Hardware Acceleration of Matrix Computations for Wireless Communication Applications

Eugénio Filipe Nogueira Costa
AFA, Portuguese Air Force Academy, Portugal
IST, Technical University of Lisbon, Portugal
efcosta.af@emfa.pt

Abstract

Wireless communication applications for multi-antenna systems require the efficient computation of matrix inversion and decomposition problems. Matrix inversion is also a computationally intensive process. This paper presents an efficient and scalable architecture for the inversion of square matrices with complex elements. The design is based on the QR decomposition algorithm with modified Gram-Schmidt orthogonalization. The use of modified Gram-Schmidt makes this architecture numerical stable and accurate for well-conditioned non-singular matrices. An optimization technique is proposed, such that the algorithm is split in several stages and operations, which can be efficiently parallelized. The results obtained show that the proposed architecture is significantly faster than previously published FPGA implementations. The matrix inversion core can achieve a throughput of up to 0.34M updates per second, for 8 by 8 matrices of complex values, on a Xilinx Virtex-6 XC6VLX760 executing at 167MHz. Further, this new architecture is easily extendable for other matrix sizes.

1. Introduction

Matrix operations are common in wireless communication applications. The theme of this work concerns the specification, design and implementation of a dedicated multiprocessor for inversion of complex square 8x8 matrices in FPGA.

Due to its robustness regarding the selection of fading frequency, high spectral efficiency and low computational complexity, Orthogonal Frequency Division Multiplexing (OFDM) is one of the most promising wireless communication technologies, particularly in high data rates. Often used in conjunction with OFDM, Multiple Input Multiple Output (MIMO) systems improve the efficiency of wireless communication systems by using multiple antennas and receiving stations, resulting in an increased channel capacity and a reduction of inter-symbolic interference (ISI).

Every MIMO-OFDM receiver must equalize the received signal in order to remove the effect of the channel on the signal. The equalization process involves matrix inversion operations, and the size of the arrays, depends on the number of transmitter and receiver antennas [1] [2].

The high number of operations and high data rates involved impose stringent requirements for real time processing, such that the choice of the platform and design architecture become crucial factors.

Reconfigurable hardware systems can provide faster real-time applications for high data throughputs.

The main advantage of FPGAs is the ability to develop, test and implement the design in a reduced time. The cost of the development/fabrication in FPGA is also considerably lower than the ASIC.

The main objective of this thesis was to research, analyze, and evaluate matrix inversion algorithms, with subsequent specification and implementation of the best one. The adopted algorithm was first implemented in MATLAB and later specified in VHDL, synthesized and mapped for Xilinx Virtex-4, 5 and 6 FPGAs (the first one for demonstration purpose only).

The rest of this document is organized as follows. Section 2 offers a brief overview of wireless communications technology and matrix inversion algorithms. Section 3 presents an analysis of matrix inversion using the QR Decomposition method based on Gram-Schmidt orthogonalization (MGS). Section 4 presents an analysis of the most adequate arithmetic representation. In section 5, the MGS algorithm for 8x8 complex matrices is described, specified and optimized. Section 6 describes the design architecture. Section 7 presents the resources, performance and accuracy results. The conclusions are presented in section 8.

2. Multi-Antenna Systems and Matrix Inversion Algorithms

In the last decade the frequency spectrum has become a very precious resource and consequently, the spectrum reserved for wireless communications is very limited. Thus, the evolution in data rates is supported by advanced design techniques based on multi-antenna systems with computational complex algorithms. Multi-antenna systems consist of two or more antennas at the transmitter and/or receiver. There are two classes of multi-antenna systems: SMART and MIMO [13].

A SMART system consists of multiple antennas at the transmitter or receiver (multiple input single output (MISO) or single input-multiple output (SIMO)), where the signals from multiple antennas are combined, according to specific algorithms. This system can be used to increase the capacity and coverage of a mobile communications system (beamforming); to improve the quality of the channel (filtering) and to decrease the dispersion delay.

The beamforming is an important issue in wireless communications. To form the basic structure of a beamforming radar a linearly arranged array of antennas is required.
This technique is used to achieve the maximum reception capacity in a particular direction in the presence of noise. In the communication process, the signals from each antenna are multiplied by a set of complex coefficients and subsequently transmitted. The coefficients are used to estimate the arrival of a signal coming from a desired direction, while the remaining signs of the same frequency, but other directions are rejected (because of the different coefficients).

The coefficients are calculated iteratively, through algorithms like least mean squares (LMS), sample matrix inversion (SMI) or recursive least squares (RLS). The RLS is usually preferable because of its numeric properties (stability) and fast convergence. The RLS tries to find a set of coefficients that minimize the sum of the quadratic error

$$Ax = b + e$$

(1)

where $A$ is the matrix that contains the observed information with noise, $b$ is the training sequence and $x$ is the unknown coefficient that minimizes the error $e$ of each vector. The application of RLS can be seen in Eq (2).

$$\min (|b - Ax|)$$

(2)

where the calculation of $x$ implies a matrix inversion problem [13].

