FPGA implementation of Singular Value Decomposition

Karthikeyan N
Department of Electrical Engineering
Indian Institute of Technology Madras
Chennai - 600036, India
ee12s008@ee.iitm.ac.in

Sathyanarayanan S
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai – 600036, India
sathya@cse.iitm.ac.in

Vamsi Krishna S
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai - 600036, India
vamc@cse.iitm.ac.in

Shankar Balachandran
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai – 600036, India
shankar@cse.iitm.ac.in

***Abstract*—Singular value decomposition has been used in signal processing, image processing, principal component analysis, robotics and my other real time applications. These applications demand fast processing of large datasets. SVD needs large amount of computation. In this paper, we present the parallel implementation of Singular Value Decomposition in FGPA. SVD is implemented using two sided Jacobi algorithm to attain parallel systolic array architecture. The diagonal processing elements compute the rotation parameter and propagate to other off-diagonal processing elements in row and column for row and column operations respectively. The CORDIC algorithm is used in each processing element and optimized to have one bit rotation parameter which reduces the hardware requirement and routing overhead. The design has been tested and verified in Xilinx Spartan-6 LX45 FPGA for 4x4 asymmetric matrix.**

Keywords—SVD; Jacobi algorithm; FPGA; CORDIC

I. INTRODUCTION

Singular Value Decomposition is a matrix factorization which decomposes any MxN rectangular matrix to a MxM orthogonal matrix, MxN diagonal matrix and NxN orthogonal matrix. MxN diagonal matrix contains the singular values of the rectangular matrix. SVD decomposition equation can be written as in

$$A = U \Sigma V^T \quad (1)$$

SVD is used in many applications such as Low rank approximation, Image Compression, Estimation & Inversion, pseudo inverse, principal component analysis etc. In many cases in real world applications, SVD computation needs to be done in real time, which demands a fast method for SVD computation. Parallel algorithms always fast in computing SVD. Implementing such parallel algorithm in FPGA will be suitable for real time applications. The paper is organized as follows. Section II discusses objective and requirements of the design. Section III discusses about possible applications and an example. Section IV discusses about choice of approach and algorithm explanation. Section V explains the testing strategy used and the testing results. Section VI gives the insight of hardware used in the design. We conclude the paper after giving timing information of the design.

II. OBJECTIVE

To implement Singular Value Decomposition (SVD) of a real matrix in FPGA with following requirements

1) The design should run at minimum frequency of 50MHz.

2) Once the input is loaded into the BRAM module and run signal is raised, the design takes 1000 clock cycles to complete the operation and load the values into BRAM module.

3) The design computes SVD for symmetric matrices of maximum dimension 4X4 with each element as 16-bit.

III. APPLICATIONS

A. Possible Applications

Hardware implementation of SVD is used in real time processing for Image processing applications such as face recognition, motion detection etc., Analysis of the kinematic and dynamic characteristics of robotic manipulators [3], Signal processing applications, mainly least squares type problems, Computation of pseudo inverse, MIMO communication systems.

Such real-time applications need a hardware implementation of SVD. For fast computation of SVD, we go for parallel implementation approach in hardware.

B. Target Example

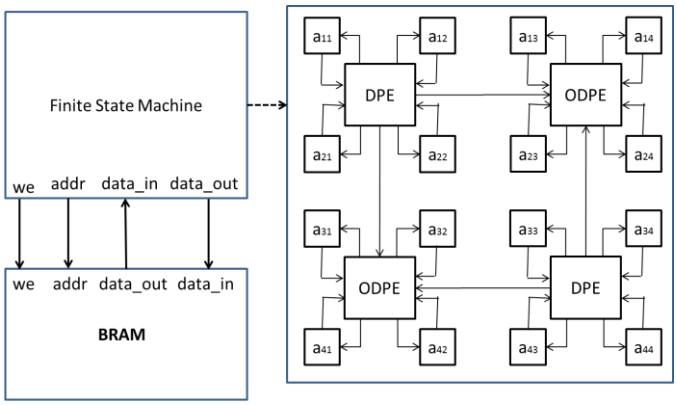
SVD is used in MIMO channel communication systems. MIMO systems offer increased capacity and data rate without any increase in bandwidth or power. Paper [4] shows how SVD can be used in MIMO systems to decompose a MIMO channel into parallel sub-channels. So the computation speed for SVD should be very high in those real-time systems. We cannot go

for a Parallel computer based approach because often the communication system may be mobile. Hardware, on-board solution is needed which can be achieved by implementing parallel computation of SVD in FPGA or ASIC.

