivado Design Suite*. Technical Report WP416. Xilinx. https://www.xilinx.com/support/documentation/white_papers.
- [18] Marco Gerards, Jan Kuper, André Kokkeler, and Bert Molenkamp. 2009. Streaming reduction circuit. In *Proceedings of the 12th IEEE Euromicro Conference on Digital System Design, Architectures, Methods, and Tools*. IEEE, Los Alamitos, CA, 287–292.
- [19] Kahn Gilles. 1974. The semantics of a simple language for parallel programming. *Information Processing* 74 (1974), 471–475.
- [20] Gene H. Golub and C. F. Van Loan. 2013. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD.
- [21] Zhi Guo, Walid Najjar, Frank Vahid, and Kees Visser. 2004. A quantitative analysis of the speedup factors of FPGAs over processors. In *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM, New York, NY, 162–170.
- [22] Bilel Hadri, Hatem Ltaief, Emmanuel Agullo, and Jack Dongarra. 2010. Tile QR factorization with parallel panel processing for multicore architectures. In *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'10)*. IEEE, Los Alamitos, CA, 1–10.

- [23] Tom Hill. 2009. *Xilinx DSP Design Platforms: Simplifying the Adoption of FPGAs for DSP*. Technical Report WP357. Xilinx. https://www.xilinx.com/support/documentation/white_papers.
- [24] Mark Horowitz. 2014. Computing's energy problem (and what we can do about it). In *Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC'14)*. IEEE, Los Alamitos, CA, 10–14.
- [25] Alston S. Householder. 1958. Unitary triangularization of a nonsymmetric matrix. *Journal of the* 5, 4 (1958), 339–342.
- [26] Andrew Kerr, Dan Campbell, and Mark Richards. 2009. QR decomposition on GPUs. In *Proceedings of the 2nd ACM Workshop on General Purpose Processing on Graphics Processing Units*. ACM, New York, NY, 71–78.
- [27] David B. Kirk and W. Hwu Wen-Mei. 2016. *Programming Massively Parallel Processors: A Hands-On Approach*. Morgan Kaufmann, Waltham, MA.
- [28] Martin Langhammer and Bogdan Pasca. 2018. High-performance QR decomposition for FPGAs. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, 183–188.
- [29] Micron. 2018. Convey PDK Reference Manual. Retrieved September 1, 2018 from <https://www.micron.com/support>.
- [30] Micron. 2020. Micron SB-852 Wolverine II. Retrieved April 04, 2021 from <https://www.micron.com/products/advanced-solutions/advanced-computing-solutions/hpc-single-board-accelerators/sb-852>.
- [31] Sergio D. Muñoz and Javier Hormigo. 2015. High-throughput FPGA implementation of QR decomposition. *IEEE Transactions on Circuits and Systems II* 62, 9 (2015), 861–865.
- [32] NVIDIA. 2018. CUBLAS NVIDIA's Dense Linear Algebra on GPUs. Retrieved June 1, 2018 from <https://developer.nvidia.com/cublas>.
- [33] NVIDIA. 2018. CUDA Toolkit Documentation. Retrieved September 4, 2018 from <https://docs.nvidia.com/cuda/>.
- [34] NVIDIA. 2019. NVML API Reference. Retrieved September 4, 2019 from <https://docs.nvidia.com/deploy/nvml-api>.
- [35] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros, Daniele G. Spampinato, and Markus Püschel. 2014. Applying the roofline model. In *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*. IEEE, Los Alamitos, CA, 76–85.
- [36] Safaa S. Omran and Ahmed K. Abdul-Abbas. 2018. Fast QR decomposition based on FPGA. In *Proceedings of the IEEE International Conference on Advanced Science and Engineering (ICOASE'18)*. IEEE, Los Alamitos, CA, 189–193.