MIMO systems have multiple antennas at both transmitter ($N_t$) and receiver ($N_r$). The number of antennas doesn’t need to be the same in the transmitter and receiver however in this study we consider an equal number of antennas. MIMO systems have also the capability of spatial multiplexing. The spatial multiplexing improves the capacity of the channel to transmit simultaneously several parallel data streams, which are sent by various antennas as shown in the example in Figure 1.

![Figure 1 – Data stream multiplexed onto 4 antennas in a MIMO system.](image1)

However, the use of spatial multiplexing with high data rates makes the MIMO channels frequency selective. To resolve this problem MIMO systems are used in conjunction with OFDM technology, which transforms the channels on a set of flat frequencies, decreasing the complexity in the receiver.

In the transmitter, the data goes through: a process of modulation; spatial-time encoding; inverse Fast Fourier Transform (IFFT); cyclic prefix or guard interval in order to remove the inter-symbolic interference (ISI) and inter-carrier interference (ICI) [4]; and finally, data is transmitted by $N_t$ antennas. In the receiver, the data goes through: cyclic prefix removing process; FFT, space demultiplexing by channel estimation and finally, demodulation. The block diagram of the system is presented in Figure 2.

![Figure 2 – Block diagram of MIMO-OFDM communication system. Source: Chang et al. 2009 [3].](image2)

The equivalent model can be described in:

$$Y = HX + w$$

(3)

where $Y$ represents the complex matrix obtained on the receiver, $H$ represents the channel complex matrix, $X$ represents the transmitted modulated matrix, and $w$ represents the Gaussian white noise.

In order to estimate the transmitted data, Eq. (3) can be exploited as a least squares (LS) problem. The solution can be obtained using techniques such as Minimum Mean Square Error (MMSE) or zero forcing equalization. Each of these techniques requires matrix inversions [3] [6].

A typical MIMO-OFDM system needs to equalize matrices in less than 1 milisecond, which is the requirement for a real-time implementation [6] [11].

A. Matrix Inversion Methods

Matrix inversion methods for hardware implementation have generated a tremendous research interest. The inverse of a square matrix $A$ is represented by $A^{-1}$, and satisfies the condition

$$A \times A^{-1} = I,$$

(4)

where $I$ is the identity matrix. The explicit process of matrix inversion is a computationally intensive method, especially when the size of the matrix exceeds residual values (matrix not larger than 4).

For small sized matrices analytic method, such as direct analytical matrix inversion (using adjugate matrix) or blockwise analytic matrix inversion, are commonly used. The complexity of the mentioned methods grows very fast, with the increase of the matrix’s size [1] [7].

There are stable methods for matrix decomposition that simplify the computation process, even when the size of the matrices increases, such as, LU decomposition, Cholesky decomposition and QR Decomposition. The choice of the method depends on the characteristics of the matrix. However, the QR Decomposition provides greater stability and flexibility considering the different characteristics of matrices, including the ability to process $m \times n$ matrices, with $m \neq n$ [1][2]. QR Decomposition is also a usual resource in other matrix transformations and can therefore be reused, saving hardware area [13].

B. QR Decomposition

The QR decomposition can be applied to any type of matrix [1] and consists of the decomposition of a given matrix in two matrices, an orthogonal matrix and an upper triangular matrix. Thus the QR decomposition of a matrix $A$ is represented as the product of an orthogonal matrix ($Q$) by an upper triangular matrix ($R$):
\[ A = QR \]  
\[ Q^* = Q^T = I \]  
\[ Q^{-1} = Q^* \]  
\[ A^{-1} = R^{-1}Q^* \]  

The solution of the inversion problem is given by

Generalizing \( M_{m \times n}(\mathbb{R}) \) to \( M_{m \times n}(\mathbb{C}) \), \( Q \) is an unitary matrix with the following characteristics:

\[ Q^*Q = QQ^* = I \]  
\[ Q^{-1} = Q^* \]  

\( Q^* \) denotes the hermitian transpose of matrix \( Q \). The solution of the matrix inversion in the complex domain (\( \mathbb{C} \)) is:

\[ A^{-1} = R^{-1}Q^* \]  

This method has three distinct phases: the first one is the QR Decomposition, which has higher computational complexity, the second phase consists of the upper triangular matrix inversion \( R \) and the last phase consists of the multiplication of the inverted triangular matrix by the unitary hermitian transpose matrix \( Q^* \). Due to its triangular structure, the inversion of the matrix \( R \) is a relatively simple process. It is calculated by back substitution using the identity matrix:

\[ RR^{-1} = I \]  

The \( Q^* \) matrix does not require any calculation as it only requires memory manipulation [2]. Figure 3 shows the sequence of calculations for a generic matrix inversion based on QR Decomposition.

![Figure 3 – Stages of matrix inversion using QR decomposition.](image)

There are three main alternative methods for QR Decomposition: Classical Gram-Schmidt orthogonalization (CGS), Givens rotations (GR) and Householder Reflections (HR). The Modified Gram-Schmidt (MGS) is an alternative method based on CGS. It has been later developed in order to guarantee higher stability and numerical precision.

