High-performance QR Decomposition for FPGAs

Martin Langhammer, Bogdan Pasca
Intel, Programmable Solutions Group

***Abstract*—QR decomposition (QRD) is of increasing importance for many current applications, such as wireless and radar. Data dependencies in known algorithms and approaches, combined with the data access patterns used in many of these methods, restrict the achievable performance in software programmable targets. Some FPGA architectures now incorporate hard floating-point (HFP) resources, and in combination with distributed memories, as well as the flexibility of internal connectivity, can support high-performance matrix arithmetic. In this work, we present the mapping to parallel structures with inter-vector connectivity of a new QRD algorithm. Based on a Modified Gram-Schmidt (MGS) algorithm, this new algorithm has a different loop organization, but the dependent functional sequences are unchanged, so error analysis and numerical stability are unaffected. This work has a theoretical sustained-to-peak performance close to 100% for large matrices, which is roughly three times the functional density of the previously best known implementations. Mapped to an Intel Arria 10 device, we achieve 80us for a 256x256 single precision real matrix, for a 417 GFLOP equivalent. This corresponds to a 95% sustained to peak ratio, for the portion of the device used for this work.**

I. Introduction

The nature of FPGA architecture is changing. Originally, devices contained soft logic and programmable interconnect, with the majority of the die area devoted to programmability. Embedded functions started being added, first memory blocks, followed by hard multipliers, and more recently, floating-point support. The embedded features have also changed the power envelope of the FPGA; on one hand the hardened features allow for much greater functionality, measured either per mm^2, or by functional density. On the other hand, the greater power density in the ever increasing embedded features has left less thermal capacity for the traditional functionality within the typical FPGA package and for environmental constraints.

Contemporary applications therefore need to rely less on the traditional soft logic for datapath construction and instead utilize the embedded features. Soft logic will only be used for assembling datapaths, as well as supplying a resource for bit-level operations such as state machines and control logic. Software programmable solutions, such as CPUs and GPUs, are limited by the data transfer paths of a load-store architecture. In contrast, FPGAs are hardware programmable solutions, with flexible and configurable data movements. In this paper, we will show that we can take advantage of multiple degrees of freedom in data movement: parallelism as well as independence in data sources and destinations. For instance, data can be written back to memory and simultaneously to another datapath, all without instruction overhead.

We select MGS for the QRD implementation, as it lends itself readily to be parallelized, as all data accesses are column-wise. In contrast, Householder QRD requires alternating row and column accesses. Although Householder QRD exhibits better stability than MGS, the MGS implementations described in this paper reduce cumulative rounding errors.

In this paper, we will use sustained to peak performance as our quantitative measure, which can clearly show computational and algorithm efficiency. This can also scale well for comparisons with future FPGA and other architectures, where the resource mix and/or speed changes. As a qualitative measure, we will also examine resource use - our goal is to improve efficiency by using the smallest possible amount of soft logic, and map as much of the design to memories and DSP Blocks.

II. Previous Work

Typically, FPGA implementations of QRD have focused on small matrices, with relatively low throughputs. Often, logic intensive approaches such as CORDIC have been employed, even in recent works. In [1], a 4x4 16-bit fixed-point QRD is implemented using an unrolled CORDIC approach, requiring 2671 4LUTs and 12 DSP48E of a Xilinx Virtex5 device, with a clock frequency of 254MHz. The first floating-point (IEEE-754 single-precision) FPGA QRD was in [2], where a 7x7 systolic array was implemented. This design required 126K 4LUTs, 102 DSP48E Blocks, and 56 BRAMs of a Xilinx Virtex5 device, achieving a clock rate of 132MHz. Neither publication explicitly specifies whether the number format is real or complex, although [3] describes a 4x4 complex QRD using the SGR (Squared Givens Rotation) method on a custom floating-point format (6-bit exponent, 14-bit fraction) in an earlier work.

