FPGA Implementation of Gram-Schmidt QR Decomposition Using High Level Synthesis

Parth Desai[1], Semih Aslan[2] and Jafar Saniie[1]

[1]Department of Electrical and Computer Engineering
Illinois Institute of Technology
Chicago, IL, 60616, U.S.A.

[2]Department of Electrical and Computer Engineering
Texas State University
San Marcos, TX, 78666, U.S.A.

Abstract—**In this paper, a high precision QR decomposition implementation on FPGA is discussed. The Gram-Schmidt algorithm is used and synthesized from high level language to Register Transfer Level (RTL) Verilog HDL, using High Level Synthesis (HLS). The RTL Code is synthesized and implemented on Xilinx Zedboard FPGA. The IP core that is generated during simulation is designed for 10x10 floating point and fixed point implementation which can be scaled up for higher matrix sizes. The results for scaled up 100x100 matrix implementation are also discussed. Fixed point data type provides twice as fast implementation and consumes nearly 30% less area compared to floating point implementation with no more than 3% error in precision, for 10x10 matrix size. The error percentage would vary with the size of matrix as well.**

Keywords—FPGA, Gram-Schmidt, HLS, QR Decomposition, ZedBoard.

I. INTRODUCTION

In this paper, an IP core that is designed using High Level Synthesis (HLS) is discussed. The implemented core performs QR decomposition of 10x10 input matrix **A** and generates two 10x10 matrices, **Q** and **R** as outputs. The data type used here is floating point data type which gives higher precision and dynamic range [1]. An alternative to this is also introduced, which consume less resources and uses 16-bit fixed point data type, but has a precision tradeoff [1]. The design is further scaled up to 100x100 matrix size for both floating and fixed points implementations using loop unrolling technique [2] [3] which is used to increase the performance is also discussed. Based on the data type used, two different designs are created for various applications where precision and resource utilization is desired.

This paper is structured as follows: Section II provides a brief discussion about the QR decomposition algorithm using Gram-Schmidt. Section III discusses the HLS design flow and implementation, and section IV discusses the proposed design implementation results.

II. QR DECOMPOSITION USING GRAM-SCHMIDT

A. QR Decomposition Techniques

There are various applications of QR decomposition in the field of linear algebra and digital signal processing [4]. It is used in various signal processing applications such as echo cancellation, Multiple Input Multiple Output (MIMO) and beamforming systems [5]. Design and verification of a QR decomposition is complex and requires higher time-to-market (TTM). To reduce TTM and create more efficient design a HLS design and verification method is proposed.

The aim of QR decomposition is to factorize a linearly dependent matrix A in to two different matrices, **Q** and **R**, where **Q** is unitary matrix and **R** is upper triangular matrix, such that **A** = **QR**. There are several QR factorization algorithms used for hardware implementations, such as Givens Rotation, Householders transform, and Gram-Schmidt [5]. These algorithms have different computational power, numerical stability and latency [6]. We will discuss Gram-Schmidt algorithm hardware implementation and verification in this paper.

B. Gram-Schmidt Algorithm

In certain applications, the computation is simplified if we have orthogonal vectors. We can produce an orthogonal set of vectors **T'** from **T** with same span as **T**. This is Gram-Schmidt Orthogonalization [7]. So, for set of vectors $\mathbf{T} = \{\mathbf{p_1}, \mathbf{p_2}, \ldots\ldots, \mathbf{p_n}\}$, we must find a set of vectors $\mathbf{T'} = \{\mathbf{q_1}, \mathbf{q_2}, \ldots\ldots, \mathbf{q_{n'}}\}$ with $n' \leq n$ so that, $\mathrm{span}\{\mathbf{q_1}, \mathbf{q_2}, \ldots\ldots, \mathbf{q_{n'}}\} = \mathrm{span}\{\mathbf{p_1}, \mathbf{p_2}, \ldots\ldots, \mathbf{p_n}\}$ and $\langle \mathbf{q_i}, \mathbf{q_j} \rangle = \delta_{i,j}$.