Considering the higher complexity of the method based on Householder reflections [3], the MGS and the GR are the methods that have recently received more attention for reconfigurable hardware implementation. Both can provide high performance, stability, precision and considerable reduction of hardware area.

However, when comparing the number of operations required by each method, for matrices of size \( m \times n \), the MGS stands out with a complexity of \( mn^2 \) whilst the GR has a \( 2mn^2(1 - \frac{n}{m}) \) complexity. For \( n \times n \) matrices, the number of MGS operations is 3/4 of number of operations of the GR. For this reason the MGS has a significant advantage both in terms of hardware area reduction and/or throughput improvement [3] [6].

### 3. Gram-Schmidt Orthogonalization

Gram-Schmidt orthogonalization is a direct method to obtain the \( Q \) and \( R \) matrices. Considering the following \( m \times n \) matrix in columns vector notation,

\[ X = \begin{bmatrix} X_{11} & \cdots & X_{1n} \\ \vdots & \ddots & \vdots \\ X_{m1} & \cdots & X_{mn} \end{bmatrix} = [x_1 \cdots x_n]. \]  

(11)

This method consists of an iterative process used to obtain an orthogonal base \( \{u_1, \ldots, u_m \} \) of a vector subspace \( \mathbb{R}^{m \times n} \) from the base \( \{a_1, \ldots, a_n \} \) of \( \mathbb{R}^{m \times n} \)

\[ u_1 = a_1 \]
\[ u_2 = a_2 - \text{proj}_{u_1} a_2 \]
\[ u_3 = a_3 - \text{proj}_{u_1} a_3 - \text{proj}_{u_2} a_3 \]
\[ \vdots \]
\[ u_m = a_n - \text{proj}_{u_1} a_n - \text{proj}_{u_2} a_n - \cdots - \text{proj}_{u_{m-1}} a_n \]

(12)

where \( \text{proj}_{u_1} a_i \) represents the \( a_i \) orthogonal projection of the vector subspace \( \mathbb{R}^{m \times n} \) of \( A \)

\[ \text{proj}_{u_1} a_i = \begin{cases} a_i & \text{if } u_i \cdot a_i > 0 \\ -a_i & \text{if } u_i \cdot a_i < 0 \end{cases} \]

(13)

and \( \langle x, y \rangle \) denotes the inner product of \( x \) by \( y \).

The CGS algorithm is shown in a compact form in Eq. (14) which allows an efficient memory implementation, due to its high parallelism level and because the columns of the matrix \( Q \) can be written over the original \( A \) matrix.

\[ q_k = \left( a_k - \sum_{i=1}^{k-1} r_{ik} q_i \right) / r_{kk} \]

(14)

where

\[ r_{kk} = q_k^T q_k \]

(15)

and

\[ r_{kk} = \| u_k \| \]

(16)

However, in the presence of round-off errors during the fixed precision computation, the orthogonality of \( Q \) cannot be ensured and sometimes can ever be completely lost. Because of its instability and low precision the Gram-Schmidt orthogonalization is not usually used in its classic version, but in another version called Modified Gram-Schmidt.

The MGS algorithm is obtained by applying small changes to the CGS algorithm. By swapping \( a_k \) by \( u_k \) in the linear combinations of \( q_k \), during the calculation process of the orthogonal vectors, the MGS can be made more stable (less susceptible to round-off errors) [6] [10].

QR decomposition based on the CGS and MGS algorithms are compared in Figure 4.

![Figure 4 – QR decomposition algorithms for \( n \times n \) matrices, based on: CGS (a) MGS (b). Font: Persson, 2007 [9](image)]
Each decomposition begins with the copy of matrix \( A \) to a temporary matrix \( U \) (inst.: 1). Each element of the \( R \) diagonal is obtained by the calculation of the euclidean norm of the \( U \) columns (inst.: 2):

\[
\begin{align*}
    r_{11} &= ||u_1|| = \sqrt{(u_{1_1})^2 + (u_{2_1})^2 + \cdots + (u_{m_1})^2} \\
    \vdots \\
    r_{nn} &= ||u_n|| = \sqrt{(u_{1_n})^2 + (u_{2_n})^2 + \cdots + (u_{m_n})^2}
\end{align*}
\]  

(17)

where \( u_i^k \) represents the \( i \) column of the temporary \( U \) matrix obtained in each iteration \( k \), since the \( U \) matrix is dynamic and updated at every iteration. The \( Q \) matrix results from the division of the \( U \) columns by its euclidean norm (inst.: 3):

\[
\begin{align*}
    q_{11} &= \frac{u_{1_1}}{r_{11}} \iff q_{12} = \frac{u_{1_2}}{r_{11}}, \quad q_{13} = \frac{u_{1_3}}{r_{11}}, \quad \cdots, \quad q_{1n} = \frac{u_{1n}}{r_{11}} \\
    q_{n1} &= \frac{u_{n_1}}{r_{nn}} \iff q_{n2} = \frac{u_{n_2}}{r_{nn}}, \quad q_{n3} = \frac{u_{n_3}}{r_{nn}}, \quad \cdots, \quad q_{nn} = \frac{u_{nn}}{r_{nn}}
\end{align*}
\]  