In [4], a vectorized algorithm was introduced, which was able to calculate a much larger complex QRD on an FPGA efficiently. In [5] this QRD was mapped onto an Arria 10 device to take advantage of the floating-point DSP Blocks; a 200x100 QRD required 423 DSP Blocks and 12K ALMs, or about 25% of the device. A 78 GFLOP performance was calculated, which corresponds to 30% sustained to peak of the consumed resources, at a 300MHz clock rate. From [4], the QRD requires about 45% of the FLOPs of a simple STAP radar processing core. As the QRD is a n^3 process, and the other key signal processing components (such as generating the covariance matrices, finding the interference covariance matrix, and calculating the steering vectors using back-substitution) are n^2, the QRD will increase as a fraction of the computational load with increasing system complexity.

Another recent MGS floating-point FPGA implementation is described in [6]. Although poorly documented, we calculate

that the presented 128x64 matrix requires 1.4GFLOPs at the stated throughput. From [7], the number of DSP48E blocks can support 37.6 GFLOPs at the stated frequency, which corresponds to a 3.7% sustained to peak ratio. Interestingly, we will see that this is very close to the expected hardware implementation of the unoptimized canonic MGS.

In [8], the GPU is evaluated for radar signal processing applications, with particular emphasis on QRD, and higher level algorithms (STAP) containing QRD. The work explains that matrix sizes of 200x100 are representative of the complexity for this type of application. The QRD on individual matrices has a very low performance of a fraction of 1% sustained to peak, although a STAP system using multiple parallel iterations of QRD approached 23% sustained to peak; however, it must be noted that only the R portion of the result was calculated in this case. In [9], GPU results for QRD showed 15% sustained to peak performance for matrices of size 8192x8192, and 25% for larger matrices. Smaller matrices were not reported, but the steep slope in GFLOPs from the 8K to 9K sizes suggests that smaller matrices performed poorly. Kerr [10] also reports a 8K matrix QRD throughput of about 15%, but smaller matrices appear to have a much lower throughput. These three sources correlate closely; a QRD algorithm implemented on a GPU may achieve a sustained to peak of 15% for larger matrices, or carefully organized smaller matrices, but will show dramatically lower performance for typical matrix sizes used in embedded applications.

III. Algorithm

We will first review the canonic MGS algorithm in order to highlight the importance of the sequence of operations on expected throughput. We will then show the best known algorithm for parallel datapaths, which approaches 50% sustained to peak throughput, and confirm that although the grouping of operations is different than the canonic MGS, the order of dependent operations is essentially the same. Finally, the new MGS algorithm, which trends towards 100% sustained to peak, will be introduced. The grouping of operations is again changed, but the order of dependent operations is the same.

The input matrix is A, a $n \times n$-element square matrix formed out of the column vectors $a_k, k \in \{1..n\}$:

$$A = [a_1|a_2|...|a_n].$$

We denote by $\langle x,y \rangle$ the dot-product $x^T y$, and by $|x|$ the norm of the column vector x. The L2-norm used in this article is:

$$|x| = |(x_1,...,x_n)^T| = \sqrt{x_1^2 + ... + x_n^2}$$

For the purposes of illustration, we will use a 256×256 matrix as the basis for our discussion. This will allow us to show the effects of realistic datapath and operator latencies for current FPGAs in the context of the QRD. In this paper, we will use the following terminology: a *vector operation* denotes a dot-product and a *scalar operation* denotes a multiply-add or multiply-subtract.

Algorithm 1: Gram-Schmidt–based QR factorization

```
1 function QRD (A);
  Input  : Matrix A (square)
  Output: Matrix Q (orthogonal) and R (upper-triangular)
2 for i=1:n do
3     r_{i,i} = |a_i|
4     q_i = a_i / r_{i,i}
5     for j=i+1:n do
6         r_{i,j} = <q_i, a_j>
7         a_j = a_j - r_{i,j} q_i
```

A. Canonic MGS Algorithm

The canonic MGS algorithm is shown in Algorithm 1. There are n outer loop iterations $i \in \{1..n\}$ - in this case, 256. At the beginning of each iteration ($k \in \{1..n\}$), the norm of the column vector a_k, is calculated. This can be computed as the dot-product $\langle a_k, a_k \rangle$, followed by a square root. Both the dot-product and the square root are long latency operations.