IV. Design Strategy

A. Choice of Approach

This project aims at implementing SVD calculation algorithm which is suited for FPGAs. There are many methods like Jacobi method; QR method and Hestenes (one-sided rotation) method are available for calculating SVD values. But, Jacobi algorithm is preferred because of its simplicity and the parallelism that can be exploited. This implementation aims at improving the speed of the traditional Jacobi algorithm by exploiting parallelism available. Initially, Singular Value Decomposition problem is reduced to symmetric Eigen Value Decomposition problem. After reduction, two sided Jacobi rotation method is used to find EVD. Jacobi algorithm is preferred because of its simplicity, regularity and local communication.



DPE – Diagonal Processing Element
ODPE – Off Diagonal Processing Element

Systolic Array Architecture for 4 X 4 matrix

B. Implementation Methodology

In this method, given NxN matrix is divided into N/2 2x2 sub problems and solved. This method uses CORDIC algorithm for doing Jacobi rotations. Jacobi Algorithm for Symmetric Eigen value computation is explained as follows.

In two sided rotation Jacobi SVD algorithm, the matrix A is diagonalised by multiplying by left sided rotation matrix J(p, q, θ)' and right sided rotation matrix J(p, q, θ) in

$$A = J(p,q,\theta)' * A * J(p,q,\theta) \quad (2)$$

for all (p,q) combinations, p,q ϵ 1 to N

Multiplication by left rotation matrix only modifies p^{th} and q^{th} rows of matrix A. Similarly, multiplication by right rotation matrix only modifies p^{th} and q^{th} columns. The remaining values remain unchanged. This provides an opportunity to apply two Jacobi rotations in parallel for two non-overlapping values of (p,q).

This allows us to perform many row operations (left rotation) in parallel and also for columns. Thus, this parallel algorithm is suitable for hardware implementation. For matrix,

$$\begin{pmatrix} A(p,p) & A(p,q) \\ A(q,p) & A(q,q) \end{pmatrix} = \begin{pmatrix} a & b \\ b & d \end{pmatrix}$$

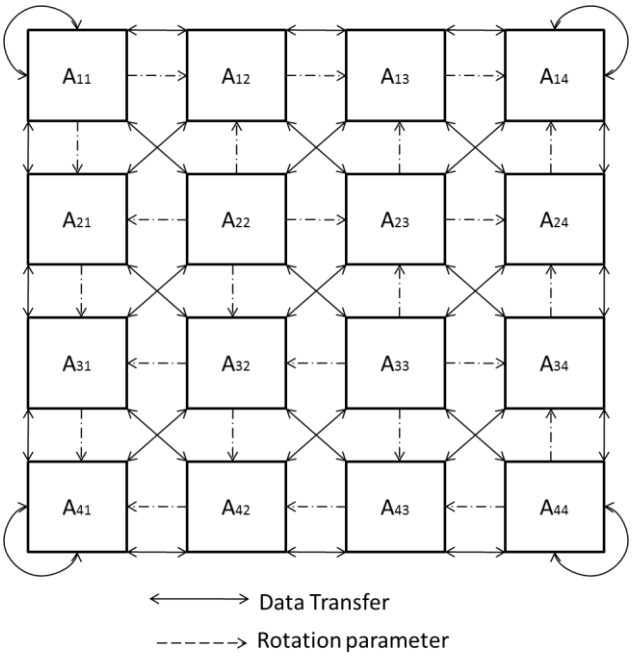
θ is computed as follows,

$$\theta = \frac{1}{2} tan^{-1}\left(\frac{2 * b}{d - a}\right) \quad (3)$$

This θ can be propagated to the (p,q) rows of matrix A for left rotation and same is done for right rotation (columns). If we have NxN matrix, we can do left rotation & right rotations for n/2 (p,q) combinations in parallel. It takes two cycles (one for left rotation and another for right rotation). To cover all p,q combinations, it takes (n-1) left and right rotations. So, over all operation take (n-1)x(2)x(n/2) cycles for complete Jacobi SVD algorithm. This complete cycle is called one sweep. Jacobi algorithm may need 3 to 6 sweeps in order to get correct SVD values.

