ENHANCING PERFORMANCE OF TALL-SKINNY QR FACTORIZATION USING FPGAS

Abid Rafique, Nachiket Kapre and George A. Constantinides

Electrical and Electronic Engineering Department
Imperial College London
London, UK
email: {a.rafique09, n.kapre, g.constantinides}@imperial.ac.uk

ABSTRACT

Communication-avoiding linear algebra algorithms with low communication latency and high memory bandwidth requirements like Tall-Skinny QR factorization (TSQR) are highly appropriate for acceleration using FPGAs. TSQR parallelizes QR factorization of tall-skinny matrices in a divide-and-conquer fashion by decomposing them into sub-matrices, performing local QR factorizations and then merging the intermediate results. As TSQR is a dense linear algebra problem, one would therefore imagine GPU to show better performance. However, the performance of GPU is limited by the memory bandwidth in local QR factorizations and global communication latency in the merge stage. We exploit the shape of the matrix and propose an FPGA-based custom architecture which avoids these bottlenecks by using high-bandwidth on-chip memories for local QR factorizations and by performing the merge stage entirely on-chip to reduce communication latency. We achieve a peak double-precision floating-point performance of 129 GFLOPs on Virtex-6 SX475T. A quantitative comparison of our proposed design with recent QR factorization on FPGAs and GPU shows up to 7.7× and 12.7× speed up respectively. Additionally, we show even higher performance over optimized linear algebra libraries like Intel MKL for multi-cores, CULA for GPUs and MAGMA for hybrid systems.

1. INTRODUCTION

QR factorization is a fundamental problem in linear algebra where an $m \times n$ matrix A is factorized into an $m \times m$ orthogonal matrix Q and an $m \times n$ upper triangular matrix R. Of particular interest is the QR factorization of tall-skinny matrices where $m \gg n$ and the aspect ratio can be 5 to 1, 100 to 1 or in some cases even 100,000 to 1. Matrices with such extreme aspect ratios exist naturally in many practical applications of QR factorization. These include least squares data fitting [1], stationary video background subtraction [2] and importantly s-step Krylov methods [3] and block iterative methods [4] used for solving linear systems and eigenvalue problems. These applications demand high-performance tall-skinny QR factorization. Subtracting the stationary video background from a 10-second surveillance video, for example, requires over a teraflop of computation [2].

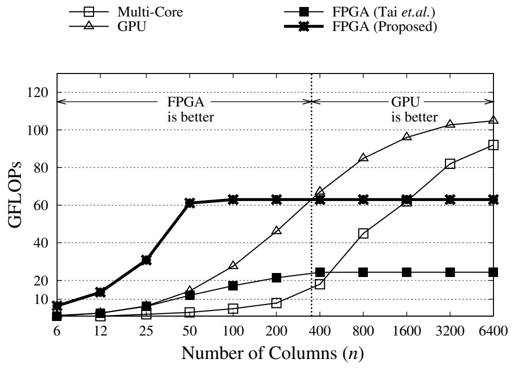


Fig. 1. Performance Scaling Trends for double-precision QR Factorization (No. of rows m = 6400).

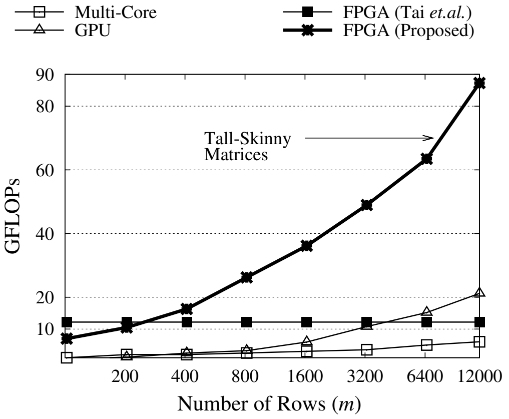


Fig. 2. Performance Scaling Trends for double-precision QR Factorization (No. of columns n = 51).

Traditionally, in high performance linear algebra libraries like LAPACK and Intel MKL, QR factorization is performed using Block Householder QR [5]. Block Householder QR comprises two kernels, a communication-bound *panel fac-*

978-1-4673-2256-0/12/$31.00 ©2012 IEEE

torization with dominant matrix-vector multiplications followed by a compute-bound *trailing matrix update* with dominant matrix-matrix multiplications. While most of the time is spent in trailing matrix update for square matrices, it is the sequential panel factorization which dominates in case of extremely tall-skinny matrices. Recently, Demmel $et.al.$ [6] proposed Tall-Skinny QR (TSQR), a communication-avoiding algorithm for tall-skinny matrices which parallelizes the panel factorization by decomposing it into tiles, performing local QR factorization on tiles using Householder QR and then merging the results (see Section 2.1). The key idea is to do less communication at the cost of more computations because of the ever increasing gap between advancements in compute (GFLOPs) and communication (bandwidth and latency) capabilities of modern architectures.