In an FPGA with HFP cores, a 256-length dot-product may require 25 or more cycles (depending on the target frequency), with an additional 10 to 20 cycles for the square root, which in practice is often replaced by the lower latency inverse square root and a multiply. This creates a data dependency for the following divide function, which creates a 45-cycle stall before the following column can be processed.

The following columns $j \in \{i+1..n\}$ are processed by the inner loop. The first operation is a dot-product, followed by a multiply-subtract. The dot-product again has a relatively long latency, and creates a data dependency for the multiply-subtract. In contrast, the multiply-subtract has a relatively short latency, of typically 5 clock cycles.

Looking at the number of clock cycles required for a single iteration for mid-algorithm ($i = 128$), the normalization calculation requires around 45 cycles, and the following 127 inner loop iterations require 30 cycles each. Clearly, this is very inefficient, with the sustained to peak ratio in the region of 3%, based on the amount of time stalled, however, the true cost can be much higher, if the scalar and vector structures are on independent datapaths.

B. Improved MGS Algorithm

An improved form of the MGS algorithm is presented in [4] and rewritten as Algorithm 2. The algorithm loops and operations are restructured in order to reduce and hide loop-carried dependencies.

Examining the multiply-add operation $a_j = a_j - r_{i,j} q_i$ (line 7 in Algorithm 1) we can substitute the definitions of $r_{i,j}$ and q_i to make an equivalent calculation:

$$a_j = a_j - \frac{\langle a_i, a_j \rangle}{|a_i|} \frac{a_i}{|a_i|} = a_j - \frac{\langle a_i, a_j \rangle}{\langle a_i, a_i \rangle} a_i$$

The new update equation for a_j removes the $r_{i,j}$ and $q_{i,j}$ computations from the critical path and requires computing the scalar division $rn_{i,j}/r2_{i,i}$. The $rn_{i,j}$ elements are processed

Algorithm 2: QRD improved algorithm (from [4])

```
1  function QRD (A);
   Input  : Matrix A (square)
   Output: Matrix Q (orthogonal) and R (upper-triangular)
2  for i=1:n do
3      r2_{i,i} = ⟨a_i, a_j⟩
4      for j=i+1:n do
5          rn_{i,j} = ⟨a_i, a_j⟩
6      r_{i,i} = sqrt(r2_{i,i})
7      q_i = a_i / r_{i,i}
8      for j=i+1:n do
9          a_j = a_j − (rn_{i,j} / r2_{i,i}) a_i
10     for j=i+1:n do
11         r_{i,j} = rn_{i,j} / r_{i,i}
```

in a separate loop (line 5 in Algorithm 2) to that containing the a_j updates. As these elements become available at the output of the dot-product unit, they feed directly into a second parallel compute core. This core computes the quotients $rn_{i,j}/r2_{i,i}$ and square-roots $\sqrt{r2_{i,i}}$. When all the a_i, a_j inputs are fed into the dot-product unit, and as the first quotient becomes available, the main compute cores will start processing the a_j updates. After all a_j are updated we can proceed to the next outer loop iteration.

The vector calculation (dot-product and the normalization) requires different hardware than the multiply-subtract calculation. As all of the vectors for the r calculation must have been read before the multiply-add begins, typically one structure will be mostly idle while the other is being utilized. This results in a maximum sustained-to-peak performance of roughly 50%.

The architecture proposed in [4] requires parallel transfers between memories and the datapath cores – the dot-product and mult-subtract units. The potentially distant geographical fanout from the memory blocks may limit timing closure and requires more elaborate pipelining.

C. Proposed MGS Algorithm

In [11] we presented a new loop structure of the MGS algorithm where the scalar datapath feeds the vector datapath. This makes near 100% sustained to peak performance possible, as both datapaths are active at the same time. Additionally, the memory fanout is reduced to 1 as memories connect directly to the scalar datapath, and the scalar datapath feeds the vector datapath in turn.