In our design, we designed a 2-D systolic array architecture of processing elements to compute SVD using 2x2 SVD problems by two sided Jacobi rotation algorithm.

C. Systolic Array Architecture



Systolic array architecture for 8 X 8 matrix with data interchange

Square mesh connected array architecture is used to compute the SVD of N X N matrix. The N X N matrix is divided into (N/2)*(N/2) 2 X 2 matrices. A processing element (PE) is allocated to each 2 X 2 sub-matrix. Each element of matrix is stored in a 16-bit register. The PEs can be classified into diagonal PEs and off-diagonal PEs. The sub-matrices are interconnected by input and output lines for transmitting

rotation parameters. The diagonal PEs calculate θ and propagate the rotation parameters to the corresponding two rows (p,q) and two columns. The off-diagonal PEs get the rotation parameters from corresponding diagonal PE and apply the rotation to the 2x2 matrix. During left rotation, the off-diagonal PE receives rotation parameter from diagonal PE in the row and similarly during right rotation, it receives rotation parameters form diagonal PE in the column.

After left iteration, the rotated matrix values are stored back to registers. After right rotation, the rotated matrix values are subjected to interchange and then stored to registers. This interchange is needed in order to get all (p,q) combinations. The interchange between the PEs is shown in the above diagram. Data interchange is directly coded into State machine of the Systolic array in Verilog code. The Processing elements perform the rotation operation on the matrix elements. Two strategies for implementation of PE are considered.

D. θ computation based strategy

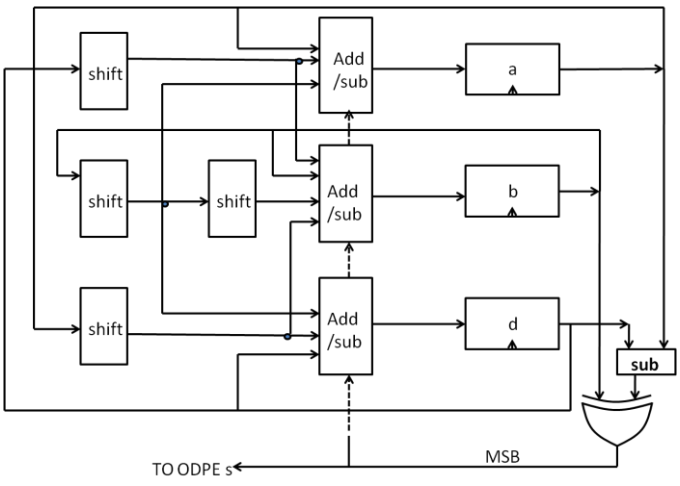
In this strategy, diagonal PE calculates the θ using CORDIC algorithm. CORDIC (CO-ordination Rotation Digital Computer) can be implemented to compute $\tan^{-1}(y/x)$. y can be given as 2*b and x can be provided as (d-a). The θ will be stored in another 16 bit register. Then, θ is propagated to corresponding p,q rows/columns and the rotation is applied to matrix elements using CORDIC rotation. So, the diagonal PE needs one CORDIC vector module and one CORDIC rotation module and one 16-bit register for holding θ.

E. Direct rotation strategy

In the above method, the CORDIC will compute θ based on direction of rotation. Similarly, during CORDIC rotation mode, direction of rotation is generated back again from θ. Instead of computing θ, propagating θ and then again compute direction of rotation in CORDIC rotation module, we can directly give direction to off-diagonal PEs. Main advantage is that, Diagonal PE does not need extra register for storing θ. When the θ based strategy is used for large NxN matrix, the routing for propagation of θ to each off-diagonal PEs becomes more complex because it needs 16 bit wires to be routed to each PE in row and column. In this method, we are using only one bit to be propagated to each PE, so it reduces the routing complexity and also increases speed of computation. In θ based method, θ computation and rotation will need 2 cycles. In this method, rotation needs only 1 cycle which reduces the computation time.

F. Diagonal PE

Diagonal PE will diagonalise the matrix in one cycle instead of applying two rotations. During diagonalisation, the direction of rotation is propagated to the off-diagonal PEs. During left rotation, the matrix is diagonalised, the direction of rotation is propagated to rows and the diagonalised values are not stored. Again the same operation is done for right rotation, the direction is propagated to columns and the diagonalised values are subjected to interchange and stored to registers. The matrix diagonalisation is implemented using CORDIC algorithm. In this algorithm, since it does not need θ computation, it does not need any ROM.