(18)

\( r_j \) with \( i \neq j \) are obtained by the inner product of \( Q \) columns by the temporary \( U \) columns (inst.: 4):

\[
\begin{align*}
    r_{ij} &= q_i u_j = q_{i1} u_{j1} + q_{i2} u_{j2} + q_{i3} u_{j3} + \cdots + q_{in} u_{jn}, \quad j \in \mathbb{N}, \quad 2 \leq j \leq n \\
    \vdots \\
    r_{n1} &= q_{n1} u_{n1} = q_{n1} u_{n1} \frac{u_{n2}}{r_{nn}}, \quad q_{n3} u_{n3} \frac{u_{n4}}{r_{nn}}, \quad \cdots, \quad q_{nn} u_{nn} \frac{u_{nn}}{r_{nn}}
\end{align*}
\]  

Finally, (inst.: 5) the \( u_{i+1}, \cdots, u_n \) columns of temporary U matrix are updated:

\[
\begin{align*}
    u_{i+1} &= a_i, \quad 1 \leq i \leq n, \\
    u_{n+1} &= u_{n} - r_{n,i} q_{i}, \\
    \vdots \\
    u_{n} &= u_{n-1} - r_{n,n-1} q_{n-1}
\end{align*}
\]  

(20)

When adapting the MGS algorithm for the complex domain matrix (C) we can do some simplifications in the algorithm. These changes are shown in Figure 5, where a comparison with the algorithm from Figure 4 b) is made.

\[
\begin{array}{ll}
\text{for } j = 1: n & \text{for } j = 1: n \\
\text{for } i = 1: n & \text{for } i = 1: n \\
\text{if } ||u_i|| & \text{if } i = 1: n \\
& r_i = ||u_i|| \\
& q_i = u_i/r_i \\
& r_{ij} = q_j^u_i \\
& u_j = u_j - r_{ij} q_i \\
\end{array}
\]  

(1)

\[
\begin{array}{ll}
\text{for } j = 1: n & \text{for } j = 1: n \\
\text{for } i = 1: n & \text{for } i = 1: n \\
\text{if } (i + 1): n & \text{if } (i + 1): n \\
\text{if } j = (i + 1): n & \text{if } j = (i + 1): n \\
& r_i = \sqrt{u_i^* u_i} \\
& q_i = u_i / r_i \\
& r_{ij} = q_j^u_i \\
& u_j = u_j - r_{ij} q_i \\
\end{array}
\]  

(2)

\( a \) \quad \text{b)}

Figure 5 – Comparison between the generic MGS (a) and its adaptation to complex domain (b).

The major changes can be seen in (inst.: 1), where the operation \( u_i^u \) produces a result with null imaginary part and positive real part and consequently the reciprocal and square root operations will have real positive operands. This will reduce the number of operations, the area of hardware and the propagated error.

The Inversion of the upper triangular matrix \( R \) is performed by back substitution, as presented in Figure 6.

\[
\begin{array}{l}
\text{for } j = 1: n \\
\text{for } i = 1: (j - 1) \\
\text{for } k = 1: (j - 1) \\
\text{for } k = 1: (j - 1) \\
\text{for } k = 1: (j - 1) \\
\end{array}
\]  

(1)

\[
\begin{array}{l}
w_{ij} = 1 / r_{ij} \\
w_{ij} = w_{ij} + w_{ik} a_{kj} \\
w_{ij} = -w_{ij}
\end{array}
\]  

(2)

(3)

\[
\begin{array}{l}
\end{array}
\]

Figure 6 – Upper triangular matrix \( R \) inversion algorithm. Font: Irturk et al. 2009 [1].

The inversion of \( R \) matrix is performed column-by-column (\( R^{-1} \) is shown as \( W \)). The diagonal elements \( w_{ii} \) are calculated in (inst.: 1) from \( \frac{1}{r_{ii}} \). The remaining elements of the matrix are obtained through an iterative process of multiplication and addition (inst.: 2), with later multiplication by \( w_{ij} \) (inst.: 3). Finally, a matrix multiplication is performed \((WQ')^*\) or, according to the changes identified in Figure 5 b), \((WQ)^*\).

4. Fixed-Point Arithmetic

There are two different types of numeric representation in binary notation: fixed-point and floating-point. The fixed-point arithmetic allows us to represent a limited and reduced range, but with absolute constant precision. Floating-point arithmetic uses an exponential scale factor which makes possible the representation of numbers in higher ranges. However, its accuracy is relative (it decreases with the increase of the represented value).

For the representation of known ranges, considering an equal number of bits, the use of floating-point may become too expensive and may lead to inefficient designs. The fixed-point implementation, with a priori knowledge of the ranges and based on a careful overflow analysis, allows stable and efficient hardware designs [11].

In this work, a two’s complement fixed-point format was adopted. In this notation, a binary word has a sign bit, an integer part and a fractional part.

The fixed-point format can be represented as \( A(N, p) \), where \( N \) is the word’s length and \( p \) is the number of the fractional part bits.