The algorithm works as follows [5] [7].

- First step is to normalize the vector by applying (1) to calculate $\mathbf{q_1}$.

$$q_1 = \frac{p_1}{||p_1||} \tag{1}$$

- Compute the difference in projection between $\mathbf{p_2}$ onto $\mathbf{q_1}$ and $\mathbf{p_2}$ by using (2) which is shown in the Fig. 1.

$$e_2 = p_2 - \frac{\langle p_2, q_1 \rangle}{||q_1||^2} q_1 = p_2 - \langle p_2, q_1 \rangle q_1 \tag{2}$$

If $\mathbf{e_2} = 0$ than it is discarded. If $\mathbf{e_2} \neq 0$ than we normalize as in (3).

$$q_2 = \frac{e_2}{||e_2||} \tag{3}$$

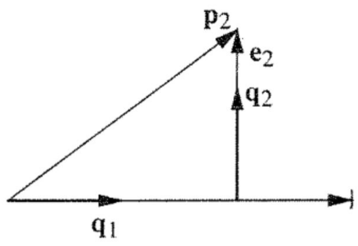


Fig. 1. Orthogonal Projection of Vectors [5]

- Similarly, we can obtain $\mathbf{e_3}$ and $\mathbf{q_3}$ as shown below in (4) and (5).

$$e_3 = p_3 - \langle p_3, q_1\rangle q_1 - \langle p_3, q_2\rangle q_2 \tag{4}$$

$$q_3 = \frac{e_3}{||e_3||} \tag{5}$$

- So similarly, we have following generalized form of $\mathbf{e_k}$ $\mathbf{q_k}$ as in (6) and (7) respectively.

$$e_k = p_k - \sum_{i=1}^{k-1} \langle p_k, q_i\rangle q_i \tag{6}$$

$$q_k = \frac{e_k}{||e_k||} \tag{7}$$

So now matrix $\mathbf{A = \{p_1, p_2, \ldots\ldots, p_n\}}$ and $\mathbf{Q}$ can be obtained by stacking vectors $\mathbf{q_n}$ such that, $\mathbf{Q = \{q_1, q_2, \ldots, q_n'\}}$ and similarly $\mathbf{R}$ can be obtained as shown below,

$$R = \begin{bmatrix} ||p_1|| & \langle p_2, q_1\rangle & \langle p_3, q_1\rangle & \cdots & \langle p_n, q_1\rangle \\ & ||e_2|| & \langle p_3, q_2\rangle & \cdots & \langle p_n, q_2\rangle \\ & & ||e_3|| & \cdots & \langle p_n, q_3\rangle \\ & & & & \vdots \\ & & & \cdots & ||e_n|| \end{bmatrix}$$

This gives decomposition of **A** in to two matrices **Q** and **R**.

III. High Level Synthesis Design Flow and Implementation

For the implementation [8], the algorithm is first coded in C++ language. This code is than synthesized to RTL level Verilog HDL code using HLS. HLS creates IP block that has the same behavior as described in higher level language. This IP block is then combined with the Zynq IP Cores to synthesize and implement [8] the complete design on FPGA.

A. HLS Design Flow

The HLS design flow [9] [10] simply converts C/C++ code into HDL after some modifications. The algorithm in C++ is modified to create a separate test-bench and top function. The HLS tool, for top function, supports certain libraries for mathematical function but doesn't support standard I/O library, so a separate top module must be designed which would perform the desired function. Designed top function can be tested using a test-bench created using standard C++ libraries. This top function will be converted to an IP block which will perform the desired function, like the top function.

After this we run the C simulation to test our results using GCC/CLANG compiler embedded in the tool. After it passes we run the C synthesis to generate the IP block with RTL level Verilog HDL code. The generated IP block is tested against C test-bench which we call C/RTL co-simulation. Complete high level synthesis design flow is shown in Fig. 2.