This paper presents a custom architecture for TSQR which bridges the performance gap that still exists between the parallel potential available in TSQR and that which has been exploited by mapping on variety of parallel architectures like multi-cores [7], Graphical Processing Units (GPUs) [8] and Field Programmable Gate Arrays (FPGAs) [9]. In order to motivate, we show the performance scaling trends for QR factorization as a function of matrix width in Fig. 1. We observe that as the matrix gets skinny, the proposed architecture clearly outperforms the existing implementations and the performance improvement is even more pronounced for extremely tall-skinny matrices as shown in Fig. 2. Although TSQR is a dense problem suited to GPUs, the performance is limited due to memory-bound Basic Linear Algebra Subroutines (BLAS) Level 2 operations in local QR factorizations (see Section 2.2) and the communication latency of the global memory used in the merge stage. We show how we can exploit fine-grain parallelism in BLAS Level 2 operations by using high-bandwidth on-chip memories and deeply-pipelined floating-point cores (see Section 4.3). We additionally perform the merge stage entirely on-chip to avoid communication latency (see Section 4.4).
The key contribution of this paper are thus:

- A high-throughput deeply-pipelined architecture for Householder QR to perform local QR factorizations.
- Mapping of TSQR on the same architecture using pipeline parallelism and avoiding communication latency by merging the intermediate results entirely on-chip.
- Quantitative comparison with optimized linear algebra libraries showing a 21.7× - 50.2× (31.5× geo.mean) over Intel MKL, 7.28× - 36.7× (16× geo.mean) over MAGMA [10] and 52× - 258× (117× geo.mean) over CULA [11].
- Quantitative comparison with a custom architecture for FPGA proposed in [9] and a highly optimized GPU-based implementation in [8] showing a speedup of 0.57× - 7.7× (2.2 × geo.mean) and 3.38× - 12.70× (6.47× geo.mean) respectively.

2. BACKGROUND

2.1. Tall-Skinny QR

The Tall-Skinny QR (TSQR) factorizes the matrix in a divide-and-conquer fashion with optimal communication between processing elements [6]. It comprises a *local QR stage* followed by a *merge stage* as shown in Fig. 3 (a) and 3 (b, c & d) respectively. In the local QR stage, the input matrix A$\in \mathbb{R}^{m\times n}$ is divided into small tiles A$_i \in \mathbb{R}^{b_R\times n}$ where b_R is the number of rows in the tile ($b_R = 2n$ for binary tree) and there are L = $\lceil \frac{m}{b_R} \rceil$ tiles in total. These tiles may then be factorized in parallel using various techniques such as Householder QR [4]. In the merge stage, the R $\in \mathbb{R}^{n\times n}$ factors of local QR factorizations are stacked and factorized in a tree fashion. We get a final R factor in Fig. 3 (d) and a series of small Vs which, if needed, can be used to explicitly compute the orthogonal matrix Q.

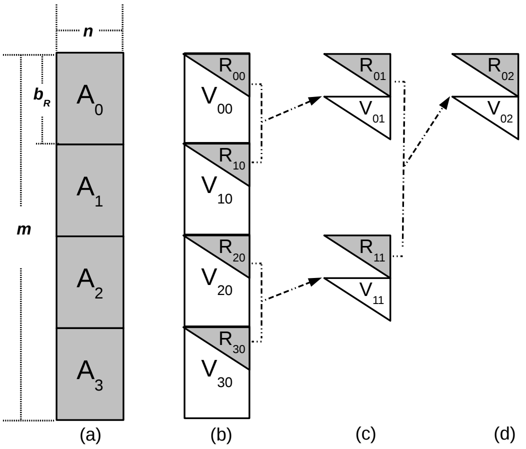


Fig. 3. Tall-Skinny QR Facotorization [6], (a) local QR stage. (b, c & d) merge stage.

The QR factorizations in local QR and merge stages can be performed by various methods including Modified Gram-Schmidt (MGS), Givens rotations, Cholesky QR and Householder QR [4]. We pick Householder QR due to its high numerical stability [6].

2.2. Householder QR

In Householder QR, a matrix A $\in \mathbb{R}^{m\times n}$ is factorized as R = Q$_n \cdots$Q$_2$Q$_1$A where Q = Q$_1^{-1} \cdots$Q$_{n-1}^{-1}$Q$_n^{-1}$ and Q$_k$ = $\begin{pmatrix} I_{k-1} & 0 \\ 0 & H_k \end{pmatrix}$. The matrix H$_k$ is the Householder transformation matrix and is computed as H$_k$ = I$_k$ $- \tau_k$v$_k$v$_k^T$ where v$_k$ is called the *Householder reflector* for column k having length $m{-}k{-}1$. The computational complexity of Householder QR is $O(mn^2)$ and is primarily dominated by BLAS Level 1 and 2 operations as shown in Algorithm 1.