The range of a word in the format \( A(N, p) \) is

\[
[-\frac{2^{N-p}}{2^p} \cdots \frac{2^{N-p-1}}{2^p}]
\]

(21)

To obtain a number from the fixed-point format we can use the expression bellow

\[
R = \frac{1}{2^p} \sum_{i=0}^{N-2} 2^i B_{IT},
\]

(22)

where \( i \) is the bit index and \( BIT \) is its status (0/1).

In this work, the arithmetic operations required are: addition; subtraction; multiplication; division and square root. The addition and subtraction operations are simpler. The availability of embedded DSP blocks in today’s FPGA has also simplified the multiplication.
5. MGS Specification and Optimization

In order to understand and to optimize the algorithm and to know its data dependences, it was decomposed in three main stages and ten fundamental operations designated as $OP_i$, $i = 1, 2, ..., 10$. The architecture was defined for $8 \times 8$ matrices. Smaller matrices are processed using the same circuit structure, but require a lower number of operations.

In the first phase, the orthogonal $Q$ matrix, triangular matrix $R$ and the temporary matrices $U^k$, $k = \{2, 3, ..., 8\}$ are calculated. In the second phase the triangular $W$ matrix ($W = R^{-1}$) is obtained. In the third and final the $W$ matrix is multiplied by the $Q^T$ matrix.

Phase 1 consists of the following operations:

1. $OP_1$: column $U_1^i$ × column $U_1^j = a + jb$ (23)
2. $OP_2$: $\sqrt{a + jb} = (a' + jb') = R_{li}$ (24)
3. $OP_3$: $\frac{1}{a' + jb'} = (a'' + jb'')$ (25)
4. $OP_4$: $Q_{ki} = U_{k1}^i \times (a'' + jb'')$ (26)
5. $OP_5$: $R_{ki} = \text{column } Q_i \times \text{column } U_k^i$ (27)
6. $OP_6$: $(a + jb) = R_{ki} \times Q_{li}$ (28)
7. $OP_7$: $U_{l,k}^{i+1} = U_{l,k}^i - (a + jb)$ (29)

Phase 2 consists of the following operations:

1. $OP_8$: $W_{li} = \frac{1}{R_{li}}$ (30)
2. $OP_9$: $(a + jb) = \text{row } W_i \times \text{column } R_k$ (31)
3. $OP_{10}$: $W_{l,k} = -(a + jb) \times W_{l,k}$ (32)

Phase 3 consists of $OP_{10}$:

$$B_{i,k} = \text{row } W_i \times \text{row } Q_k$$ (33)

A. Architectural exploration

The design has gone through a process of architectural exploration, where three distinct architectural models have been used to analyze and specify the system.

The first model was essentially for functional analysis. The words length used was 32 bits in $A(32, 24)$ format, and most resources were shared. The sharing of resources has two drawbacks: low scalability and inability to implement pipelining. The 32 bit word length lead to two additional problems: excessive space in memory needed to allocate the temporary results and excessive number of DSPs to implement the multiplications. Data storage was implemented with 36 Kb dual port BRAMs.

Subsequently, considering the amount of BRAMs and DSPs consumed and the throughput of the first model, a second model was developed which uses dynamic word lengths with point position variable.

A study in MATLAB [21] was conducted to evaluate the most adequate fixed-point representation, considering minimization of the overflow errors and maximization of the accuracy, and area/performance requirements. Consequently, we adopted to use 18 and 26 bit word lengths to represent the matrices elements (26 bits for the $U$ matrix elements, and 18 bits for the others matrices elements).

Considering the intensive use of the vector multiplier and its latency (7 clock cycles), this operator was modified to make pipelining possible. To minimize the critical path, it was implemented using two pipeline stages in each DSP. Some redundancies in the recalculation of $W_{li}$ elements (in phase 3) were also eliminated, which removed 8 complex reciprocal operations.

The third model incorporated the final algorithmic changes based on the implementation of the hermitian transpose. Consequently, the reciprocal and square root were significantly simplified, because the operands became real ($\mathbb{R}$) and $OP_2$ and $OP_3$ were adjusted too. Indirectly, $OP_1$, $OP_6$ and $OP_{10}$ operations were also adjusted. Another modification was the implementation of control pipelining, by including registers in some control out paths, to minimize the critical path delays.

Figure 7 shows the block diagram of the final hardware architecture and Table 1 resumes the main aspects of the three developed models.
The implementation of s. 

s do not vary with the three percentage 

\[ 8 \times 8 \] matrix also decreased (less than half pipelining, the num critical path reduction. 

\[ 8 \times 8 \] also made possible the adequate values bt 

According to the absolute maximum values 

XC5VLX330T device 

The simulations was performed to evaluate the overflow errors and analyze the results accuracy. These simulations allowed us to estimate the range result values of each operation \( OP_i \) and to define the most adequate word point position. These tests were performed over a sample of 5000 complex 8x8 matrices. Table 2 shows the distribution of the results according to the absolute maximum values obtained for each operation.