Algorithm 3 updates [11] with operation mapping considerations. The first part (lines 2-6) still requires the vector datapath, which is used to calculate the normalization ($ir_{1,1}$) and $s_{1,j}$ values for the first pass, using the optimizations of the improved MGS algorithm described previously. The main loop (lines 9-19) calculates q_i and all of the a_j updates (the multiply-subtract values), and writes these back to memory. At

Algorithm 3: Proposed QRD algorithm

```
1  function QRD (A);
   Input  : Matrix A (square)
   Output: Matrix Q (orthogonal) and R (upper-triangular)
2  p_{1,1} = ⟨a_1, a_1⟩
3  ir_{1,1} = 1 / sqrt(p_{1,1})   /* via the reciprocal square-root */
4  for j=2:n do
5      p_{1,j} = ⟨a_1, a_j⟩
6      s_{1,j} = p_{1,j} / p_{1,1}   /* via the divider              */
7      r_{1,j} = p_{1,j} × ir_{1,1}  /* via the multiplier           */
8  for i=1:n-1 do
9      q_i = a_i × ir_{i,i}
10     for j=i+1:n do
11         a_j = a_j − s_{i,j} a_i
12         if j=i+1 then
13             p_{i+1,i+1} = ⟨a_{i+1}, a_{i+1}⟩
14             ir_{i+1,i+1} = 1 / sqrt(p_{i+1,i+1})
15             r_{i+1,i+1} = sqrt(p_{i+1,i+1})
16         else
17             p_{i+1,j} = ⟨a_{i+1}, a_j⟩
18             s_{i+1,j} = p_{i+1,j} / p_{i+1,i+1}
19             r_{i+1,j} = p_{i+1,j} × ir_{i+1,i+1}
20 q_n = a_n × ir_{n,n}
```

the same time, the $ir_{i+1,i+1}$ and the $s_{i+1,j}$ values for the next loop iteration are calculated from the a_j just computed; the multiply-subtract datapath feeds both the memory write port, as well as the vector datapath, so that the scalar and vector datapaths are utilized simultaneously. The first ir in each loop iteration (line 14) is the normalization value, and is calculated by the inverse square root of the dot-product of the first vector written back. For subsequent js the $s_{i+1,j}$ values (line 18) are the quotients resulting from dividing the dot-products $p_{i+1,j}$ by the pre-computed $p_{i+1,i+1}$ used for the first ir. The q vector, which is the first output of each loop, is also calculated using the scalar datapath; for this case the input to the subtracter is zeroed.

The pre-loop values (lines 2-6) are also calculated with the chained scalar and vector datapaths. The scalar datapath is bypassed, by latching one of the multiplier inputs to zero. Therefore only one datapath structure is needed, thereby reducing routing stress, removing the need for a multiplexer, and simplifying the control logic.

The final pass consists of a single vector division implemented as a multiplication by the inverse. This is again handled by the scalar structure, similar as for the q calculations of the previous columns, by zeroing one input of the subtracter.

D. Architecture

Figure 1 shows the architecture of a QRD core implementing the new MGS algorithm. The RAM block is made up of one memory per column element, so that the entire column

