QR Decomposition using FPGAs

Michael Parker
Intel Programmable Systems Group
San Jose, CA. USA

Volker Mauer
Intel Programmable Systems Group
High Wycombe, UK

Dan Pritsker
Intel Programmable Systems Group
San Diego, CA. USA

Abstract— **This paper describes the architecture and implementation of a high performance QR decomposition IEEE754 single precision floating point core, using a modified Gram-Schmidt algorithm. Using Intel's new floating point Arria 10 FPGAs, synthesis is used to generate column high functional units, giving $O(n^2)$ processing times. The modified Gram-Schmidt algorithm is expressed in a different order to combine the elements of the column calculations at a later stage, which reduces the data dependencies in a deeply pipelined hardware implementation. Special vector structures in the tools and hardware architecture are supported to maximize performance. The matrix sizes are parameterized, with both square and rectangular complex matrixes supported. Two versions of the QRD core, using an example matrix size of 50x100 are presented, one of a conventional FPGA architecture (Stratix V) and the other on Floating Point FPGA (Arria 10) with comparative data on logic, registers, DSP, memory resources and Fmax. Matrix throughput and GFLOPS/W results are also provided. In addition, results for specific matrix sizes common in wireless MIMO processing are also presented**.

Keywords—FPGA, Arria 10, FP DSP block, QRD, MIMO, Cholesky

I. Introduction

Matrix processing is a heavily used technique in communications, radar, medical imaging and many other applications. It is particularly prevalent in system with have many antennas, perform MIMO (multiple input, multiple output) processing. In 5G wireless for example, QRD is used in both MIMO processing and amplifier digital predistortion adaptation. In radar, QRD is used in space-time adaptive processing (STAP) can be used to extract signals well below the noise floor. The matrix sizes used are normally modest, but the throughput and processing requirements can be very high.

Matrix inversion techniques, such as QR Decomposition, typically involve very small and large numbers in the computation process, which requires a dynamic range that is impractical to support using fixed point numerical representation. Using Intel's floating point Arria 10 FPGAs, high performance matrix processing can be accomplished using IEEE754 single precision floating point numerical representation throughout. Each DSP block in the Arria 10 contains both a single precision floating point multiplier and adder (or accumulator). In addition, a special vector dot-product mode is supported. FPGAs are known for efficient, high throughput, and low latency fixed point DSP processing, but with this architectural innovation, Intel FPGAs are ideally suited for these applications.

This main focus of the paper describes the architecture and implementation of a high performance QR decomposition IEEE754 single precision floating point core, using a modified Gram-Schmidt algorithm. Parallel floating point processing circuits are used to generate column high functional units, providing $O(n^2)$ processing times. The modified Gram-Schmidt algorithm is expressed in a different order to combine the elements of the column calculations at a later stage, which reduces the data dependencies in a deeply pipelined hardware implementation. The complex matrix sizes are parameterized, with both square and rectangular matrixes supported.

II. QR Decomposition Algorithm

A matrix A can be factored into the product of a matrix Q and a matrix R, where:

$$A = QR$$

The Q matrix is orthogonal, and the R matrix is upper right triangular. The method used for the decomposition is Gram-Schmidt orthogonalization, and is summarized below.

A = **QR**, often used to solve **Ax = b**, where **x** is unknown
Find **d** = $\mathbf{Q^T b}$, then **Rx** = **d** can be solved by back substitution
A = $[a_1, a_2, a_3, \ldots a_n]$ where a_i are column vectors

scaler $proj_v(a) = \langle v, a_i^T \rangle / \langle v * v^T \rangle$ where $\langle a, b^T \rangle$ is dot product

$u_1 = e_1$ and $e_1 = u_1 / norm\ (u_1)$
$u_2 = a_2 - proj_{u1}(a_2)$ and $e_2 = u_2 / norm\ (u_2)$
$u_3 = a_3 - proj_{u1}(a_3) - proj_{u2}(a_3)$ and $e_3 = u_3 / norm\ (u_3)$