### Table 1 – Comparison between the tree models results.

<table>
<thead>
<tr>
<th>Model</th>
<th>Flip Flops</th>
<th>LUTs</th>
<th>DSP48E1s</th>
<th>BRAMs</th>
<th>Clock cycles</th>
<th>Critical path (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1311</td>
<td>7576</td>
<td>130</td>
<td>27</td>
<td>1077</td>
<td>22.231</td>
</tr>
<tr>
<td>2</td>
<td>6464</td>
<td>9332</td>
<td>47</td>
<td>19</td>
<td>571</td>
<td>7.682</td>
</tr>
<tr>
<td>3</td>
<td>5776</td>
<td>8748</td>
<td>36</td>
<td>19</td>
<td>485</td>
<td>6.155</td>
</tr>
</tbody>
</table>

All values were obtained under the same conditions for a Xilinx XC5VLX330T device [17] [18]. The main difference is the number of DSPs and consequently the critical path reduction. Thanks to the implementation of pipelining, the number of clock cycles needed to process a complex 8x8 matrix also decreased (less than half of the initial number).

### B. Overflow Error Analysis

A set of Matlab simulations was performed to evaluate the overflow errors and analyze the results accuracy. These simulations allowed us to estimate the range result values of each operation \( OP_i \) and to define the most adequate word point position. These tests were performed over a sample of 5000 complex 8x8 matrices. Table 2 shows the distribution of the results according to the absolute maximum values obtained for each operation.

### Table 2 – Distribution of tests according to the absolute maximum values obtained for each operation results.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>5%</th>
<th>1%</th>
<th>0.1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>A(26,24)</td>
<td>A(26,24)</td>
<td>A(26,24)</td>
</tr>
<tr>
<td>Q</td>
<td>A(18,17)</td>
<td>A(18,17)</td>
<td>A(18,17)</td>
</tr>
<tr>
<td>R</td>
<td>A(18,15)</td>
<td>A(18,15)</td>
<td>A(18,15)</td>
</tr>
<tr>
<td>W</td>
<td>A(18,14)</td>
<td>A(18,13)</td>
<td>A(18,12)</td>
</tr>
<tr>
<td>B</td>
<td>A(18,14)</td>
<td>A(18,13)</td>
<td>A(18,12)</td>
</tr>
</tbody>
</table>

Considering the results obtained, and the relevance of the amount of fractional bits in each operation to the final accuracy, we defined three levels of overflow error risk (to be considered in the system implementation): 0.1%, 1% and 5%. Analyzing the occurrences, we calculated that the difference in the levels of overflow error risk affects only the representation format of the \( W \) and \( B \) matrices elements. The resulting values of the operations that affect the elements of the other matrices imply always a fixed representation format, because their absolute maximum values do not vary with the three percentage ranges considered. The format of binary words used to represent the elements of each matrix is given in Table 3.

### Table 3 – Binary words length and format used to matrix representation.

<table>
<thead>
<tr>
<th>Matrix</th>
<th>5%</th>
<th>1%</th>
<th>0.1%</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>A(26,24)</td>
<td>A(26,24)</td>
<td>A(26,24)</td>
</tr>
<tr>
<td>Q</td>
<td>A(18,17)</td>
<td>A(18,17)</td>
<td>A(18,17)</td>
</tr>
<tr>
<td>R</td>
<td>A(18,15)</td>
<td>A(18,15)</td>
<td>A(18,15)</td>
</tr>
<tr>
<td>W</td>
<td>A(18,14)</td>
<td>A(18,13)</td>
<td>A(18,12)</td>
</tr>
<tr>
<td>B</td>
<td>A(18,14)</td>
<td>A(18,13)</td>
<td>A(18,12)</td>
</tr>
</tbody>
</table>

### 6. Design Architecture

The datapath unit includes five main arithmetic units: a vector subtractor unit; a single multiplier unit; a vector multiplier unit; a reciprocal and a square root units. All design units work with the same clock frequency and all the registers are edge-triggered positive. There is a single control unit based on a Moore machine.

The data storage units are based on 36 Kb dual port BRAMs and they host all matrices involved in the process.

The vector subtraction unit was designed to subtract complex vectors, with 8 or less elements (with 26 bit word length). The unit has 16 arithmetic subtractors and 16 output registers.

The single multiplication unit was designed to perform the multiplication of complex values by real scalars. In order to be able to perform multiplications with one DSP and one additional LUT, the complex values have 26 bits size for each word (real and imaginary part), and the real scalar words have 19 bits. The result is truncated to 36 bits. The DSPs of multipliers have two pipeline

![Figure 7 – Block diagram of final architecture model.](image-url)
stages. The internal cell block diagram of the single multiplier unit is presented in Figure 8.

Figure 8 – Internal cell block diagram of single multiplier unit.

The vector multiplier unit was designed to provide two types of operations: the multiplication between complex vectors and multiplications of complex vectors by complex scalars. It uses a single DSP with two pipeline stages per multiplier, the first operand has 26 bits, the second operand has 19 bits and the result is truncated to 36 bits per word.

The vector multiplier unit is split in 7 pipeline stages and can handle vectors with 8 elements. The result of the multiplication of vectors by scalars is available at the end of the fourth stage. The resources used in the vector multiplier unit are indicated in Table 4.