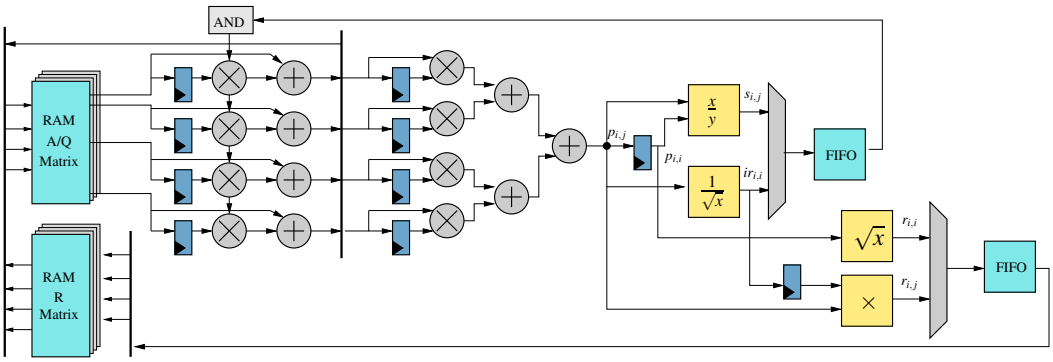

Figure 1. Proposed MGS Algorithm Architecture

can be accessed in parallel. The first column in an iteration, a_i, is always latched at the input of the multipliers. The other input of the multipliers is a common input from the math functions, either the inverse square root for the normalization of the vector ($ir_{i,i}$, the subtracter is zeroed in this case), or the divider ($s_{i,j}$), to apply the projection of the first vector onto the rest. A FIFO, or other delay mechanism, is provided to align the output of the math functions (which are generated for the previous iteration) with the vectors a_j for the current iteration. The new a_j is written back to the RAM block, and at the same time is input to the vector datapath.

The vector datapath latches the first vector it receives for an iteration, which is the normalized a_i, or normalized vector, and calculates the inner product of $p_{i,i} = \langle a_i, a_i \rangle$. This is latched at the denominator input of the divider, and is also used at the input of the inverse square root function, to generate the normalization value for the next iteration $ir_{i,i}$. The first dot-product $p_{i,j}$ is passed through the square root block in order to compute $r_{i,i}$. The output of the inverse square root ($ir_{i,i}$) is latched at the input of the multiplier. The following vector outputs, the inner products of $p_{i,j} = \langle a_i, a_j \rangle$, are divided by the latched value to produce $s_{i+1,j}$, used to calculate the angle between the first and following vectors for the next pass. At the same time $p_{i,j}$ values are multiplied by the latched $ir_{i,i}$ for obtaining $r_{i,j}$.

From Figure 1, it can be seen that for single-precision and when targeting HFP-enabled FPGAs, most of the functionality can be mapped to the HFP blocks. All the FP multiplies and the adders are mapped to the HFP blocks in gray. The math functions ($\div, \sqrt{x}, 1/\sqrt{x}$) together with the muxing and latching logic require small amounts of soft logic resources. The counting and decoding logic in the controller uses a relatively negligible amount of logic.

Although the implementation and analysis presented here is a fully parallel implementation, a multi-cycle version is also possible. Resource savings would primarily be in the floating point scalar and vector datapaths, with the elementary functions unchanged. The number of memory bits is also constant, with any change in the number of RAM blocks dependant on the ratio of matrix size to memory size.

IV. Performance Analysis

In this section we will estimate the expected sustained to peak performance of the new MGS algorithm, assuming real-world implementations. Although the theoretical sustained to peak performance is close to 100%, pipelining will reduce this by introducing stalls, as well as a smaller component of initial pipeline fill. The larger the matrix, the lower the incidence of stalling.

Although QRD is an n^3 process, our implementation will require n^2 steps as we will read an entire column vector per clock. For a $n \times n$ matrix, there will be n iterations, with the number of columns processed per iteration decreasing by 1 every pass. Ignoring pipelining, QRD therefore takes $n^2/2$ cycles.

The roundtrip through the multiply-subtract core is relatively short, comprising of the registered memory interfaces, the four or five pipeline stages of the multiply-subtract core, and a small number of additional registers, such as the multiplier input latch, and the registered multiplexer (to provide external read and write access to the RAM) on the input of the memory. Additional pipelining may be used for fitting purposes; breaking up long paths for the many floating-point datapaths can improve the timing closure of complex designs. Typically, the total delay in the multiply-subtract path will be less than 10. The scalar roundtrip path will have minimal impact on system performance, which will be dominated by the vector and elementary function depth instead, and will include the scalar outbound (to the output of the subtracter). Before the next iteration can be started, the angle between the previously calculated vector and the current vectors must be available. This will be the sum of the scalar outbound depth, the vector depth, and the division operator, along with the various hold latches and pipelines added for fitting. The scalar outbound depth is close to the scalar roundtrip path, perhaps 8. The vector depth is dependent on the vector size, and in current devices is several times the scalar path:

$$\text{vectorDepth} = 8 + (4 + 3\lceil log_2(n) \rceil) + 15 + x.$$

The divider has a latency of about 15, n is the column height of the matrix, and x represents the additional pipeline depth -