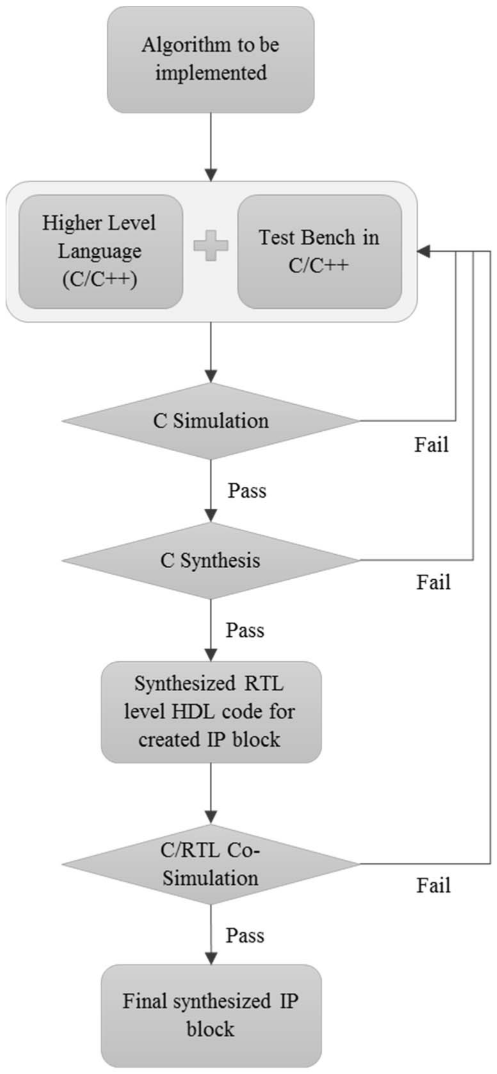

Algorithm to be implemented
Higher Level Language (C/C++)
Test Bench in C/C++
C Simulation
Fail
Pass
C Synthesis
Fail
Pass
Synthesized RTL level HDL code for created IP block
C/RTL Co-Simulation
Fail
Pass
Final synthesized IP block

Fig. 2. HLS Flow [11]

On passing all the test, final IP block is generated. This IP block is to be exported to the FPGA. This whole process is discussed in the next section.

B. *Overall Synthesis and Implementation Flow*

After generating the RTL block using HLS, this IP block needs to be connected to Zynq Processing System and the Block RAMs in the Zynq7020 chip, to complete the design. The input matrix and output matrix are stored in the Block RAM which could be accessed via Zynq using BRAM controllers. This whole design is synthesized and physically implemented, to test the timing and power estimates based on the final physical implementation. After implementation, a bit-stream file is generated which is imported to ZedBoard. The implemented design is tested using the test-bench and the results could be compared between the hardware and software in terms of timing and precision/percentage error. The Complete Synthesis and implementation flow is shown in Fig. 3.