Table 4 – Resources used by the vector multiplier unit.

<table>
<thead>
<tr>
<th>DSPs</th>
<th>ADD/SUBs</th>
<th>ADDs</th>
<th>REGs</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>16</td>
<td>14</td>
<td>26bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>19bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>45bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>36bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>

The reciprocal unit \( \left( \frac{1}{x} \right) \) was designed to process operands with 32-bit word length. It uses a polynomial 1st order approximation, with \( y_0 \) and \( m \) coefficients stored in a look up table (36Kb BRAM).

\[
\frac{1}{x} = y_0 + m(x - x_0)
\]  (34)

This unit has a latency of 8 clock cycles and includes a starting signal to control the result storage during phase 1. The adopted method estimates the reciprocal function solution for an \([x_0, x_1]\) interval, where the function is differentiable. We chose the minimax polynomial approximation, because it presents the smallest maximum approximation error in the range [15]. The storage coefficients were calculated for the normalized interval \([0.5, 1]\). Figure 9 shows its block diagram.

Figure 9 – Internal cell block diagram of reciprocal unit.

The square root unit \( \sqrt{x} \) was designed to process operands with 32-bit word length, and it also uses a polynomial 1st order minimax approximation. This unit uses two look up tables (two 36Kb BRAM) because of the exponential behavior (2\(x \) scaling factor) of the square root function. Table 1 stores the approximation coefficients with scale factors with odd exponents (2, 8, 32, 128, …) while table 2 stores the approximation coefficients with scale factors with even exponents (4, 16, 64, 256, …).

7. Results

The HDL software used in this project was the Xilinx Integrated Software Environment (ISE) 13.2 [16]. The target FPGA devices were the Virtex-4 XC4VLX25 (speed grade of -12), the Virtex-5 XC5VLX330T (speed grade of -2) and the Virtex-6 XC6VLX760 (speed grade of -2).

For the Virtex-4 implementation and because of the device resource limitations a few changes had to be made to the circuit. The 36Kb BRAMs were replaced by 18 Kb BRAMS (in some cases used in parallel) and the DSP48Es were replaced by DSP48Es and LUTs [19][20].

Table 5 summarizes the resources occupation. Table 6 presents the time results based on the critical path analysis, after place & route. Table 7 shows the number of clock cycles needed to process different matrix sizes.

Table 5 – Resources consumption for 8x8 complex matrix inversion core on a Virtex-5 FPGA.

<table>
<thead>
<tr>
<th>Vector Subtractor Unit</th>
<th>Flip-Flops</th>
<th>LUTs</th>
<th>BRAMS</th>
<th>DSP48Es</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flip-Flops</td>
<td>0</td>
<td>896</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Single Multiplier Unit</td>
<td>161</td>
<td>209</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Vector Multiplier Unit</td>
<td>4585</td>
<td>6008</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>Reciprocal Unit</td>
<td>233</td>
<td>393</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SQRT Unit</td>
<td>234</td>
<td>467</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Total (processor)</td>
<td>5776</td>
<td>9918</td>
<td>19</td>
<td>36</td>
</tr>
</tbody>
</table>

Table 6 – Time results for the different devices.

<table>
<thead>
<tr>
<th></th>
<th>Virtex-4</th>
<th>Virtex-5</th>
<th>Virtex-6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical path (ns)</td>
<td>8.37</td>
<td>6.23</td>
<td>5.98</td>
</tr>
<tr>
<td>Clock Frequency (MHz)</td>
<td>119.5</td>
<td>162.5</td>
<td>167.1</td>
</tr>
<tr>
<td>Total time (µs)(^1)</td>
<td>4.06</td>
<td>3.02</td>
<td>2.90</td>
</tr>
</tbody>
</table>

Table 7 – Clock cycles needed for different matrix sizes.

<table>
<thead>
<tr>
<th>Matrix size</th>
<th>Clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x2</td>
<td>120</td>
</tr>
<tr>
<td>3x3</td>
<td>178</td>
</tr>
<tr>
<td>4x4</td>
<td>236</td>
</tr>
<tr>
<td>5x5</td>
<td>298</td>
</tr>
<tr>
<td>6x6</td>
<td>360</td>
</tr>
<tr>
<td>7x7</td>
<td>422</td>
</tr>
<tr>
<td>8x8</td>
<td>485</td>
</tr>
</tbody>
</table>

\(^1\) For 8 by 8 matrices

The results show that our core uses 5776 flip-flops, 9918 LUTs, 19 BRAMS (16 for matrix storages) and 36 DSPs (32 for the vector multiplication unit). The maximum frequency estimated after place & route is 120 MHz, 162
MHz and 167MHz, for Virtex-4, Virtex-5 and Virtex-6, respectively. The number of clock cycles has a linear growth with the matrix sizes (≈ 60 clock cycles/number of matrix columns).

Table 8 presents the comparison with other works (4 by 4 matrix cores implemented on Virtex-4 FPGA). Some factors have been considered, such as the different arithmetic, real or complex domain and matrix sizes supported.