Algorithm 1 Householder QR [4]

```
– Notations –
– A(k:m, j) represents a column vector starting from row $k$ to row $m$ –
– $x_k$ represents $k^{th}$ column vector –
– $x_k(i)$ represents element $i$ in vector $x_k$ –
– $ddot$(x, y, $n$) represents $x^T y$ for vectors having length $n$ –
– $axpy$($\alpha$, x, y, $n$) represents $y \leftarrow \alpha x + y$ with x, y of length $n$ –
Require: A matrix $A \in \mathbb{R}^{m \times n}$
  for $k = 1$ to $n - 1$ do
    – Generate Householder reflector –
    $x_k := A(k : m, k)$                                  (hqr1)
    $d_1 :=$ ddot$(x_k, x_k, m - k - 1)$                   (hqr2)
    $d_2 := \sqrt{d_1} = \|x_k\|_2$                        (hqr3)
    $v_k := x_k$                                          (hqr4)
    $v_k(1) := x_k(1) + \mathrm{sign}(x_k(1)) d_2$         (hqr5)
    $d_3 :=$ ddot$(v_k, v_k, m - k - 1)$                   (hqr6)
    $\tau_k := \frac{-2}{d_3}$                             (hqr7)

    – Update trailing columns of A –
    for $j = k$ to $n$ do
      $y_j := A(k : m, j)$                                (hqr8)
      $d_4 :=$ ddot$(y_j, v_k, m - k - 1)$                 (hqr9)
      $d_5 := \tau_k d_4$                                  (hqr10)
      $y_j^{'} :=$ axpy$(d_5, v_k, y_j, m - k - 1)$        (hqr11)
      $A(j : m, j) : = y_j^{'}$                            (hqr12)
    end for
  end for
  return The upper triangular part of A containing the matrix $R \in \mathbb{R}^{n \times n}$ and matrix
  $V \in \mathbb{R}^{m \times n}$ where individual columns are indexed $v_k$ and a vector $t_k \in \mathbb{R}^{n \times 1}$
  containing $\tau_k$ values.
```

3. RELATED WORK

We survey recent work on tall-skinny QR factorization on parallel architectures like multi-cores [7], GPUs [8] and FPGAs [9]. While the performance of multi-cores is good for square matrices (90 GFLOPs with 58.8% efficiency), it decreases signficantly for tall-skinny matrices (2 GFLOPs with 1.2% efficiency) . This is because of the dominant matrix-vector multiplications in Householder QR (inner loop of Algorithm 1) which are less efficient on multi-cores due to low memory bandwidth. We observe 7× performance improvement with GPU for tall-skinny matrices because they fine-tuned the matrix-vector multiplication by keeping the matrix inside the register file. However, with a limited number of registers per multiprocessor inside GPU, there are fewer threads to saturate the floating-point units. Additionally, the intermediate results are merged using global memory leading to high communication latency. We therefore see an extremely low efficiency for tall-skinny matrices, *e.g.* 2.8% efficiency on Nvidia C2050 for 6400×51 matrix. We discuss this in more detail in Section 6.2. We observe low performance with previous FPGA-based QR factorization as the architecture is optimized for large square matrices. Table 1 summarizes the performance of different implementations with the year, method, GFLOPs and the efficiency for square and tall-skinny matrices. We use 6400×6400 as the square matrix in order to compare against the results reported in [7] and 6400×51 as the tall-skinny matrix since 51 is the maximum number of columns in our design as discussed in Section 5.

Table 1. Comparison of QR Factorization (Square: 6400×6400, Tall-Skinny (TS): 6400×51)

Ref.	Year	Method	Device	Matrix Structure	GFLOPs	Efficiency (% Peak)
[8]	2010	CAQR	Nvidia C2050	Square **TS**	104.7 **14.8**	20.3% **2.8%**
[7]	2010	Tile CAQR	Intel E7340	Square **TS**	90 **2.0**	58.8% **1.2%**
[9]	2011	Tile CAQR	Virtex-6 LX760	Square **TS**	24 **12**	11% **7%**
This Paper	2012	TSQR	Virtex-6 SX475T	**TS**	**62**	**36%**

4. PROPOSED ARCHITECTURE

4.1. Parallelism

4.1.1. Coarse-Grain Parallelism in TSQR

In TSQR, tiles in the local QR stage and within each merge stage can be factorized in parallel as shown in Fig. 3.

4.1.2. Fine-Grain Parallelism in Householder QR

We show the data-flow graph (DFG) of Algorithm 1 in Fig. 4. From the DFG, we observe that the main computationally extensive parts are the BLAS Level 1 $ddot$ operations, *i.e.* dot products $x_k^T x_k$ ($hqr2$), $v_k^T v_k$ ($hqr6$) and $y_j^T v_k$ ($hqr9$) where $k \leq j \leq n$. $ddot$ operation has a sequential latency of $O(n)$ but it can be implemented as a tree-reduction circuit with a $O(\log n)$ latency. Additionally, the inner loop of Algorithm 1 can be fully unrolled to compute $y_j^T v_k$ ($hqr9$) in parallel for different values of j.

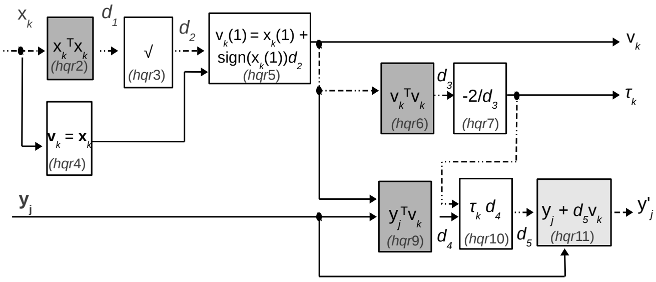


Fig. 4. Householder QR DFG showing the dark grey blocks for $ddot$ and the light grey block for $axpy$. The critical path of the Householder QR is shown as the blocks connected using dotted arrows.

There is another BLAS Level 1 operation $axpy$, *i.e.* multiplication of vector by a scalar followed by a vector by vector addition operation ($hqr11$). $axpy$ has a sequential latency of $O(n)$ but since it is a data-parallel operation it can be fully unrolled to complete with $O(1)$ latency.