Diagonal PE Block Diagram

Jacobi rotation matrix can be written as

$$\begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix} = \cos(\theta)\begin{pmatrix} 1 & \tan(\theta) \\ -\tan(\theta) & 1 \end{pmatrix} \quad (4)$$

Diagonalisation can be written as

$$cos^2(\theta)\begin{pmatrix} 1 & -\tan(\theta) \\ \tan(\theta) & 1 \end{pmatrix}\begin{pmatrix} a & b \\ b & d \end{pmatrix}\begin{pmatrix} 1 & \tan(\theta) \\ -\tan(\theta) & 1 \end{pmatrix} \quad (5)$$

$$= 0.3687\begin{pmatrix} a - 2sb + s^2 d & b - sd + sa - s^2 b \\ b - sd + sa - s^2 b & d + 2sb + s^2 a \end{pmatrix} \quad (6)$$

Where s = tan(θ) is implemented using CORDIC algorithm. In CORDIC algorithm, this above matrix operation is repeated for $s = 2^{-i}$, for i = 0 to 15. This can be written in Verilog code as

```
sb2 = ((b0>>>i)<<<1);
s2a = ((a0>>>i)>>>i);
s2b = ((b0>>>i)>>>i);
s2d = ((d0>>>i)>>>i);
sa  = (a0>>>i);
sd  = (d0>>>i);
                if(dir == 1'b1) begin
a0 = a0 + sb2 + s2d;
b0 = b0 - sa - s2b + sd ;
c0 = c0 - sa - s2b + sd ;
d0 = d0 - sb2 + s2a;
end
else begin
a0 = a0 - sb2  + s2d;
b0 = b0 + sa - s2b - sd ;
```

c0 = c0 + sa - s2b - sd ;

d0 = d0 + sb2 + s2a;

end

In CORDIC algorithm, this $\cos^2(\theta)$ term is implemented using hardware multiplier of constant value 0.3687. During implementation of hardware multiplier for 0.3687, if we truncate the result, the SVD values are reducing in each iteration. So, we implemented rounding and the SVD values were not reducing and the values stabilized.

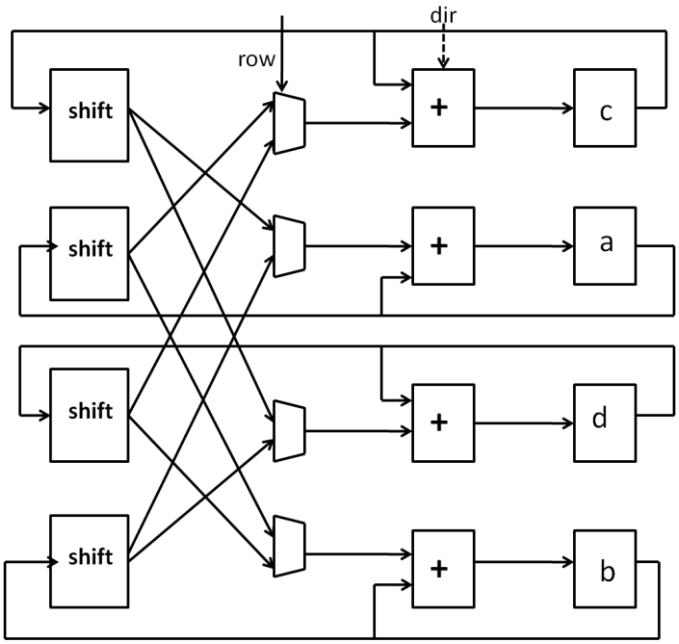
Diagonal PE has the following inputs, four 16 bit numbers of 2x2 matrix and start signal. Once the start signal goes high, the CORDIC rotation starts from i=0 to 15. After 16 clock cycles, the output done is set to high and the diagonalised values are given at output. During rotation, the rotation direction is set to 0 if it is positive rotation ($+2^{-i}$), & set to 1 if it is negative rotation (-2^{-i}).

The direction of rotation (i.e. $+2^{-i}$ or -2^{-i}), is determined by tan(θ)'s sign. Instead of finding (2*b)/(d-a) and then taking only sign bit, we use following simple logic operation to compute sign of tan(θ).