$Q = [e_1, e_2, e_3, \ldots e_n]$ where e_i are orthonormal column vectors

$$R = \begin{bmatrix} \langle e_1, a_1 \rangle & \langle e_1, a_2 \rangle & \langle e_1, a_3 \rangle \ldots & \langle e_1, a_n \rangle \\ 0 & \langle e_2, a_2 \rangle & \langle e_2, a_3 \rangle \ldots & \langle e_2, a_n \rangle \\ 0 & 0 & \langle e_3, a_3 \rangle \ldots & \langle e_3, a_n \rangle \\ 0 & 0 & 0 \ldots & \langle e_4, a_n \rangle \\ . & . & . \quad . & . \\ 0 & 0 & 0 \quad 0 & \langle e_n, a_n \rangle \end{bmatrix} \quad \text{(upper triangular)}$$

III. QR Decomposition Restructuring

The QR Decomposition can also be expressed in code below. However, this is unsuitable for a parallel implementation, such as in FPGA.

.

```
for k=1:n
   r(k,k) = norm(A(1:m, k));
   for j = k+1:n
      r(k, j) = dot(A(1:m, k), A(1:m, j)) / r(k,k);
   end
   q(1:m, k) = A(1:m, k) / r(k,k);
   for j = k+1:n
       A(1:m, j) = A(1:m, j) - r(k, j) * q(1:m, k);
   end
end
```

The algorithm can be restructured to allow suitable FPGA implementation. The ***norm*** function can be replaces by a dot-product and square root operation, which are more easily hardware realizable. Secondly, the first inner loop requires ***r(k,k)*** to be computed first, which has a long latency. Third, the second inner loop requires ***q(1:m, k)*** to be computed first, which also has a long latency. In order to remove the data dependencies the order of operations will be changed. All of the **r** terms will be calculated first. This can be before $\mathbf{q_i}$ is known.

```
for k=1:n
   r(k,k) = norm(A(1:m, k));
   r2(k,k) = dot(A(1:m, k), A(1:m,k);
   r(k,k) = sqrt(r2(k,k));
   for j = k+1:n
      r(k, j) = dot(A(1:m, k), A(1:m, j)) / r(k,k);
      rn(k, j) = dot(A(1:m, k), A(1:m, j));
      r(k, j) = rn(k,j)/ r(k,k);
   end
   q(1:m, k) = A(1:m, k) / r(k,k);
   for j = k+1:n
       A(1:m, j) = A(1:m, j) - r(k,j) * q(1:m,k);
   end
end
```

Next, the following substitutions can performed in the second inner loop. Replace **r(k,j)** with **rn(k,j) / r(k,k)** and replace **q(1:m,k)** with **A(1:m,k) / r(k,k)** .

```
for k=1:n
   r2(k,k) = dot(A(1:m, k), A(1:m,k);
   r(k,k) = sqrt(r2(k,k));
   for j = k+1:n
      rn(k, j) = dot(A(1:m, k), A(1:m, j));
      r(k, j) = rn(k,j)/ r(k,k);
   end
   q(1:m, k) = A(1:m, k) / r(k,k);
   for j = k+1:n
      A(1:m, j) = A(1:m, j) - r(k,j) * q(1:m,k);
      A(1:m, j) = A(1:m, j) - rn(k,j)/ r(k,k) * A(1:m,k) / r(k,k) ;
   end
end
```

Then operations can be re-ordered into two functional groups. Every r term depends on the norm of the first column for the current iteration, the calculation of which requires a deep pipeline. The same datapath used for the vector product required for the norm can also be used to calculate the vector operation for each r term. All vector operations for the r term calculation, $\langle \mathbf{v_i}, \mathbf{v_i} \rangle$ for $\mathbf{r_{ii}}$, and $\langle \mathbf{v_j}, \mathbf{v_i} \rangle$ for $\mathbf{r_{ij}}$ can be issued on subsequent clock cycles.