- [37] Michael Parker, Volker Mauer, and Dan Pritsker. 2016. QR decomposition using FPGAs. In *Proceedings of the IEEE National Aerospace and Electronics Conference (NAECON'16) and Ohio Innovation Summit (OIS'16)*. IEEE, Los Alamitos, CA, 416–421.
- [38] Abid Rafique, Nachiket Kapre, and George A. Constantinides. 2012. Enhancing performance of Tall-Skinny QR factorization using FPGAs. In *Proceedings of the 22nd IEEE International Conference on Field Programmable Logic and Applications (FPL'12)*. IEEE, Los Alamitos, CA, 443–450.
- [39] José M. Rodríguez-Borbón, Amin Kalantar, Sharma S. R. C. Yamijala, M. Belén Oviedo, Walid Najjar, and Bryan M. Wong. 2020. Field programmable gate arrays for enhancing the speed and energy efficiency of quantum dynamics simulations. *Journal of Chemical Theory and Computation* 16, 4 (2020), 2085–2098.
- [40] Anatoli Sergiyenko and Oleg Maslennikov. 2001. Implementation of Givens QR-decomposition in FPGA. In *Parallel Processing and Applied Mathematics*. Lecture Notes in Computer Science, Vol. 2328. Springer, 458–465.
- [41] Lesley Shannon, Veronica Cojocar, Cong Nguyen Dao, and Philip H. W. Leong. 2015. Technology scaling in FPGAs: Trends in applications and architectures. In *Proceedings of the 23rd Annual IEEE International Symposium on Field-Programmable Custom Computing Machines*. IEEE, Los Alamitos, CA, 1–8.
- [42] Scott Sirowy and Alessandro Forin. 2008. *Where's the Beef? Why FPGAs Are so Fast*. MSR-TR 2008-130. Microsoft Research, Redmond, WA.
- [43] G. W. Stewart. 2000. The decomposition approach to matrix computation. *IEEE Computing in Science & Engineering* 2, 1 (2000), 50–59.
- [44] Yi-Gang Tai, Kleanthis Psarris, and Chia-Tien Dan Lo. 2011. Synthesizing tiled matrix decomposition on FPGAs. In *Proceedings of the 21st IEEE International Conference on Field Programmable Logic and Applications (FPL'11)*. IEEE, Los Alamitos, CA, 464–469.
- [45] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. 2010. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 36, 5-6 (June 2010), 232–240. <https://doi.org/10.1016/j.parco.2009.12.005>.
- [46] Lloyd N. Trefethen and David Bau III. 1997. *Numerical Linear Algebra*. SIAM, Philadelphia, PA.
- [47] Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 39th IEEE International Conference on Parallel Processing Workshops*. IEEE, Los Alamitos, CA, 207–216.
- [48] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel Math Kernel Library. Retrieved February 23, 2021 from <https://software.intel.com/en-us/mkl>.

- [49] Qian Wang, Xianyi Zhang, Yunquan Zhang, and Qing Yi. 2013. AUGEM: Automatically generate high performance dense linear algebra kernels on x86 CPUs. In *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis*. IEEE, Los Alamitos, CA, 1–12.
- [50] Xinying Wang, Phillip Jones, and Joseph Zambreno. 2014. A reconfigurable architecture for QR decomposition using a hybrid approach. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*. IEEE, Los Alamitos, CA, 541–546.
- [51] Xiaojun Wang and Miriam Leeser. 2009. A truly two-dimensional systolic array FPGA implementation of QR decomposition. *ACM Transactions on Embedded Computing Systems* 9, 1 (2009), 3.
- [52] David S. Watkins. 2004. *Fundamentals of Matrix Computations*. John Wiley & Sons, Danvers, MA.
- [53] Ling Zhuo, Gerald R. Morris, and Viktor K. Prasanna. 2007. High-performance reduction circuits using deeply pipelined operators on FPGAs. *IEEE Transactions on Parallel and Distributed Systems* 18, 10 (2007), 1377–1392.

Received February 2020; revised October 2020; accepted January 2021