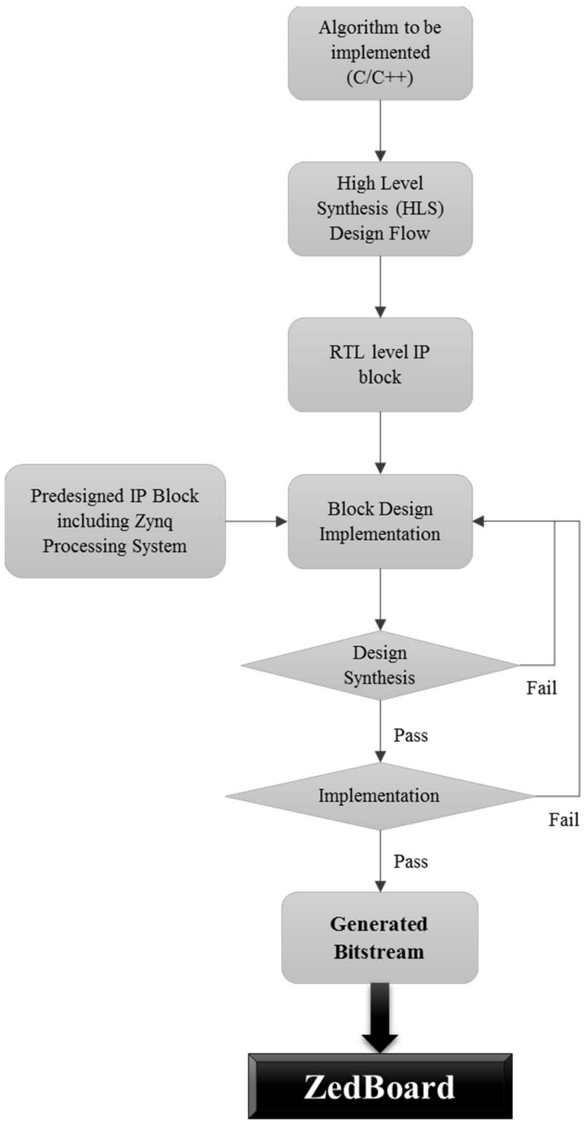

Algorithm to be implemented (C/C++)
High Level Synthesis (HLS) Design Flow
RTL level IP block
Predesigned IP Block including Zynq Processing System
Block Design Implementation
Design Synthesis
Fail
Pass
Implementation
Fail
Pass
Generated Bitstream
ZedBoard

Fig. 3. Design Implementation flow [11]

In the next section the implementation results are discussed for floating point matrix input and outputs.

IV. IMPLEMENTATION RESULTS

In these implementations two different type of data types are used.

- 32-bit floating point data type
- 16-bit fixed point data type

A test bench is created in C++ which takes a randomly generated 10x10 matrix as an input. The implementation results are compared with MATLAB. The algorithm was implemented in C++ and it was synthesized to RTL level using High Level Synthesis, and an IP block was generated. The IP block generated was then implemented on ZedBoard which has Zynq 7000 and a dual core Cortex-A9 processors.

The implementations with floating point and fixed point data type discussed here having their advantages and can be used for applications based on the requirements. The floating-point implementation provides nearly 100% precision with the resource utilization shown in Table I for post synthesis results.

TABLE I. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FLOATING POINT DATA TYPE

Name	Utilization	Total Available	% Utilization
LUT	5862	53200	11.02
FF	4250	106400	3.99
DSP48E	6	220	2.73
BRAM_18K	2	280	0.71

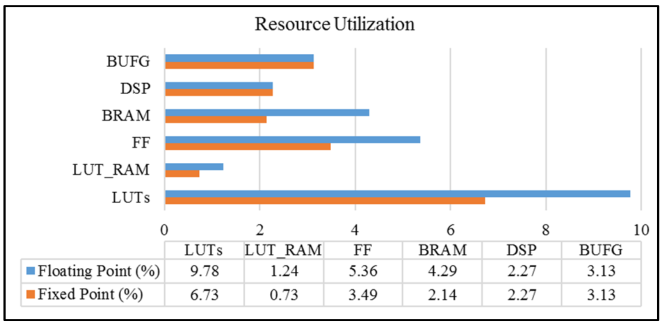
Resource utilization for fixed point data type for post RTL synthesis result is shown in Table II.

TABLE II. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FIXED POINT DATA TYPE

Name	Utilization	Total Available	% Utilization
LUT	5291	53200	9.95
FF	3960	106400	3.72
DSP48E	6	220	2.73
BRAM_18K	6	280	2.14

For fixed point data type implementation, there is nearly 30% reduction in resource utilization compared to floating point implementation. The comparison of resource utilization between the floating point and fixed point data type is shown in the Fig. 4. These results are for post implementation stage.