Following the vector operation, a separate circuit can be used for the square roots and inverse square roots. The square root and divide operations are scheduled as late as possible, after the vector operations. All of the **r** terms can therefore be calculated one per clock, without wait states, once the pipeline is full.

```
for k=1:n
   r2(k,k) = dot(A(1:m, k), A(1:m,k);
   for j = k+1:n
      rn(k, j) = dot(A(1:m, k), A(1:m, j));
   end
   for j = k+1:n
       A(1:m, j) = A(1:m, j) – (rn(k,j) / r2(k,k)) * A(1:m,k);
   end
end

for k=1:n
   r(k,k) = sqrt(r2(k,k));
   for j = k+1:n
      r(k, j) = rn(k,j)/ r(k,k);
   end
   q(1:m, k) = A(1:m, k) / r(k,k);
end
```

The upper loop can run with few stalls, as there are no long latency math operations. This is where the bulk of the computation is performed. The **r** terms are then used as one of the inputs to the vector multiplier, the first column for the outer loop is generated, and all of the subsequent columns updated. The only data dependency is between the first **r** term output of the datapath and the start of the $\mathbf{q_i}$ vector calculation, which may introduce a wait state between the two parts of the inner loop. As the pipeline depth of the datapath will be fixed, but the inner loop reduces by one with each pass, a wait state will be required at some point in the processing of the matrix, which once started, will increase by one clock per pass.

The lower loop can run as data becomes available, and is relatively latency insensitive.

IV. Computation Units

FPGAs have incorporated hundreds or even thousands of fixed point multipliers in specialized DSP blocks, making the FPGA the most efficient and powerful programmable signal processing architecture. This capability, along with roughly equal numbers of specialized memory blocks, has long been recognized by developers of wireless, military, broadcast and medical systems. Each of the Arria 10 DSP blocks can be independently configured in three modes: 18 bit fixed point, with two multipliers, 27 bit fixed point with one multiplier or a new single precision floating point mode with one multiplier and adder. The floating point mode is depicted in Fig 1.

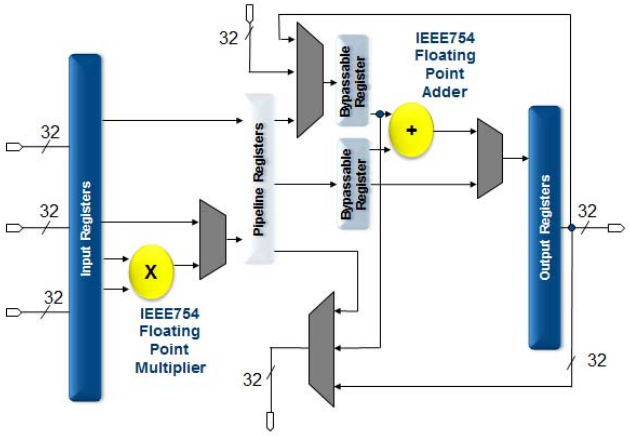

Fig 1: Floating Point DSP Blocks

The DSP block floating point mode also includes a vector mode, which allows full use of all multipliers and adders to compute dot products, which is the bulk of the processing in many algorithms, including QR, Cholesky and SV decompositions.

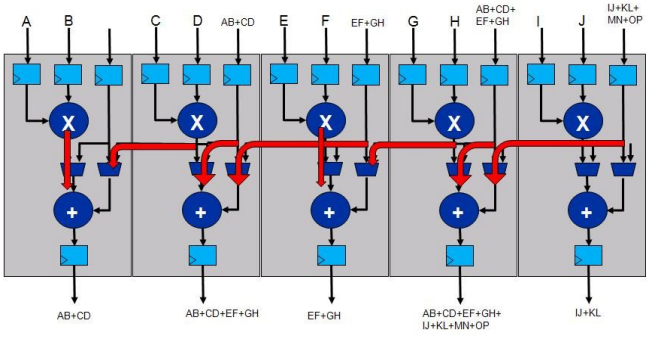

Fig 2: Vector dot-product mode

To implement a vector dot-products engine which can compute one output per clock cycle, a bank of multipliers spanning the vector size is used, and followed by an adder tree. This structure can be implemented using the FPGA DSP blocks, using a recursive, pipelined structure as shown in Fig 2. Both dedicated paths between DSP blocks as well as FPGA routing is used to construct this, resulting in the use of nearly all the floating point adders in the same DSP blocks used for the vector multiply. This structure provides one vector dot-product result per clock cycle.