4.2. Work vs. Critical Latency

We now explore the gap between the parallelism that is available and the parallelism that can be exploited with limited resources. The latency of the critical path of fully-parallel TSQR is given by Eq. (1) and (2). T_{TSQR} contains a single T_{hqr} term for local QR factorization plus $\lceil\log_2 \mathrm{L}\rceil \times \mathrm{T}_{hqr}$ for all the merge stages (assuming all QR factorizations in local QR stage and within each merge stage are performed in parallel). T_{hqr} represents the latency of Householder QR from Algorithm 1. Referring to Eq. (2), $c_1\lceil\log_2 n\rceil + c_2$ is the latency of the critical path shown in Fig. 4, $\lceil\log_2 n\rceil$ is the latency of *ddot* and c_1, c_2 are constants representing latencies of the double-precision floating-point operators (c_1 = 24 and c_2 = 237 based on latency values from Xilinx Coregen Library). $n-1$ corresponds to the iterations of the outer loop of Algorithm 1.

$$\mathrm{T}_{TSQR} = (\lceil\log_2 \mathrm{L}\rceil + 1)\mathrm{T}_{hqr} = (\lceil\log_2 \frac{m}{2n}\rceil + 1)\mathrm{T}_{hqr} \quad (1)$$

$$\mathrm{T}_{hqr} = (n-1)(c_1\lceil\log_2 n\rceil + c_2). \quad (2)$$

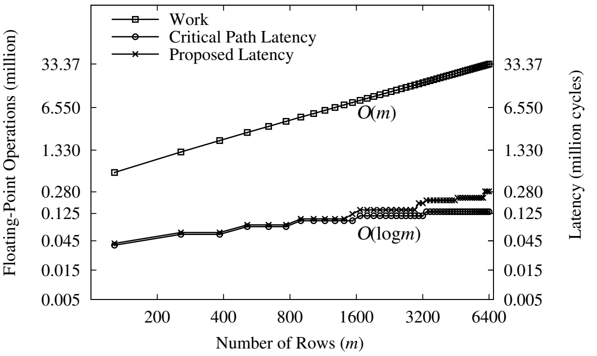


Fig. 5. Work vs. Critical Latency (n = 51).

We show the work ($2mn^2 - \frac{2}{3}n^3$ FLOPs) vs. critical latency of fully-parallel TSQR in Fig. 5. From the figure we observe that there is a considerable gap between latency of sequential and fully-parallel implementation of TSQR *e.g.* a 278× speed up can be achieved for a 6400×51 matrix and this gap increases with the increase in the height of the matrix. We also plot the proposed latency and show that it approaches critical latency of the fully-parallel implementation. We now discuss the parallel architecture which achieves the proposed latency.

4.3. Parallel Architecture for Householder QR

Before introducing our proposed design, we briefly survey a few architectures already presented for Householder QR. Systolic architecture is introduced in [12] for QR factorization in real-time signal processing applications having very small matrices ($n \sim$ a few 10s). Recently, Tai $et.al.$ [9] presented an architecture comprising linear array of processing elements (PEs) for the Householder QR targeting large square matrices. Each PE is responsible for computing an iteration of the outer loop in Algorithm 1. Starting from the first column, first PE of the linear array computes the Householder reflector and then performs the trailing update on the second column. Meanwhile, first PE is performing trailing update on rest of the columns, this second column is then transferred to the second PE for repeating the same calculations. Hence, this architecture only exploits fine-grain parallelism within a single tile without paying attention to the coarse-grain parallelism as discussed in Section 4.1.1. Our architecture is novel in a sense that it exploits both coarse-grain as well as fine-grain parallelism. We now discuss how we exploit the fine-grain parallelism in Householder QR and leave the coarse-grain parallelism until Section 4.4.

The computational complexity of Householder QR is $O(mn^2)$ with an $O(mn)$ memory operations for an $m{\times}n$ input matrix. We design our architecture to exploit this arithmetic intensity. As identified in Section 4.1.2, there are two main computational blocks in Householder QR, $i.e.$ $ddot$ and $axpy$. We take advantage of high on-chip memory bandwidth of the FPGA (∼20 TB/s) and use a deeply-pipelined tree-reduction circuit for *ddot* which is capable of computing a new dot product at every clock cycle. We share the circuit for computing dot products like ($hqr2$), ($hqr6$) and ($hqr9$). We store the matrix A_i in a banked row fashion and feed the vectors x_k or y_j to the dot product circuit during different phases of Algorithm 1. We use a parallel-in parallel-out shift register to store v_k. We completely unroll $axpy$ operation and use an array of multipliers and adders to perform this operation in parallel. The parallel architecture is shown in Fig. 6 where the number of floating-point units grow linearly as given by Eq. (3).

$$\text{Total FP Units} = 8n + 3. \quad (3)$$

We exploit all the fine-grain parallel potential available within Householder QR except that we do not unroll the inner loop of Algorithm 1 due to high memory bandwidth requirements and instead, use pipelining to feed a new dot product operation in each clock cycle to perform $\mathrm{y}_j^T \mathrm{v}_k$ ($hqr9$) for different values of j. The latency of QR factorization of $\mathrm{A}_i{\in}\mathbb{R}^{2n\times n}$ using the proposed architecture is given by Eq. (4).

$$\mathrm{T}'_{hqr} = \frac{n(n+1)}{2} + (n-1)(c_1\lceil\log_2 n\rceil + c_2). \quad (4)$$

Comparing Eq. (2) and (4), we can see that the term $\frac{n(n+1)}{2}$ is introduced due to pipelined implementation of inner loop of Algorithm 1.