*assign numerator=((d0-a0)^(2*b0))<0?-1:0;*

MSB of numerator signal will give the sign of tan(θ). 2*b0 is simple arithmetic shift left operation; d0-a0 requires a subtractor. This simplified computation of direction of rotation, requires only less hardware.

G. Non-diagonal PE



Non Diagonal PE Block Diagram

Non-Diagonal PE has the following inputs, four 16 bit numbers of 2x2 matrix, row, direction-i and direction-j. The outputs are four 16-bit numbers and done signal. In non-diagonal PE, the rotation has to come from either row/column diagonal PEs. During row operation, row signal is kept high by the systolic array State machine, and during column operation, row signal is held low. Based on the row signal, the direction for rotation is chosen as either direction-i (row) or direction-j (column). After 16 clock cycles, the output done is triggered high. After each rotation, the output values are stored back to registers in systolic array.

Off-diagonal PE does the following row operation for $\begin{pmatrix} a & b \\ b & d \end{pmatrix}$ matrix as

$$0.6073 * \begin{pmatrix} a - sc & b - sd \\ c + sa & d + sb \end{pmatrix} \quad (7)$$

Similarly, for column operation is done as

$$0.6073 * \begin{pmatrix} a - sb & b + sa \\ c - sd & d + sc \end{pmatrix} \quad (8)$$

Multiplication by 0.6073 is done using hardware multiplier with rounding. The rounding is done by adding last bit to the previous bit.

This block contains 4 adder/subtractors, 4 barrel shifters of size 16, and a hardware constant multiplier implemented using shift and add operation and several MUXes.

H. State machine for Systolic array

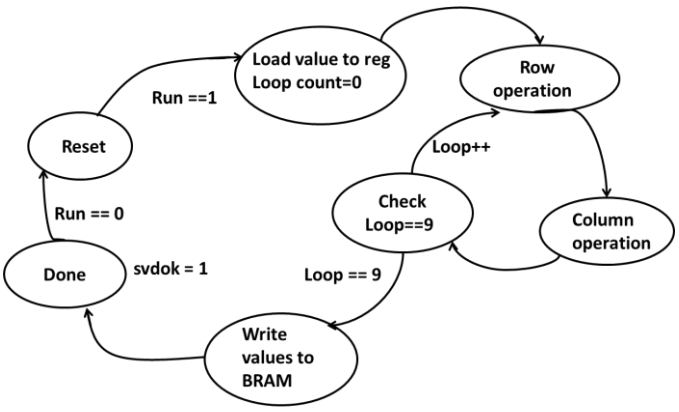
This State machine takes care of mainly three operations,

1) Loading the values of matrix from BRAM to registers.

2) Perform SVD operations (3 sweeps)

3) Writing computed SVD values back to BRAM.

During loading values from BRAM to registers, State machine will generate address to be read (0 to 15) and store it in corresponding registers in systolic array. During SVD operations, State machine gives the signals whether it is row or column operation to PEs and also it will interchange the values after each column operations before storing the values. For each sweep, state machine performs (n-1) pairs of row and column rotations and it performs 3 sweeps.



Systolic Array State Diagram

During writing SVD values to BRAM, State machine generated the respective address and asserts write enable to BRAM. After writing all the values from registers of systolic array to BRAM, it asserts the output signal svdok to high.

The design is implemented in FPGA board and tested with 4X4 symmetric matrix. The hardware takes 384 clock cycles to compute SVD. The values are written back to BRAM for 4X4 symmetric matrix.

The design take 24 clock cycles for loading the initial matrix values from BRAM to systolic array. Writing computed SVD values to BRAM takes 18 clock cycles. Computation of SVD values takes 342 cycles.

Each rotation (row/column) takes 19 cycles distributed as, 1 cycle for reading values from registers into CORDIC module, 16 cycles for CORDIC rotation, and 2 cycles for writing rotated values into registers.

1 sweep needs = 19x2x3 = 114 cycles.

3 sweeps needs = 3x114 = 342 cycles.

The error ranges are generally low unless the SVD values are too low (<2) or input values are high such that overflow occurs during CORDIC operation. In order to reduce errors due to truncation, we increased the internal hardware of the CORDIC to work in 32 bit arithmetic. But the no of clock cycles needed is 16 for CORDIC. We verified the results and the values are same as the simulation results.