V. FPGA Impplementation

Two parallel datapaths are used. The first datapath computes the inner product of two input vectors, and the second datapath computes various combinations of roots and inverse square roots. Vector operations can be used for the column processing, by either entire column or multi-clock sub-column calculations.

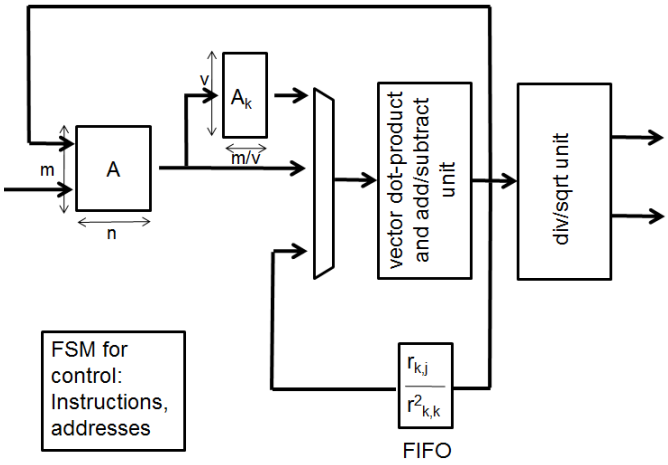

Fig 3. QRD Computational Structures

Both the input (***A***) and processing ***(Q)*** matrices are stored on a row basis, using FPGA block memories, allowing an entire column to be accessed in a single cycle. At the start of each iteration, the entire first column ($\mathbf{v_i}$) is read, and the norm calculated. The inner product of the column is calculated through the first datapath, which is stored for use on the input of the second datapath. The second datapath generates the inverse norm of the first inner product, and then uses the stored first inner product $\langle \mathbf{v_i},\mathbf{v_i} \rangle$ to generate the following values, which are the inner product of the current column and first column $\langle \mathbf{v_j},\mathbf{v_i} \rangle$, divided by the inner product of the first column by itself. Each clock cycle the columns are read, creating a stream of continuous norm combinations. This sequence of operations generates the results of the first part of the inner loop.

$$r_{ii} = \|v_i\|$$
for $j = i + 1$ ***to n do***
$$r_{ij} = v_j \, v_i / \|v_i\|$$
end for

The same sequence of columns is read a second time, timed so that the columns arrive at the first datapath at the same time as the norm combinations for that column ($\mathbf{r_{ii}},\mathbf{r_{ij}}$). The first column ($\mathbf{v_i}$) is stored, and the first datapath then calculates the new column iterations, which are a column subtracted from that column's norm combination multiplied by the latched first

column. This sequence of operations generates the results of the second part of the inner loop.

$$q_i = v_i / r_{ii}$$
$$\textbf{for } j = i + 1 \textbf{ to } n \textbf{ do}$$
$$v_j = v_j - r_{ij} v_i$$
$$\textbf{end}$$

Once the $\boldsymbol{Q}$ matrix has been calculated, the $\boldsymbol{R}$ matrix can be computed from the $\boldsymbol{Q}$ matrix and the input matrix $\boldsymbol{A}$.

$$R = Q^T A$$

The transposed rows of $\boldsymbol{Q}$ are columns, which are available in parallel due to the row memory architecture used in the core. The calculation of $\boldsymbol{R}$ is therefore a set of dot products of the columns of $\boldsymbol{Q}$ and $\boldsymbol{A}$. Output elements can be generated one per clock cycle.