The post synthesis results show that the implementation with fixed point data type is twice as fast as floating point data type. The maximum latency for floating point design is 21,073 clock cycles and with fixed point design is 10,591 clock cycles. The clock frequency used is 100MHz. These latencies are for post RTL synthesis results.



	LUTs	LUT_RAM	FF	BRAM	DSP	BUFG
🟦 Floating Point (%)	9.78	1.24	5.36	4.29	2.27	3.13
🟧 Fixed Point (%)	6.73	0.73	3.49	2.14	2.27	3.13

Fig. 4. Post-Implementation Resource Utilization Comparison for 10x10 Matrix

The 10x10 matrix was scaled up to 100x100 matrix size. The resource utilization increased by a small factor with increase in the latency. Table III and Table IV shows post RTL synthesis resource utilization results for floating point data type and fixed point data type respectively.

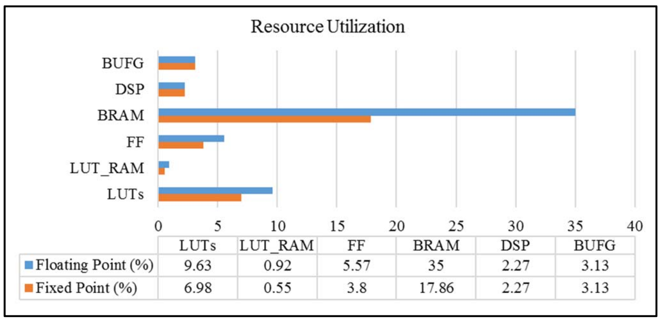
TABLE III. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FLOATING POINT DATA TYPE 100X100 MATRIX

Name	Utilization	Total Available	% Utilization
LUT	6093	53200	11.45
FF	4391	106400	4.13
DSP48E	6	220	2.73
BRAM_18K	2	280	0.71

TABLE IV. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FIXED POINT DATA TYPE 100X100 MATRIX

Name	Utilization	Total Available	% Utilization
LUT	5467	53200	10.28
FF	4240	106400	3.98
DSP48E	6	220	2.73
BRAM_18K	50	280	17.86

Fig. 5 shows comparison between floating point and fixed point, post implementation results. It shows that fixed point implementation has nearly 40 percent reduction in resource utilization compared to floating point implementation and is 3 times faster as well.



	LUTs	LUT_RAM	FF	BRAM	DSP	BUFG
🟦 Floating Point (%)	9.63	0.92	5.57	35	2.27	3.13
🟧 Fixed Point (%)	6.98	0.55	3.8	17.86	2.27	3.13

Fig. 5. Post-Implementation Resource Utilization Comparison for 100x100 Matrix

To increase the performance of 100x100 matrix implementation, loop unrolling technique is used. Two different implementations, one with unrolling factor 2 and another with unrolling factor 4 are presented. This technique shows reduction in latency with the tradeoff of area utilization.

Table V and Table VI shows post RTL synthesis results for 100x100 matrix with loop unrolling factor of 2. This implementation show 32% reduction in latency for floating point and 25% reduction in latency for fixed point implementation with respect to original 100x100 implementation.

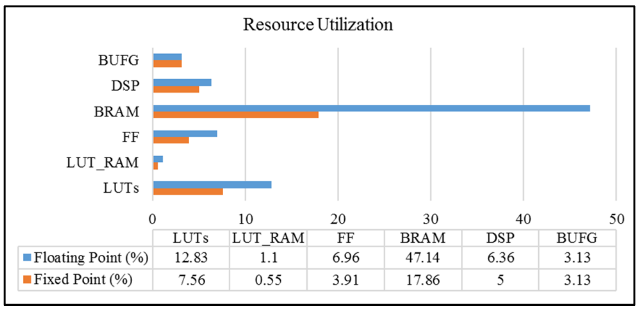
TABLE V. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FLOATING POINT DATA TYPE 100X100 MATRIX WITH UNROLL FACTOR 2