Table I
LATENCY OF THE QRD COMPONENTS FOR VARIOUS CONFIGURATIONS

n, v	Type	Latency			
		div	rsqrt	scalar	vector
64, 64	real	17	11	4	26
128, 128	real	17	11	4	30
256, 256	real	17	11	4	34
384, 384	real	17	11	4	34
512, 512	real	17	11	4	38

this is usually small. For our 256×256 example, the vector depth would be in the order of 55 cycles. If the number of columns in the iteration is greater than the vector depth, then this latency would be hidden in the processing time of the iteration, as the result of the vector pipeline is not required until the iteration is complete.

The QRD implementation would therefore require approximately the following number of cycles:

$$t_{\text{QRD}} = \sum_{i=1}^{n} \max(i, \text{vectorDepth})$$

For our 256×256 example, this is in the order of 34,025 cycles, compared to the untimed maximum of 32,768, for a sustained to peak ratio of 96%. The amount of time for the pre-main loop calculation, which only uses the vector datapath, is included here as the n^{th} sum; although the scalar datapath is not used for this first pass through the core, the data still flows through it.

For smaller matrices, such as 64x64, the latency of the pipeline is exposed, leading to a lower efficiency, here 64%. A multi-cycle version (m cycles per column, which will reduce datapath resources by $1/m$) will require the following cycles:

$$t_{\text{QRD}} = \sum_{i=1}^{n} \max(m \cdot i, \text{vectorDepth}) \quad (1)$$

Our 64x64 case increases to 8570 from 3570 cycles (2.4x) but improves system efficiency by about 50%, as the datapath structures are about 25% the size.

V. RESULTS

Our proposed algorithm is implemented in a core generator written in C++which takes as input the matrix dimension n, the target FPGA and the objective frequency. The datapath structures of our design (the scalar and vector cores), and the math functions (division and inverse square root), are generated for the Arria 10 device using the Intel DSPBA [12].

Table I presents the latencies of these compute cores when targeting the Arria 10 device (-I1), with an objective frequency set to 400MHz, for IEEE754 single-precision. With increasing n the vector latency increases roughly with $2 + 4\log_2(n)$. As it can be observed, as n increases from 128 to 512 the latency increases by only 8 cycles for a fully pipelined implementation.

The synthesis results for this architecture, for both real and complex versions, are presented in Table II.

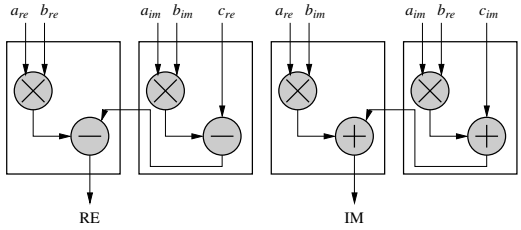

Figure 2. Complex Mult-Add Mapping to HFP DSP Blocks

Table II
SYNTHESIS RESULTS FOR VARIOUS SIZE QRD DESIGNS

	n, v	Results			
		Freq.	ALMs	DSPs	M20K
Real					
ours	64, 64	461 MHz	4808	135	72
	128, 128	445 MHz	11207	265	136
	256, 256	430 MHz	22013	526	264
	384, 384	388 MHz	32726	787	392
	512, 512	371 MHz	43404	1047	520
Complex					
ours	32,32	455 MHz	7814	270	76
[5]	32,8	368 MHz	11K	68	31
[5]	32,16	368 MHz	16.9K	118	44
ours	64,64	407 MHz	14826	530	139
[5]	64,8	368 MHz	13.6K	68	75
[5]	64,16	368 MHz	19.5K	118	83
ours	128,128	366 MHz	28865	1050	267
[6] (Figure 7)	128x64,-	150 MHz	7114 LUT	24 DSP48	72 BRAM_18K
[6] (Figure 8)	128x64,-	147 MHz	75900 LUT	392 DSP48	128 BRAM_18K