As described above, the QRD core is computed with maximum parallelism and throughput. However, an additional option is provided to set the vector size to be a fraction, typically ½ or ¼ of the column length. This has two beneficial effects. First, it provides a close to proportional effect on the computational resources and of the QRD core. Reducing the vector size reduces the impact of the longer latency divide and inverse operations, so often has a less than proportional effect on the matrices per second throughput, as shown in the tables of results.

VI. QRD Resource and Performance Results

Results are shown for several QRD core parameterizations. Due to the novelty of the FPGA floating architecture in Arria10, two sets of results are presented. The upper figure in each box is for Arria 10, and the lower figure is for an implementation on Stratix V FPGA. The Stratix V FPGA has a similar architecture and performance to Arria 10, excepting the floating point DSP. This arrangement helps contrast the impact of the single precision floating point DSP block. Lastly, power measurements are not provided, as only a portion of the FPGA resources are used, and the static power consumption would skew the QRD core results. The resources and clock rate shown are generated from the Quartus II software, and represent a timing closed FPGA design. The reduction on logic and routing due to the Arria 10 floating point DSP block allow Arria 10 to achieve significantly higher Fmax than Stratix V designs. QRD FLOPS per [m x n] matrix is computed using the standard equation

$$\boldsymbol{FLOPS = 8mn^2 + 6.5n^2 + mn}$$

Compiled and timing closed results are provided in table below. For both Stratix V (conventional FPGA architecture) and Arria 10 (floating point FPGA architecture), the medium speed grade device is used. In Stratix V, the floating point functionality is constructed by using programmable logic and fixed point DSP blocks. Arria 10 fully hardens the floating point computation al units, providing significant advantages.

The Arria 10 device used is 10AS066N2F40I2SGES. This device is 660 kLE of logic resources, and contains 1688 DSP blocks (3376 floating point units), with a peak floating point rating of 1.5 TFLOPS.

TABLE I. QRD with/without Floating Point DSP

Device	Stratix V	Arria 10	Stratix V	Arria 10
Matrix (complex)	50x100	50x100	100x200	100x200
Vector (complex)	50	50	50	50
Logic Elements	279K	49.4K	283K	54.8K
DSP blocks	227	424	228	423
M20K memory	230	188	281	241
Fmax (MHz)	259	328	260	300
Latency (us)	45	28	213	174
Matrices per sec	31628	50853	5920	7195
GFLOPS	43.8	70.3	64.3	78.1

Comparing results between a conventional and the new floating point FPGA architecture shows an approximately 20-50% increase on QRD throughput, with a corresponding logic reduction of about 80%. DSP block usage is increased with Arria 10, but that is a worthwhile trade-off, as the logic and routing are the main components of the FPGA. Hard blocks such as the DSP consume minimum area within the FPGA, relative to logic.

TABLE II. Medium size QRD for wireless MIMO

Device	Arria 10	Arria 10	Arria 10	Arria 10
Matrix (complex)	32x32	32x32	64x64	64x64
Vector (complex)	8	16	8	16
Logic Elements	29.4K	44.9K	36.1K	50.5K
DSP blocks	68	118	68	118
M20K memory	31	44	75	83
Fmax (MHz)	368.64	368.64	368.64	368.64
Latency (us)	19.4	15.3	105	62.9
Matrices per sec	51.5K	65.4K	9.47K	15.9K
GFLOPS	13.9	17.6	20.2	33.8

TABLE III. Small size QRD for wireless MIMO

Device	Arria 10	Arria 10	Arria 10	Arria 10
Matrix (complex)	16x16	16x16	16x32	16x32
Vector (complex)	4	8	4	8
Logic Elements	19.8K	24.6K	49.8K	69.0K
DSP blocks	44	70	42	68
M20K memory	16	22	16	22
Fmax (MHz)	368.64	368.64	368.64	368.64
Latency (us)	7.39	6.10	16.9	12.7
Matrices per sec	135K	164K	59.0K	78.7K
GFLOPS	4.7	5.7	8.2	10.9