Name	Utilization	Total Available	% Utilization
LUT	7914	53200	14.88
FF	5632	106400	5.29
DSP48E	19	220	8.64
BRAM_18K	4	280	1.43

TABLE VI. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FIXED POINT DATA TYPE 100X100 MATRIX WITH UNROLL FACTOR 2

Name	Utilization	Total Available	% Utilization
LUT	6074	53200	11.42
FF	4596	106400	4.32
DSP48E	16	220	7.27
BRAM_18K	50	280	17.86

Fig. 6 shows comparison between floating point and fixed point, post implementation results of 100x100 matrix with unroll factor of 2. It shows that fixed point implementation has nearly 50 percent reduction in resource utilization compared to floating point implementation and is nearly 3 times faster. The latency has decreased at the cost of area utilization.



	LUTs	LUT_RAM	FF	BRAM	DSP	BUFG
🟦 Floating Point (%)	12.83	1.1	6.96	47.14	6.36	3.13
🟧 Fixed Point (%)	7.56	0.55	3.91	17.86	5	3.13

Fig. 6. Post-Implementation Resource Utilization Comparison for 100x100 matrix with unroll factor 2

Similar to the unrolling factor 2, Table VII and Table VIII shows post RTL synthesis resource utilization results for unrolling factor 4. This implementation shows nearly 52% reduction in latency for floating point and 45% reduction in latency for fixed point implementation compared to the original 100x100 matrix implementation. As a tradeoff to the latency improvement, resource utilization has increased by 15% from original percent of resource utilization for both floating point and fixed point implementation.

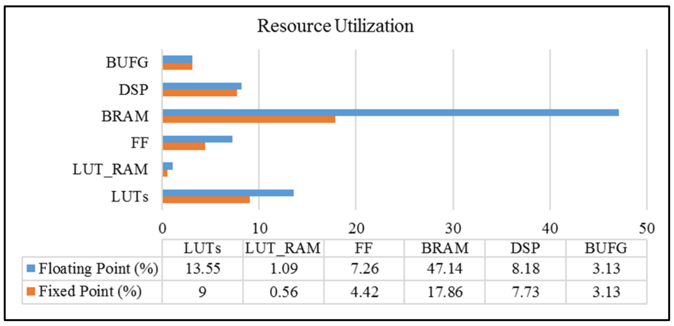
TABLE VII. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FLOATING POINT DATA TYPE100X100 MATRIX WITH UNROLL FACTOR 4

Name	Utilization	Total Available	% Utilization
LUT	8349	53200	15.69
FF	5950	106400	5.59
DSP48E	27	220	12.27
BRAM_18K	4	280	1.43

TABLE VIII. POST RTL SYNTHESIS RESOURCE UTILIZATION FOR FIXED POINT DATA TYPE 100X100 MATRIX WITH UNROLL FACTOR 4

Name	Utilization	Total Available	% Utilization
LUT	7223	53200	13.58
FF	5574	106400	5.24
DSP48E	28	220	12.73
BRAM_18K	50	280	17.86

Fig. 7 shows comparison between post implementation results of floating point and fixed point, for 100x100 matrix with unroll factor of 4. It shows that fixed point implementation has nearly 46 percent reduction in resource utilization compared to floating point implementation and is nearly 2.8 times faster. The latency has decreased at cost of resource utilization.



	LUTs	LUT_RAM	FF	BRAM	DSP	BUFG
🟦 Floating Point (%)	13.55	1.09	7.26	47.14	8.18	3.13
🟧 Fixed Point (%)	9	0.56	4.42	17.86	7.73	3.13

Fig. 7. Post-Implementation Resource Utilization Comparison for 100x100 matrix with unroll factor 4

TABLE IX. LATENCY TABLE FOR 100X100 MATRIX SIZE

Method Implemented	Floating point data type latency (ms)	Fixed Point data type latency (ms)
Without Loop Unrolling	209	64
Loop Unrolling by factor 2	141	48
Loop Unrolling by factor 4	99	35

Table IX above shows significant improvement in the latency for 100x100 matrix size for different implementations, both for floating point and fixed point data type, with and without loop unrolling technique. These latencies are for post synthesis results and for 100MHz clock frequency.