I. Non-symmetric case

Every 2x2 non-symmetric matrix can be converted to symmetric matrix and EVD algorithm can be applied for the resulting matrix to get the SVD values. Hence, symmetric SVD algorithm can be extended to do generalized SVD computations. This 2x2 symmetrisation requires 16 cycles and the corresponding rotation parameter is applied for row off-diagonal PEs for rotation.

Steps for Symmetrisation is

1) Find θ for (p,q) diagonal 2x2 matrix

$$\theta = tan^{-1}\left(\frac{b-c}{d+a}\right) \tag{9}$$

2) Apply the transformation A = J(p,q, θ) ' * A
Diagonal PE does the following symmetrisation operation for $\begin{pmatrix} a & b \\ b & d \end{pmatrix}$ matrix as

$$0.6073 * \begin{pmatrix} a - sc & b - sd \\ c + sa & d + sb \end{pmatrix} \tag{10}$$

In Verilog code, this can be done in one step with CORDIC implementation. Symmetrisation operation takes 19 cycles. One row/column operation takes 19 cycles. Hence, one sweep takes 171(19*3*3) cycles and three sweeps take 513 cycles. The hardware takes 555 clock cycles to compute SVD and write the results back to BRAM.

V. TESTING STRATEGY

For checking the correctness of the SVD, the simulation results are compared with the values produced by a reference Python program.

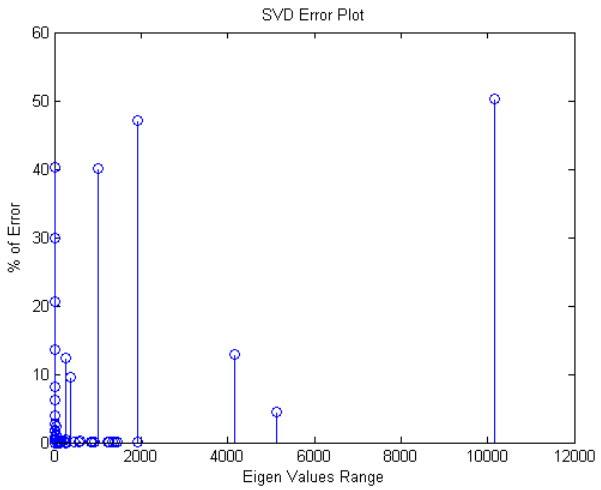
A. Simulation Strategy

Test Cases:

1. Test cases based on Range of inputs.

- All elements are of very small size.
- All elements which are in range 1 – 1000.
- All elements exceeding the value 1000.
- All elements are equal to zeros.

In all the above cases, the simulation results are very close the expected values and the percentage of error is typically less than 1. But in case of all the elements are of very small size the percentage of error is typically from 1 to 20 due to its very small input size. And also the simulation results are not matching with the expected values for the input matrices where all elements are exceeding the value 1000. This is because of arithmetic operations on these elements exceeding the 16-bit number.



The above graph presents the percentage of error values for different kinds of Eigen values.

2. Test cases on Different combinations of diagonal elements.

- Some test cases on all diagonal elements as zeros, negative and positive.
- All non-diagonal elements as zeros and different combination of diagonal elements.

In all the different combinations of diagonal and non-diagonal elements the simulation results are matching with the expected values produced by the reference program.

3. Test cases on Range of Eigen Values.

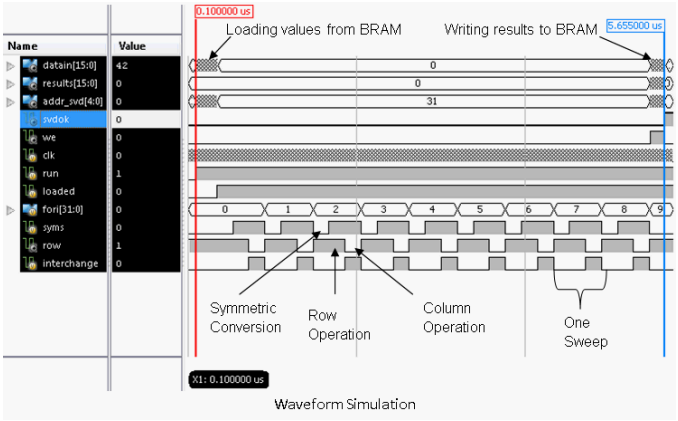
- Eigen values exceeding the value 2000.