VII. Areas for future Optimization

There are further optimizations which can be made to the QRD core, built with Intel's DSPBuilder tool for FPGAs. One optimization that has been found to be very effective on a similar algorithm, the Cholesky decomposition, is to create a multi-channels core. For smaller matrix sizes, pipeline stalls can occur more easily. This can be mitigated by using a vector size of say ¼ of the row count, but this approach limits the processing throughput. Alternatively, a vector width equal to the row count can be used for maximum throughput, which interleaved matrix processing, or channelization used. For example, the QRD core could be designed as a 16 channel core, where the rows of the different matrices are interleaved. This interleaving virtually eliminates pipeline stalls and provides for maximum computational efficiency. One penalty is that more block memory resources are required to store the interleave matrix data.

Another optimization likely for future devices is increased flexibility in the vector dot-product mode of the DSP block. Currently, this mode is set at compile time, so that separate DSP resources are required to for dot-product engine and vector add-subract engine. This results in the larger amount of DSP resources used in Arria 10 relative to from Stratix V (logic based floating point has full flexibility, and can use the same fixed point DSP resources for both functions).

VIII. Conclusions

Once multipliers became hardened in high numbers, FPGAs rapidly became the architecture of choice for high performance, fixed point signal processing. Floating point was always possible, but generally avoided due to the high amount of programmable routing and logic required, which severely comprised both performance and efficiency. Now that floating point support has been hardened in the FPGA's DSP blocks, Intel FPGAs are comparably efficient for both fixed and floating point processing. Simultaneously, MIMO, beamforming, equalization, large FFTs and other applications which require the dynamic range of floating point are increasingly prevalent. This new generation of FPGA architecture provides low power, deterministic, low latency and high performance floating point capabilities, along with the traditional attributes of FPGAs: programmable, harsh environments, ubiquitous connectivity.

References

[1] M. Fitton, S. Perry, R. Jackson, "Reconfigurable Antenna Processing using FPGA based Application Specific Integrated Processors", SDR Forum 2004

[2] X. Wang, M. Leeser, "FPGA Based Systolic Array Implementation of QR Transformation Using Givens Rotations", FPGA2008

[3] X. Wang, "Variable Precision Floating-Point Divide and Square Root for Efficient FPGA Implementation of Image and Signal Processing Algorithms", PhD Thesis, 2007, Northeastern University

[4] P.N. Ganchosov, et.al.,"FPGA Implementation of Modified Gram-Schmidt QR Decomposition".,ProRISC 2008

[5] M. Gay, “Real-time FPGA implementation of adaptive beamforming using QR decomposition”, HPEC2005

[6] Karkooti, Marjan Cavallaro, Joseph R.C. Dick, FPGA Implementation of Matrix Inversion Using QRD-RLS Algorithm,Asilomar 2005

[7] D. Boppana, K. Dhanoa, J. Kempa, FPGA based Embedded Processing Architecture for the QRD-RLS Algorithm, FCCM2004

[8] A. Irturk, Implementation of QR Decomposition Algorithm using FPGAs, MSc Thesis, 2007, UC Santa Barbara

[9] Walke, R. Adaptive Beamforming using QR in FPGA, HPEC2002

[10] Yong Dou, et.al.64-bit Floating-Point FPGA Matrix Multiplication, FPGA2005

[11] Ling Zhuo, V. Prasanna, Scalable and Modular Algorithms for Floating-Point Matrix Multiplication
on Reconfigurable Computing Systems IEEE Transactions on Parallel and Distributed Systems (TPDS), Vol. 18, No. 4, pp. 433-448, April 2007

[12] Ronald Scrofano, Ling Zhuo and Viktor K. Prasanna, *Area-Efficient Evaluation of Arithmetic Expressions Using Deeply Pipelined Floating-Point Cores,* IEEE Transactions on Very Large Scale Integration Systems (TVLSI), Vol. 16, Issue 2, pp. 167-176, February 2008

[13] ANSI/IEEE standard 754-1985, *IEEE Standard for Floating-Point Arithmetic*, 1985