V. CONCLUSION AND FUTURE WORK

In this work, a unique way of implementing Gram-Schmidt algorithm using HLS, for 10x10 matrix input and a scaled up 100x100 matrix implementation is discussed [12]. Two different data types were used which have different applications. The floating-point implementation has very high precision [12] and can be used if the precision is the priority. The fixed-point data type with 16-bit inputs provide a better option with 30 percent reduction in resource utilization and twice faster than floating point implementation, with error not more than 3 percent.

A scaled up 100x100 matrix application is also discussed with the results which shows around twice as much resource utilization post implementation and with increase in latency. Latency reduction technique, loop unrolling [2] [3], is discussed which shows significant improvement in the latency with resource utilization tradeoff. This shows a new approach for QR decomposition technique using high level synthesis and any of these implementation techniques could be selected based on application.

The future scope of this project is to improve this implementation by further optimizing the algorithm and trying to achieve higher throughput by trying different hardware/software implementation techniques.

ACKNOWLEDGMENT

We would like to thank Xilinx, Inc for their valuable support with the software and documentation.

REFERENCES

[1] S. Aslan, E. Oruklu and J. Saniie, "A high-level synthesis and verification tool for fixed to floating point conversion," *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Boise, ID, 2012, pp. 908-911.

[2] J. L. Ayala, D. Atienza, M. Lopez-Vallejo, J. M. Mendias, R. Hermida and C. A. Lopez-Barrio, "Optimal loop-unrolling mechanisms and architectural extensions for an energy-efficient design of shared register files in MPSoCs," *Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'05)*, 2005, pp. 7 pp.-.

[3] G. Velkoski, M. Gusev and S. Ristov, "The performance impact analysis of loop unrolling," *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, 2014, pp. 307-312.

[4] S. Aslan, S. Niu and J. Saniie, "FPGA implementation of fast QR decomposition based on givens rotation," *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS), Boise, ID, 2012*, pp. 470-473.

[5] S. Aslan, E. Oruklu and J. Saniie, "Realization of area efficient QR factorization using unified division, square root, and inverse square root hardware," *2009 IEEE International Conference on Electro/Information Technology*, Windsor, ON, 2009, pp. 245-250.

[6] R. T. Kobayashi and T. Abrão, "Stability analysis in Gram-Schmidt QR decomposition," in *IET Signal Processing*, vol. 10, no. 8, pp. 912-917, 10 2016.

[7] Todd K. Moon, Wynn C. Stirling, Mathematical Methods and Algorithms for Signal Processing, Prentice Hall Upper Saddle River, NJ 07458.

[8] Zynq-7000 All Programmable SoC Embedded Design Tutorial, A Hands-On Guide to Effective Embedded System Design, 2015, Available at

https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_1/ug1165-zynq-embedded-design-tutorial.pdf.

[9] Xilinx Vivado Design Suite, HLS v2016.4, 2016, Available at https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug871-vivado-high-level-synthesis-tutorial.pdf.

[10] Vivado Design Suite Tutorial, design flow overview, 2016 Available at, https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug888-vivado-design-flows-overview-tutorial.pdf.

[11] C. Desmouliers, E. Oruklu, S. Aslan, J. Saniie and F. M. Vallina, "Image and video processing platform for field programmable gate arrays using a high-level synthesis," in *IET Computers & Digital Techniques*, vol. 6, no. 6, pp. 414-425, November 2012.

[12] S. Niu, S. Wang, S. Aslan and J. Saniie, "Hardware and software design for QR Decomposition Recursive Least Square algorithm," *2013 IEEE 56th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Columbus, OH, 2013, pp. 117-120.