Table 8 – Comparison between other works and this work results.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>Skys</th>
<th>DSP4Es</th>
<th>BRAMs</th>
<th>Total (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>This work</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtex6 - 8x8</td>
<td>2575</td>
<td>36²</td>
<td>19</td>
<td>2.90</td>
</tr>
<tr>
<td>Virtex6 - 4x4</td>
<td></td>
<td></td>
<td></td>
<td>1.41</td>
</tr>
<tr>
<td>Virtex5 - 8x8</td>
<td>2788</td>
<td></td>
<td></td>
<td>3.02</td>
</tr>
<tr>
<td>Virtex5 - 4x4</td>
<td></td>
<td></td>
<td></td>
<td>1.47</td>
</tr>
<tr>
<td>Virtex4 - 8x8</td>
<td>8108</td>
<td>48</td>
<td>22</td>
<td>4.06</td>
</tr>
<tr>
<td>Virtex4 - 4x4</td>
<td></td>
<td></td>
<td></td>
<td>1.98</td>
</tr>
<tr>
<td>Irturk et al. 2009 [1]</td>
<td>19 bits (R)</td>
<td></td>
<td></td>
<td>2415 (MGS)</td>
</tr>
<tr>
<td></td>
<td>26 bits (R)</td>
<td></td>
<td></td>
<td>4656 (LU)</td>
</tr>
<tr>
<td></td>
<td>32 bits (R)</td>
<td></td>
<td></td>
<td>6540</td>
</tr>
<tr>
<td>Irturk et al. 2008 [4]</td>
<td>20 bits (R)</td>
<td></td>
<td></td>
<td>11644 (QR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12 (Cholesky)</td>
</tr>
<tr>
<td>Karkooti &amp; Cavallaro 2006 [2]</td>
<td>20 bits (C)</td>
<td></td>
<td></td>
<td>9117 (GR)</td>
</tr>
<tr>
<td>Eilert et al. 2007 [7]</td>
<td>16 bits (R)</td>
<td></td>
<td></td>
<td>1561 (Adjugate Matrix)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 (NR³)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1716 (Blockwise Decomposition)</td>
</tr>
</tbody>
</table>

Although the results indicate that Eilert et al. [7] presents the best execution time, their work only supports 4 by 4 matrices and only for real elements. The execution times shown for our implementation (2.9 µs for matrices of order 8 in XC6VLX760 device) are better than all other.

Compared to prior implementations and its limitations, (4x4 matrices) the small increase in resources occupation of our design is fully acceptable. The most significant difference is the increase in the number of BRAMs. The low number of slices is a consequence of the higher use of DSP and RAM blocks.

Our design is easily extendable to bigger matrix sizes. However, there are costs associated to resources occupation and to clock cycles (60 per matrix dimension increase). Table 9 presents the resource occupation growth with the increase of the matrix size.

Table 9 – Resource occupation growth with the increment of one column/row in the matrix size.

<table>
<thead>
<tr>
<th>DSP4Es</th>
<th>Add 36b</th>
<th>Sub 26b</th>
<th>Reg 45b</th>
<th>Reg 36b</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

² DSP4Es
³ Not Reported

We compared the ISE simulator (behavioral) results (X) with the reference values (X') (calculated by inv(MAT) function of MATLAB with double floating-point precision). The errors are denoted as ε = X - X' and the absolute value of the difference between actual and computed results is denoted as εabs = |X - X'|. However, the absolute error can be misleading. The relative error is a better choice, it is denoted as εrel = |X - X'| / |X'|. These errors were calculated for the absolute, real and imaginary parts of the results.

Five hundred 8 by 8 matrices were tested for all levels of overflow error risk (0.1%, 1% and 5%). The error distributions are shown in Figure 10, Figure 11 and Figure 12.

Figure 10 – Comparison between 0.1%, 1% and 5% of absolute relative overflow error.

Figure 11 - Comparison between 0.1%, 1% and 5% of real part relative overflow error.

Figure 12 - Comparison between 0.1%, 1% and 5% of imaginary part relative overflow error.

The results show that the 1% and 5% curves median are very close, however the 5% curves mean error is high. High mean error indicates a large number of overflow cases and high overflow errors (larger than 100%). Therefore and although the design can be synthesized with
any of the three risk levels, the 1% overflow error risk curves show that this option is the one that has the best performance.

7. Conclusion

A scalable pipeline architecture to compute 8 by 8 matrix inversions on XC4VLX25, XC5VLX330T and XC6VLX760 FPGAs was designed and implemented. We used QR decomposition based on Modified Gram-Schmidt Algorithm, embedded processing elements and fixed-point representation. A detailed error analysis was performed for overflow and accuracy issues. The word length is not fixed, however we use always 18 and 26 bits for BRAMs storage (because of the DSP48Es proprieties).

The design can execute with a maximum clock rate of 120MHz, 162MHz, and 167MHz on XC4VLX25, XC5VLX330T and XC6VLX760, respectively, and is able to achieve a throughput between 0.25M and 0.34M updates per second for 8 by 8 complex matrix inversions.

This design is easily extendable to bigger matrix sizes, and area/performance estimates for these have been also presented.

References