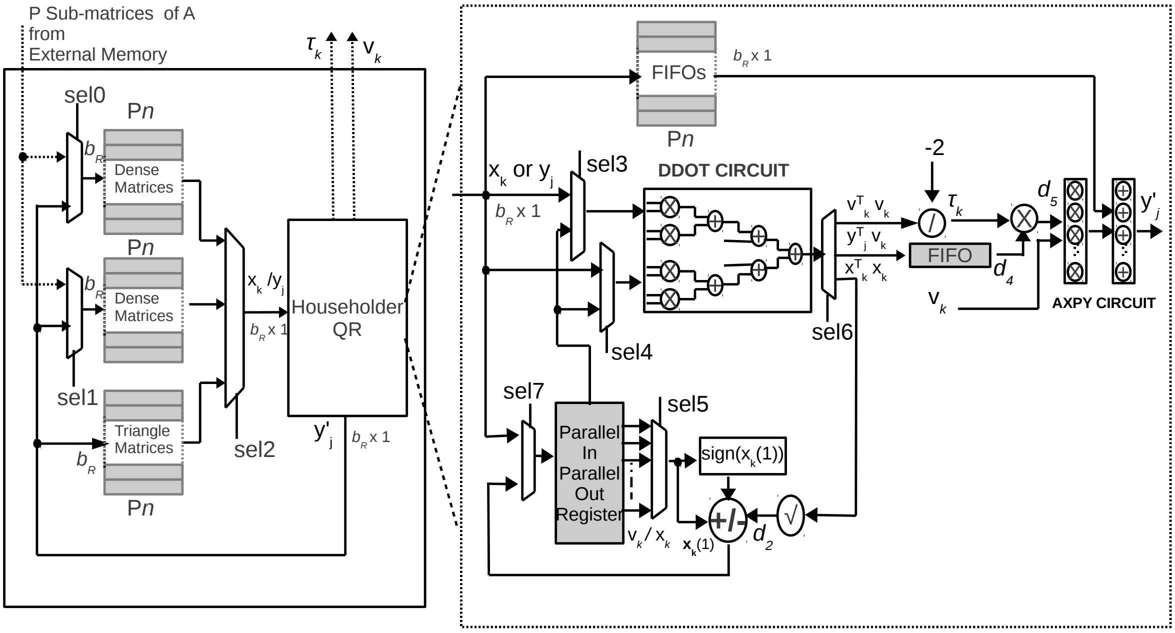


Fig. 6. Parallel Architecture for TSQR, *Dense Matrices* memory stores A_is whereas *Triangle Matrices* memory stores intermediate R factors.

4.4. Pipeline Parallelism for Mapping TSQR

For QR factorization of a single tile, the deeply-pipelined nature of the dot product circuit in Fig. 6 leads to high throughput but also considerable latency. As a result, the pipeline will be underutilized if only single tile is factorized, therefore, we exploit this mismatch between throughput and latency to factorize multiple independent tiles within TSQR. The initiation interval of this circuit is $2(n-1)+\frac{n(n+1)}{2}$ clock cycles (for $hqr2$, $hqr6$ and inner loop of Algorithm 1 containing $hqr9$) after which a new QR factorization can be streamed into this circuit. The pipeline depth (P) of the circuit is given by Eq. (5) which indicates how many QR factorizations can be active in the pipeline at one time.

$$\mathrm{P}(n) \quad = \quad \left\lceil \frac{\frac{n(n+1)}{2} + (n-1)(c_1\lceil \log_2 n\rceil + c_2)}{2(n-1)+\frac{n(n+1)}{2}} \right\rceil . \quad (5)$$

We map the QR factorizations in local QR and merge stages of the TSQR on the same architecture as a set of P Householder QR factorizations as shown in Fig. 7. The intermediate R factors are stored on-chip, therefore, there is no global communication involved in the merge stage. The total latency of TSQR is then calculated as

$$\mathrm{T}'_{TSQR} = \mathrm{T}'_{hqr} \sum_{i=0}^{\lceil \log_2 \mathrm{L} \rceil} \left\lceil \frac{\mathrm{L}}{2^i \mathrm{P}} \right\rceil . \qquad (6)$$

Referring to Eq. (6), each term corresponds to latency of a single TSQR stage shown in Fig. 3.