The complex multiply-add can be optimized to use the HFP DSP Blocks more effectively, and is depicted in Figure 2.

$$\begin{aligned} AB + C &= (a_{re} + i \cdot a_{im})(b_{re}) + i \cdot b_{im}) + (c_{re} + i \cdot c_{im}) \quad (2) \\ &= (a_{re}b_{re} - a_{im}b_{im} + c_{re}) + i \cdot (a_{re}b_{im} + a_{re}b_{im} + c_{im}) \\ &= (a_{re}b_{re} - (a_{im}b_{im} - c_{re})) + i \cdot (a_{re}b_{im} + (a_{re}b_{im} + c_{im})) \end{aligned}$$

The complex dot-product is split element-wise into 4 real dot-products of size n. The final real and imaginary components are assembled using an adder and a subtracter. The only subtlety is the dot-product $\langle x, y \rangle$ which requires a conjugate transpose of x.

The architectures presented in Table II have extra stages of pipeline on the control and other high-fanout signals. With increasing n, the physical distance between the placement of the control FSM and the controlled structures increases. A variable number of pipeline stages are added (depending on n) on these control signals. The output of the FIFO also feeds n mult-add blocks and therefore requires appropriate pipelining. The extra delays are accounted for at FSM generation time. Our synthesis results confirm that the pipelining strategy is effective for obtaining high-frequency, chip-filling designs.

Table III shows how the sustained to peak performance ratio improves with increasing matrix size. Our proposed architecture achieves over 95% sustained-to-peak performance for $n = 256$, and increases to over 98% for $n = 512$. The performance in GFlops is reported using the frequency numbers from

Table III
SUSTAINED-TO-PEAK AND ABSOLUTE PERFORMANCE

	n,v	Datapath latency	Peak Latency	Real Latency	Ratio	Perf. (GFlops)	Perf. μs
Real							
ours	64,64	51	2080	3355	61%	71.7	7.27
	128,128	55	8256	9741	84%	190.9	21.8
	256,256	59	32896	34607	95%	417.8	80.4
	384,385	60	73920	75690	97%	577.6	195
	512,512	65	131328	133408	98%	744.2	359.5
Complex							
ours	32,32	51	528	1888	27%	62.6	4.1
[5]	32,8	-	-			13.9	19.4
[5]	32,16	-	-			17.6	15.3
ours	64,64	56	2080	3620	57%	237	8.8
[5]	64,8	-	-			20.2	105
[5]	64,16	-	-			33.8	62.9
ours	128,128	61	8256	10086	81%	606.5	27.5
[6] Fig.7	128x64,-	-	-	13466627	1%	0.046	89553
[6] Fig.8	128x64,-	-	-	420163	3.7%	3.6	2852.9

Table II. We can observe that we can achieve over 744 GFlops while utilizing only 1047 DSPs out of the 1687 DSPs available in the largest Arria 10 device, or about 62% of the DSPs. The logic utilization is of 43404 ALMs out of 251680 ALMs, which is only 17%. Having a comfortable margin on logic utilization, we expect that frequency can be further improved by 10-20% for a 10% logic increase.

We have also compared our work against [5] which to our knowledge is currently the fastest complex FPGA QRD implementation. The results in [5] have the vector width always smaller than the matrix size, and are therefore expected to consume fewer resources, and have a longer latency than our proposed solution. This holds true for DSPs and memory blocks, but our solution consumes fewer ALMs.

The algorithmic improvements are also observed when analyzing latency. For $n = 64$ our implementation takes $8.8\mu s$ ($v = 64$) and is expected to take $21.55\mu s$ ($v = 16$) according to Equation 1. In comparison, the latency reported in [5] for $n = 64$, $v = 16$ is $62\mu s$. This shows that for similar hardware cost, our implementation is three times faster.

We have added the work in [6] for completeness. The sustained to peak ratio is about 3.7%, which agrees with the expectation for an unoptimized MGS mapping. This also illustrates the validity of the sustained to peak metric in comparing matrix decomposition results, as we are able to compare the quality of an algorithm and implementation independently of device resources. The Xilinx devices do not have floating point support, so we expect the soft logic utilization to be much higher, and we expect that a careful redesign would also improve frequency, as the individual operators are capable of about 3x the reported speed in a recent device [7].