For some of the Eigen values exceeding value 2000, the percentage of error is becoming high because of the results of the arithmetic operands exceed 16 bit value.

4. Test cases on Dependent rows.

- All the elements of the matrix are equal.
- Two of the matrix rows or columns are equal.

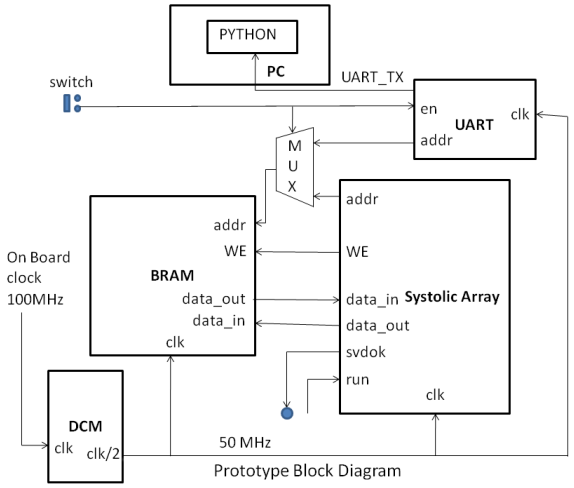
In these test cases the results generated by the simulation are not matching with the excepted results. We are getting the correct Eigen values only for the independent rows or columns.



B. *Test Hardware Description*

The matrix elements are initially stored in BRAM memory. Apart from BRAM and Systolic array, the test hardware contains DCM (Digital Clock Manager) and UART module. BRAM memory is 32x16 bits single port Block RAM with write enable.

Initially, run and reset input is made zero. UART is connected to PC and the 'serial port read' python program is run in PC. To see the initial contents of BRAM (input matrix), the press button 'SWITCH' is pressed once. BRAM contents will be transferred to PC. The python code captures the data and displays it as matrix. Then the 'run' input is asserted high. After SVD is computed, the LED (svdok) will glow. Again the 'SWITCH' is pressed, so that the computed values are transferred to PC and the python program displays it as matrix.



VI. HARDWARE

A. *Hardware inferred*

Total number of LUTs used is 11304 (41%) and number of registers used is 1445 (2%).

TABLE I. HARDWARE BLOCKS IN DESIGN

Block Type	No of Blocks
32-bit Adders/Subtractors	26
32-bit Adder Tree	32
Counters	13
32-bit Accumulator	8
Flip-flops	834
Comparators	9
32 bit shifter arithmetic right	20
FSM	2

The hardware contains number of adders and comparators of 8, 16 bits also which is not included in above table.

B. *Timing Information for SVD*

In SVD calculation the maximum delay (critical path) is due to data path from register arrays processing element PE21 (FF) to same element 14th output pin which is connected to a register again. Finally Delay which accounts for clock period is 11.244 ns. The maximum clock frequency by which the design can run is 88.9375MHz.

VII. CONCLUSION

The overall systolic array architecture for SVD does not have any hardware multipliers which is a large reduction is amount of resources used. This architecture is based on multiplexers and adders. There is a lot of routing overhead associated because the state machine of Systolic array needs access to each register to data transfer between BRAM and registers. This routing overhead can be reduced by adopting a shift register based approach for loading values into registers and also for storing values from the registers to BRAM. This approach has not been implemented, but this can be done as our future work.

REFERENCES

[1] Brent R.P., Luk F.T., Van Loan C. "Computation of the Singular Value Decomposition Using Mesh-Connected Processors". Journal of VLSI and Computer Systems, Volume I, Number 3. pp 242-270. 1983

[2] GOTZE J., PAUL S., SAUER M.: 'An efficient Jacobi-like algorithm for parallel eigenvalue computation', IEEE Transactions on Computers, 1993, Vol 42,No.9, pp.1058–1065.

[3] A. A. Maciejewski and C. A. Klein, "The singular value decomposition: Computation and applications to robotics," International Journal of Robotics Research, Vol. 8, No. 6, pp. 63-79, December 1989.

[4] I. E. Telatar, "Capacity of multi-antenna Gaussian channels," European Trans. Telecommun., vol. 10, pp. 585-595, Nov./Dec. 1999.