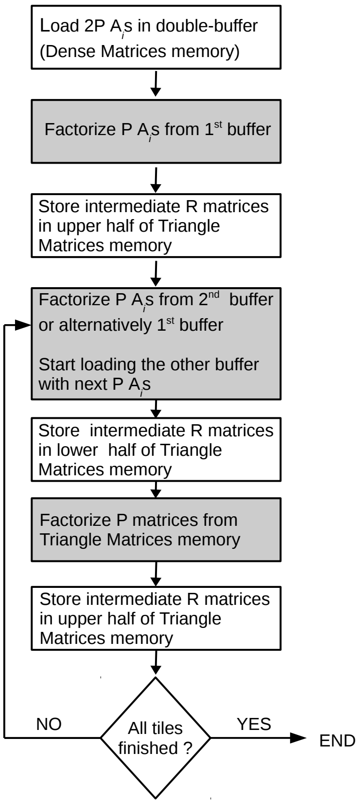

Load 2P A_is in double-buffer (Dense Matrices memory)
Factorize P A_is from 1st buffer
Store intermediate R matrices in upper half of Triangle Matrices memory
Factorize P A_is from 2nd buffer or alternatively 1st buffer
Start loading the other buffer with next P A_is
Store intermediate R matrices in lower half of Triangle Matrices memory
Factorize P matrices from Triangle Matrices memory
Store intermediate R matrices in upper half of Triangle Matrices memory
NO
All tiles finished ?
YES
END

Fig. 7. Mapping of TSQR, A_is $\in \mathbb{R}^{2n\times n}$ whereas R $\in \mathbb{R}^{n\times n}$.

4.5. I/O Considerations

We assume that the matrix is stored in an off-chip memory under supervision of a host CPU just like in the GPU case. We factorize P dense sub-matrices from the double-buffer and P sub-matrices from Triangle Matrices memory before we require a new set of P sub-matrices from the off-chip memory. Therefore, our I/O time to fetch P sub-matrices is double the computation time for QR factorization of P such sub-matrices.

$$\text{I/O bandwidth} \quad = \quad \frac{64\text{P}(3\text{b}_\text{R}n - n^2 + n)}{2\text{P}(2n - 2 + \frac{n(n+1)}{2})} \text{ bits/cycle.} \qquad (7)$$

Referring to Eq. (7), we require 64P($3b_Rn - n^2 + n$) bits to be exchanged between the FPGA and the off-chip memory where the b_Rn term comes from the size of input sub-matrix, $2b_Rn - n^2 + n$ from the v_k vectors and τ_k values for explicit formation of Q matrix. The latency of factorizing P sub-matrices after matching the pipeline depth is P($2n - 2$ $+ \frac{n(n+1)}{2}$) cycles and it is twice of this time that is available to load new set of P sub-matrices. Given the maximum value of b_R ($2n$ for binary tree) to be 102 resulting into maximum value of n = 51 (see Section 5), we find the I/O requirement to be 11.53 GB/sec (∼30% of Virtex-6 SX475T based MAX3 card from Maxeler Technologies [13]).

5. EVALUATION METHODOLOGY

The experimental setup for performance evaluation is summarized in Table 2. We use GFLOPs as the metric for comparing performance on different architectures. We use highly optimized linear algebra libraries for GPU (CULA), multi-cores (Intel MKL) and hybrid systems (MAGMA). We implement the proposed architecture in VHDL using double-precision floating-point cores and synthesize the circuit for Xilinx Virtex-6 SX475T FPGA, a device with the largest number of DSP48Es and a large on-chip capacity. The placed and routed design has an operating frequency of 315 MHz. We find out the maximum value of b_R to be 102 before 90% of the Slice LUTs, 96% of DSP48Es and 77% of BRAMs are utilized. Although, m can take on any value only limited by off-chip memory, b_R limits the maximum value of n because b_R should be greater than or equal to n (in our design b_R = $2n$ for binary tree). The value of n is appropriate for tall-skinny QR applications where it is on the order of a few 10s. The proposed design needs to be re-synthesized only if the input matrix size changes in the column dimension (n).

6. RESULTS

We now present the performance achieved by our FPGA design, compare it with optimized QR routines from linear algebra libraries and the state of the art in FPGAs, multi-cores and GPU and then discuss the underlying factors that explain our results.

Table 2. Experimental Setup.

Platform	**Peak GFLOPs** Double-Precision	**Compiler**	**Libraries**	**Timing**
Intel Xeon X5650 **(32 nm)**	63.9 [14]	gcc (4.4.3(-O3))	Intel MKL (10.2.4.032)	PAPI (4.1.1.0)
Nvidia GPU C2050 **(40 nm)**	515	nvcc	CULA-R11 MAGMA-rc5	cudaEvent-Record()
Virtex-6 SX475T **(40 nm)**	171 [15]	Xilinx ISE (10.1)	Xilinx Coregen	ModelSim

6.1. FPGA Performance Evaluation

The peak and sustained double-precision floating-point performance of the FPGA is given by Eq. (8) and Eq. (9) respectively.

$$\text{Peak Throughput} \quad = \quad 8n + 3 \quad \text{FLOPs/cycle.} \qquad (8)$$

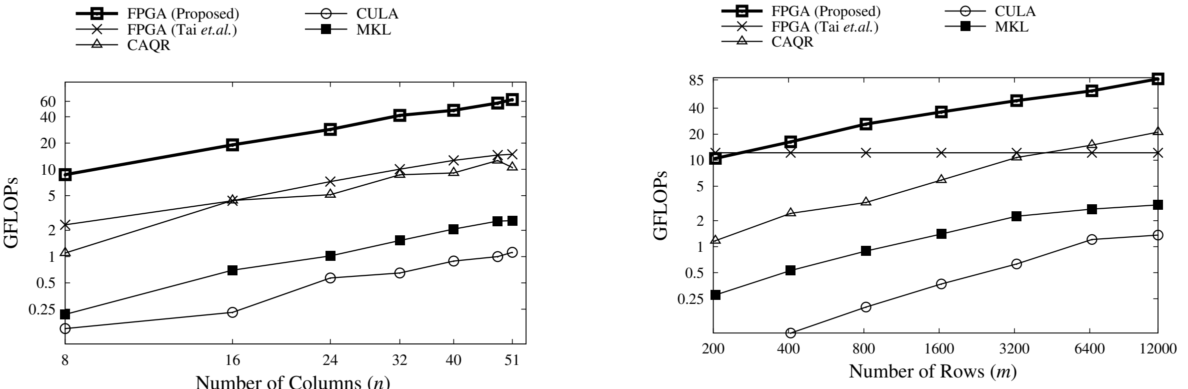
$$\text{Sustained Throughput} \quad = \quad \frac{2mn^2 - \frac{2}{3}n^3}{\text{T}'_{TSQR}}\text{FLOPs/cycle} \qquad (9)$$