VI. CONCLUSION

We have successfully demonstrated a number of contributions. First, our sustained to peak ratios approach 100% for even medium sized matrices, as commonly used for embedded applications such as radar. We have shown that even relatively large matrices, such as 256x256, can close timing near the maximum possible speed of the device, which is limited to around 460MHz in floating-point mode. Secondly, as current FPGA devices contain as many embedded floating-point functions as their GPU counterparts – in the order of 1TFLOPs on the lower-end to 10 TFLOPs on the higher-end – the higher efficiency of the FPGA solution (near 100% compared with a typical GPU ratio of 15% or less) means that FPGAs offers higher performance matrix decomposition processing. We have shown an ever increasing improvement in QRD performance, by careful tuning of the algorithm and implementation. Our results, in both efficiency (sustained to peak) and cost (resource utilization) are superior to previous hardware implementations. Finally, our new designs show low soft-logic usage (ALMs). The traditional soft logic and routing resources are used to support the embedded functions, rather than the typical reverse of this. The soft routing is used for zero overhead movement of data, for example when the output of the scalar cores are written back to memory and forward to the vector core simultaneously. Soft logic, when used, often only holds a vector constant in time, which both reduces power (as there is no toggling of the registers) and memory bandwidth, which further reduces power, as the constant vector is not continuously read from memory or a register file.

REFERENCES

[1] S. D. Muoz and J. Hormigo, "High-throughput FPGA implementation of QR decomposition," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 62, no. 9, pp. 861–865, Sept 2015.

[2] X. Wang and M. Leeser, "A truly two-dimensional systolic array FPGA implementation of QR decomposition," *ACM Trans. Embed. Comput. Syst.*, vol. 9, no. 1, pp. 3:1–3:17, Oct. 2009. [Online]. Available: http://doi.acm.org/10.1145/1596532.1596535

[3] M. Karkooti, J. R. Cavallaro, and C. Dick, "FPGA implementation of matrix inversion using QRD-RLS algorithm," in *Conference Record of the Thirty-Ninth Asilomar Conference onSignals, Systems and Computers, 2005.*, October 2005, pp. 1625–1629.

[4] V. Mauer and M. Parker, "Floating point STAP implementation on FPGAs," in *2011 IEEE RadarCon (RADAR)*, May 2011, pp. 901–904.

[5] M. Parker, V. Mauer, and D. Pritsker, "QR decomposition using FPGAs," in *2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS)*, July 2016, pp. 416–421.

[6] L. Miller, "Adaptive beamforming for radar: Floating-point QRD+WBS in an FPGA," *Xilinx Whitepaper*, Jun. 2014, https://www.xilinx.com/support/documentation/white_papers/wp452-adaptive-beamforming.pdf.

[7] *LogiCORE IP CORDIC v6.0*, 2012, https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_0/ds816_floating_point.pdf.

[8] J. Pettersson and I. Wainwright, "Radar signal processing with graphics processors (GPUs)," in *SAAB*, 2010.

[9] P. Du, P. Luszczek, S. Tomov, and J. Dongarra, "Soft error resilient QR factorization for hybrid system with GPGPU," *Journal of Computational Science*, vol. 4, no. 6, pp. 457 – 464, 2013, scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750313000161

[10] A. Kerr, D. Campbell, and M. Richards, "QR decomposition on GPUs," in *Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units*, ser. GPGPU-2. New York, NY, USA: ACM, 2009, pp. 71–78. [Online]. Available: http://doi.acm.org/10.1145/1513895.1513904

[11] M. Langhammer, "QRD for parallel arithmetic structures," in *2017 IEEE 24th Symposium on Computer Arithmetic (ARITH)*, July 2017, pp. 146–147.

[12] "DSP Builder Advanced Blockset," https://www.altera.com/products/design-software/model---simulation/dsp-builder/overview.html.