where $8n$ + 3 is the total number of floating-point units in the proposed design and $2mn^2 - \frac{2}{3}n^3$ represents FLOPs in TSQR. For an operating frequency of 315 MHz, the peak performance of our design for maximum value of $n = 51$ is 129 GFLOPs and it is observed that the efficiency of the proposed architecture is greater than 80% for tall-skinny matrices.

6.2. Comparison with GPU

We compare our work against a custom implementation of QR factorization using communication-avoiding QR (CAQR) algorithm on Nvidia C2050 Fermi [8]. In this implementation, the matrix is divided into panels which are factorized using TSQR (*hh_factor_dense* and *hh_factor_triangle*) and then the trailing matrix update (*hh_update_dense* and *hh_update_triangle*) is performed. We use Compute Visual Profiler [16] to profile GPU code for a range of extremely tall-skinny to square matrices as shown in Fig. 9. We observe that TSQR dominates in case of extremely tall-skinny matrices (∼70% of runtime contribution).

Firstly, we compare the arithmetic intensity in TSQR in GPU with our design. In GPU, the Householder QR factorizations are performed by keeping the tiles inside the register file due to its high access bandwidth (∼ 8 TB/s). The panel width is tuned to a value of 8 and a tile size of 64×8 is chosen such that it can fit into the register file of each stream multiprocessor (SM) for best performance. This restricts the arithmetic intensity in Householder QR as there are $O(mn^2)$ FLOPs for $O(mn)$ memory operations. The width of the panel (n) can be increased at the expense of



(a) GFLOPs vs. n (m = 6400)

(b) GFLOPs vs. m (n = 51)

Fig. 8. Performance Comparison with Multi-Cores (Intel MKL), GPUs (CULA, CAQR), and best FPGA work.

height (m) but it then increases the number of stages in the merge stage of TSQR. In our design, the panel width can be at most 51, a 49× increase in arithmetic intensity compared to GPU. It not only reduces the number of merge stages in TSQR but also the number of panels to be factorized, $e.g.$ for a matrix of size 6400×51, there will be seven panels which need to be factorized using TSQR in GPU whereas in our design only single TSQR factorization is required.

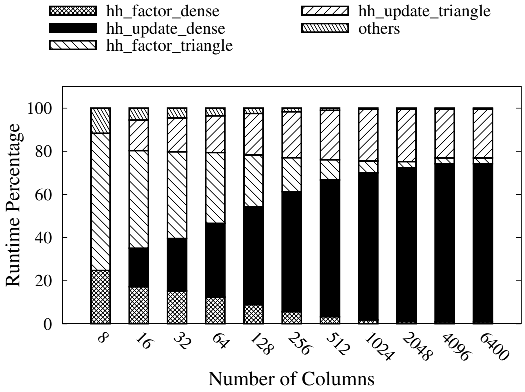


Fig. 9. GPU Performance Analysis (Number of Rows = 6400).

Secondly, we compare the type of memory and its bandwidth used in local QR and merge stages of TSQR for both cases. In GPU, during local QR stage, tiles are loaded into register file (each per SM) from global memory. The tiles are then factorized using register file as well as shared memory and the local R factors are then stored back in global memory. In the merge stage, the R factors are loaded back into register file from distributed locations of global memory and are then factorized. In our design, however, we perform both local QR and merge stages using on-chip memory to minimize memory latency. Table 3 lists the type of memories used in each stage for GPU as well as FPGA.

Table 3. Type of Memory and its peak bandwidth for GPU and proposed FPGA design in different stages of TSQR.

Kernel	GPU Memory			FPGA Memory	
	Register (∼8 TB/s)	Shared (∼1.3 TB/s)	Global (144 GB/s)	Register (∼37 TB/s)	BRAMs (∼20 TB/s)
`local QR`	√	√		√	√
`merge stage`	√	√	√	√	√

Lastly, we compare the utilization of floating-point units in GPU and our custom architecture. We identify the limiting factors in the kernels used for TSQR in GPU as shown in Table 4. Both the kernels perform QR factorization by a thread block having 64 threads and there are 8 thread blocks used per SM. Each thread has 63 registers and therefore maximum number of threads is limited and as a result occupancy is low. It is due to this low occupancy ratio particularly in $hh\_factor\_triangle$ kernel that we get very low performance for tall-skinny matrices. On the other hand, we get an almost 80% of peak performance (129 GFLOPs) for extremely tall-skinny matrices (see Section 6.1). As a result, we get a speed up of 3.38× - 12.70× (6.47× geo.mean) shown in Fig. 8(a) and 8(b).

Table 4. Limiting Factors for Tall-Skinny Matrices (6400×51) on GPU. A $warp$ comprises 32 threads and there are 48 active warps per cycle for an occupancy of 1.

Kernel	Active Warps / Active Cycles	Occupancy Ratio	Limiting Factor
`hh_factor_dense`	9.05	0.18	Registers
`hh_factor_triangle`	2.98	0.06	Registers

We also compare QR routines from CULA as well which uses Block Householder QR with dominant sequential panel factorization for tall skinny matrices. We show a speed up of 52× - 258× (117× geo.mean) due to parallel TSQR algorithm and its efficient FPGA implementation.

6.3. Comparison with Multi-Cores and Related FPGA Work

We now compare our design with QR routines from MKL and MAGMA which perform panel factorization in a sequential fashion on multi-cores. We observe a speed up of 21.7× - 50.2× (31.5× geo.mean) and 7.28× - 36.7× (16× geo.mean) over MKL and MAGMA respectively. We finally compare our design with the most recent work on QR factorization on FPGAs by Tai $et.al.$ [9]. In [9], the architecture is tuned for large square tiles and therefore we see increase in performance with increase in the width of the matrix as shown in Fig. 8(a). However, since no parallelism is exploited along the row dimension, therefore, we see constant performance as we increase the height of the matrix shown in Fig. 8(b). We, therefore, observe a speed up of 0.57× - 7.7× (2.2 × geo.mean) across a range of matrix sizes.

7. CONCLUSION

We show that customizing an architecture to the shape of the matrix for TSQR can give us up to 7.7× and 12.7× speed up over state of the start in FPGAs and GPUs respectively. We highlight the low efficiency of GPUs and multi-cores as percentage of their peak theoretical performance for QR factorization of tall-skinny matrices. We identify that on GPUs it is due to low arithmetic intensity caused by limited registers, low occupancy during the merge stage, and global communication during the merge stage where as in case of multi-cores it is primarily due to low memory bandwidth. We show how we can exploit the high on-chip bandwidth of the FPGA to design a high-throughput architecture for QR factorization and large on-chip capacity to perform the merge stage without any global communication. We conclude that even though GPU has 3× higher peak double-precision floating-point performance, by exploiting the architectural features of FPGAs we can outperform the GPU for communication-avoiding linear algebra algorithms like TSQR.

8. ACKNOWLEDGEMENT

We would like to thank Michael Anderson, PAR Lab, University of California, Berkeley for providing us the GPU source code for QR factorization [8]. Dr. George A. Constantinides would like to acknowledge the support of EPSRC (EP/I020357/1 & EP/G031576/1) and Dr. Nachiket Kapre would like to thank the Imperial College Junior Research Fellowship Scheme.

9. REFERENCES

[1] Å. Björck, *Numerical methods for least squares problems*. Society for Industrial Mathematics, 1996, no. 51.

[2] E. Candes, X. Li, Y. Ma, and J. Wright, "Robust Principal Component Analysis?" *Arxiv preprint ArXiv:0912.3599*, 2009.

[3] M. Hoemmen, "Communication-avoiding Krylov subspace methods," Ph.D. dissertation, University of California, 2010.

[4] G. Golub and C. Van Loan, *Matrix Computations*. Johns Hopkins University Press, 1996, vol. 3.

[5] R. Schreiber and C. Van Loan, "A storage-efficient WY representation for products of Householder transformations." *SIAM J. Sci. Stat. Comput.*, vol. 10, no. 1, pp. 53–57, 1989.

[6] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, "Communication-optimal parallel and sequential QR and LU factorizations," *Arxiv preprint arXiv:0808.2664*, 2008.

[7] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra, "Enhancing Parallelism of Tile QR Factorization for Multicore Architectures," *Matrix*, vol. 2, no. 4, p. 8, 2010.

[8] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, "Communication-avoiding QR decomposition for GPUs," in *Proc. IEEE Int. Symp. Parallel & Distributed Processing (IPDPS)*, 2011, pp. 48–58.

[9] Y. Tai, K. Psarris, and C. Lo, "Synthesizing Tiled Matrix Decomposition on FPGAs," in *Proc. IEEE Int. Conf. Field Programmable Logic and Applications (FPL)*, 2011, pp. 464–469.

[10] "MAGMA, Matrix Algebra on GPU and Multicore Architectures," 2011. [Online]. Available: http://icl.cs.utk.edu/magma/

[11] J. Humphrey, D. Price, K. Spagnoli, A. Paolini, and E. Kelmelis, "CULA: hybrid GPU accelerated linear algebra routines," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, vol. 7705, 2010. [Online]. Available: http://www.culatools.com/

[12] K. Liu, S. Hsieh, and K. Yao, "Systolic block Householder transformation for RLS algorithm with two-level pipelined implementation," *Signal Processing, IEEE Transactions on*, vol. 40, no. 4, pp. 946–958, 1992.

[13] "MAX3 Card." [Online]. Available: http://www.maxeler.com/content/hardware/

[14] "Intel microprocessor export compliance metrics," 2010. [Online]. Available: http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf

[15] "High Performance Computing using FPGAs," 2010. [Online]. Available: www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf

[16] "Compute Visual Profiler," 2010. [Online]. Available: http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf