FPGA BASED EIGENFILTERING FOR REAL-TIME PORTFOLIO RISK ANALYSIS

*Mustafa U. Torun, Onur Yilmaz, and Ali N. Akansu**

Department of Electrical and Computer Engineering
New Jersey Institute of Technology
University Heights, Newark, NJ 07102 USA

ABSTRACT

The empirical correlation matrix of asset returns in an investment portfolio has its built-in noise due to market microstructure. This noise is usually eigenfiltered for robust risk analysis and management. Jacobi algorithm (JA) has been a popular eigensolver method due to its stability and efficient implementations. We present a fast FPGA implementation of parallel JA for noise filtering of empirical correlation matrix. Proposed FPGA implementation is compared with CPU and GPU implementations. It is shown that FPGA implementation of eigenfiltering operator in real-time significantly outperforms the others. We expect to see such emerging high performance DSP technologies to be widely used by the financial sector for real-time risk management and other tasks in the coming years.

1. INTRODUCTION

The commonly used metric for the measurement of portfolio risk is the standard deviation of its return [1, 2]. The portfolio risk is defined as a function of pair-wise correlations between its asset returns. The empirical correlation matrix of assets in a portfolio has intrinsic noise due to market microstructure that degrades risk calculations. Eigendecomposition of empirical correlation matrix, so called eigenfiltering, is a popular and robust method to remove undesired market noise [2]. However, eigenanalysis is a computationally expensive operator. With the advent of affordable high performance computing devices such as field-programmable gate array (FPGA) and general purpose graphics processing unit (GPU), research interest on parallel algorithms has recently gained new momentum. Jacobi algorithm (JA) [3] is a highly parallel method to implement eigenanalysis of a given matrix. In this paper, we present an FPGA implementation of JA for eigenfiltering of noise for portfolio risk analysis. Performance improvement FPGA brings over GPU and CPU implementations [4, 5] is emphasized, and risk calculations on different devices are compared.

*Corresponding author: akansu@njit.edu

2. PORTFOLIO RISK AND EIGENFILTERING OF EMPIRICAL CORRELATION MATRIX FOR NOISE REMOVAL

Let $\mathbf{q} = \begin{bmatrix} q_1 & q_2 & \cdots & q_N \end{bmatrix}^{\mathrm{T}}$ represent investment allocation vector for a portfolio of N financial assets, and let $\mathbf{r} = \begin{bmatrix} r_1 & r_2 & \cdots & r_N \end{bmatrix}^{\mathrm{T}}$ be an $N \times 1$ vector comprised of asset returns where superscript T is the transpose operator. Return of an N-asset portfolio is expressed as $r_p = \mathbf{q}^{\mathrm{T}}\mathbf{r}$. Portfolio risk is defined as the standard deviation of portfolio return and calculated as follows

$$\sigma_p = \left(E\left\{r_p^2\right\} - \mu_p^2\right)^{1/2} = \left(\mathbf{q}^{\mathrm{T}}\mathbf{C}\mathbf{q}\right)^{1/2} = \left(\mathbf{q}^{\mathrm{T}}\mathbf{\Sigma}^{\mathrm{T}}\mathbf{P}\mathbf{\Sigma}\mathbf{q}\right)^{1/2}, \tag{1}$$

where $\mu_p = E\{r_p\} = \mathbf{q}^{\mathrm{T}}E\{\mathbf{r}\} = \mathbf{q}^{\mathrm{T}}\boldsymbol{\mu}$ is the expected return of the portfolio, $\boldsymbol{\mu}$ is an $N \times 1$ vector, and its elements are expected returns of assets, $\mathbf{\Sigma}$ is an $N \times N$ diagonal matrix with elements corresponding standard deviations (volatilities) of asset returns σ_i, $\mathbf{C}$ is $N \times N$ covariance matrix of asset returns, and $\mathbf{P}$ is $N \times N$ correlation matrix where $[P_{ij}] = \rho_{ij}$. We can express $\mathbf{P}$ using eigendecomposition

$$\mathbf{P} = \mathbf{\Phi}\mathbf{\Lambda}\mathbf{\Phi}^{\mathrm{T}}, \tag{2}$$

where $\mathbf{\Lambda} = diag(\lambda_1, \lambda_2, \ldots, \lambda_N)$ is a diagonal matrix with the eigenvalues as its elements, λ_k as the k*th* eigenvalue with the order $\lambda_k \geq \lambda_{k+1}$. $\mathbf{\Phi} = \begin{bmatrix} \boldsymbol{\phi}_1 & \boldsymbol{\phi}_2 & \cdots & \boldsymbol{\phi}_N \end{bmatrix}$ is an $N \times N$ matrix comprised of N eigenvectors as its columns, and $\boldsymbol{\phi}_k$ is the $N \times 1$ eigenvector corresponding to the k*th* eigenvalue, λ_k. Note that $\lambda_k \geq 0 \ \forall_k$ and $\sum_k \lambda_k = N$. Empirical correlation matrix is broken in two components through the eigenfiltering operation [2]

$$\tilde{\mathbf{P}} = \sum_{k=1}^{L} \lambda_k \boldsymbol{\phi}_k \boldsymbol{\phi}_k^{\mathrm{T}} + \mathbf{E}, \tag{3}$$

where L is the number of selected factors (eigenvalues) with $L \ll N$, and $\mathbf{E}$ is the diagonal noise matrix added in the equation in order to preserve the total variance, i.e. keeping the trace of $\tilde{\mathbf{P}}$ to be equal to N. Therefore, the elements of $\mathbf{E}$ are defined as

$$\mathbf{E} = [E_{ij}] = \left(1 - \sum_{k=1}^{L} \lambda_k \phi_i^{(k)} \phi_j^{(k)}\right) \delta_{i-j}, \tag{4}$$

where δ_i is the Kronecker delta function, $\phi_i^{(k)}$ is the i*th* element of the k*th* eigenvector. Eigenfiltered portfolio risk is calculated by replacing $\mathbf{P}$ with $\tilde{\mathbf{P}}$ (3) as

$$\tilde{\sigma}_p = \left(\mathbf{q}^{\mathrm{T}}\mathbf{\Sigma}^{\mathrm{T}}\tilde{\mathbf{P}}\mathbf{\Sigma}\mathbf{q}\right)^{1/2}. \quad (5)$$

In order to calculate eigenfiltered risk of (5) in hardware, a robust and fast eigensolver algorithm is needed. Jacobi algorithm is used due to its robustness [6] and highly parallel implementation on computing devices [3].

3. JACOBI ALGORITHM FOR EIGENDECOMPOSITION

Jacobi algorithm provides an approximate numerical solution to (2) by iteratively reducing the squared sum of the off-diagonal elements, ϵ, of matrix $\mathbf{P}$ by multiplying it from the right and left with Jacobi rotation matrix, $\mathbf{J}(p,q,\theta)$, and its transpose, respectively, and overwriting onto itself as expressed

$$\mathbf{P}^{(k+1)} = \mathbf{J}^{\mathrm{T}}(p,q,\theta)\mathbf{P}^{(k)}\mathbf{J}(p,q,\theta), \quad (6)$$

where $1 \leq p < q \leq N$. Note that multiplication from left and right corresponds to row and column updates in $\mathbf{P}$, respectively. Matrix $\mathbf{J}(p,q,\theta)$ is sparse as defined

$$\mathbf{J}(p,q,\theta) = [J(p,q,\theta)_{ij}] = \begin{cases} \cos\theta & i=p,\ j=p \\ \sin\theta & i=p,\ j=q \\ -\sin\theta & i=q,\ j=p \\ \cos\theta & i=q,\ j=q \\ 0 & \text{otherwise} \end{cases}. \quad (7)$$

Multiplications in (6) are repeated until $\epsilon < \epsilon_{THR}$ where ϵ_{THR} is a predefined threshold value. After sufficient number of rotations, matrix $\mathbf{P}$ gets closer to Λ, and the successive multiplications of $\mathbf{J}(p,q,\theta)$ leads to an approximation of eigenmatrix $\mathbf{\Phi}$ [3]. Elements of $\mathbf{J}(p,q,\theta)$, i.e. $c = \cos\theta$ and $s = \sin\theta$, are chosen such a way that the following equality holds

$$\begin{bmatrix} \bar{P}_{pp} & 0 \\ 0 & \bar{P}_{qq} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} P_{pp} & P_{pq} \\ P_{qp} & P_{qq} \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \quad (8)$$

where P_{ij} and $\bar{P}_{ij}$ are elements of $\mathbf{P}^{(k)}$ and $\mathbf{P}^{(k+1)}$ located on the i*th* row and j*th* column, respectively. $\mathbf{P}$ is a symmetric matrix, $P_{pq} = P_{qp}$. Using trivial trigonometric identities it follows from (8) that the rotation angle is equal to

$$\theta = 0.5\tan^{-1}\left[2P_{pq}/\left(P_{qq} - P_{pp}\right)\right]. \quad (9)$$

Jacobi algorithm is most efficiently implemented on a computing device with $N/2$ parallel processing units due to the sparsity of the rotation matrix $\mathbf{J}(p,q,\theta)$ by using chess tournament (CT) algorithm [3]. In CT, for N players, there are $N/2$ pairs and $N-1$ matches that have to be held such that each player matches against any other player in the group. Once a match set is completed, the first player stands still and every other player moves one seat in clockwise direction. For $N = 4$, the pairs for $N-1 = 3$ steps are defined as

$$\begin{bmatrix} p^{(1)} & p^{(2)} \\ q^{(1)} & q^{(2)} \end{bmatrix} : \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 \\ 4 & 3 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad (10)$$

Note that $p^{(2)}$ and $q^{(2)}$ are interchanged in the last step since the condition $p < q$ must hold [3]. It is noted that this interchanging is necessary after the step $N/2+1$.

4. FPGA BASED IMPLEMENTATION OF JACOBI ALGORITHM

Several VLSI [7–9] and FPGA [10–13] implementations of parallel Jacobi algorithm (PJA) are reported in the literature. Most hardware implementations of Jacobi are put on processors that employ coordinate rotation digital computer (CORDIC) [14, 15], i.e. an iterative algorithm to rotate vectors. Either a large number of processors are interconnected with systolic array (SA) [7, 13, 16] or only two processors are used in a cyclic fashion [12, 13]. For applications where only the eigenvalues are required, only half or upper-triangle SA are used [13]. We implement PJA for eigenfiltering of $\mathbf{P}$ in (2) by using full SA of CORDIC processors. We improved the prior designs reported such that the paths in SA can be reconfigured in order to support variable size input matrices as detailed below.

4.1. CORDIC Algorithm

CORDIC is a simple and efficient algorithm to rotate vectors when no hardware multiplier is available [14, 15]. Rotation of a 2×1 vector is achieved by

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}. \quad (11)$$

We can rewrite the elements of the rotated vector by using the trigonometric identity $\cos\theta = \left(1+\tan^2\theta\right)^{-1/2}$ as follows

$$x' = \frac{x - y\tan\theta}{\sqrt{1+\tan^2\theta}},\ y' = \frac{x\tan\theta + y}{\sqrt{1+\tan^2\theta}}. \quad (12)$$

In each iteration of CORDIC, i, rotation angle, θ, is restricted to $\theta = \tan^{-1}\left(\pm 2^{-i}\right)$. Therefore, computations of $y\tan\theta$ and $x\tan\theta$ can be implemented by simple binary shift operations. Specifically, in order to get closer to the target vector, CORDIC performs the following operations in each iteration

$$\begin{aligned} x_{i+1} &= K_i(x_i - y_i\mathbf{1}_{\varphi_i \geq 0}2^{-i}) \\ y_{i+1} &= K_i(x_i\mathbf{1}_{\varphi_i \geq 0}2^{-i} + y_i), \end{aligned} \quad (13)$$

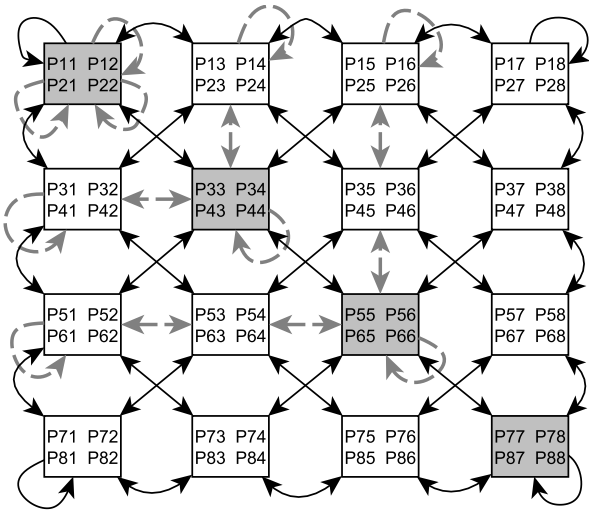

Fig. 1. Systolic array used for interconnecting CORDIC processors.

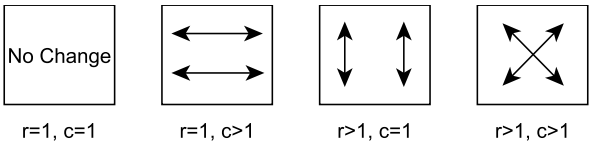

Fig. 2. Inner-exchange scheme of the processors.

where $K_i = \left(1 + 2^{-2i}\right)^{-1/2}$, $\mathbf{1}_X : X \rightarrow \{+1, -1\}$ is the indicator function, and φ_i is the residual angle updated as $\varphi_{i+1} = \varphi_i - \mathbf{1}_{\varphi_i \geq 0} \tan^{-1}(2^{-i})$ with $\varphi_0 = \theta$, $x_0 = 1$, and $y_0 = 0$. Note that calculation of φ_i requires *elementary angles*, $\tan^{-1}(2^{-i})$, that can be stored in a look up table (LUT). Moreover, calculation of K_i requires a square-root operator. In practice, an approximation to the asymptotic value of K_i, $\sim$0.60725, is used. Alternatively, values of K_i can be precalculated and stored in a LUT. Operations of CORDIC discussed so far are called *rotation mode.* It is also possible to measure the angle of a vector by using the *vectoring mode* of CORDIC. In this mode, initial values of the coordinates (13) x_0 and y_0 are set to be equal to the original coordinates. The algorithm is run such that the target rotated vector is $\begin{bmatrix} 1 & 0 \end{bmatrix}^{\mathrm{T}}$. In this mode, the indicator function given in (13) is modified as $\mathbf{1}_{y_i \geq 0}$.

4.2. CORDIC Processors and Systolic Array

There are two types of CORDIC processors (CP) in the design, namely diagonal and off-diagonal CPs. Both use rotation mode of CORDIC during the row and column updates. Additionally, diagonal CPs calculate the angle θ given in (9) and distribute it to the off-diagonal CPs on the same row and column. Note that θ in (9) is nothing else but half the angle of 2×1 vector $\begin{bmatrix} (P_{qq} - P_{pp}) & 2P_{pq} \end{bmatrix}^{\mathrm{T}}$. Hence, it is determined with using CORDIC vectoring mode. Once the angle is determined, all processors perform the row, column, and eigenvector update operations. Row update is a rotation of 2×1 vector $\begin{bmatrix} P_{pn} & P_{qn} \end{bmatrix}^{\mathrm{T}}$ where $1 \leq n \leq N$ that is performed in parallel by an array of $N/2 \times N/2$ CPs. After the completion of row update, similarly, column-update and eigenvector update operations are performed.

Fixed-point representation and truncation circuits are employed in CPs. Word length is of B bits and the decimal part is stored after the *Gth* MSB. Decimal number is calculated as

$$D = \sum_{i=0}^{B-1} b_i 2^{i-B+G}, \tag{14}$$

where b_i is the i*th* LSB with $0 \leq i < B$.

An SA [17] is used in our design to interconnect CPs. Instead of broadcasting data to each processor, SA ensures that the data travels around the network by an exchange among neighboring processors. This significantly reduces data transfer complexity. Fig. 1 displays SA of CPs for $N = 8$. Note that SA based eigensolvers reported in the literature (see [13] and references therein) only support matrices of a predetermined size. However, our improved design supports matrices with different sizes as long as they are within the maximum size range supported. This flexibility is achieved by i. turning on and off the communication between the processors, and ii. re-routing the data transfer automatically for any given matrix size.

4.3. Control and Synchronization

Our FPGA implementation is coordinated by two finite state machines (FSM). Namely, main FSM (MFSM) and solver FSM (SFSM). MFSM is responsible for overall operation of the design. It has four states named as IDLE, LOAD, SOLVE, and OUTPUT. In the LOAD state, each element of input array with size $N^2 \times 1$ representing a matrix of size $N \times N$ is received serially. When all elements are fed, MFSM switches to SOLVE state in where SFSM is started. SFSM is responsible for controlling CPs such that Jacobi rotations are performed. It has eight states named as INIT, ANGLE, ROWP, COLP, EVP, INX, OUTX, and FINISH. In ANGLE state, rotation angle defined in (9) is calculated by diagonal CPs. Next, SFSM switches to ROWP, COLP, and EVP states, that stand for row-processing, column-processing, and eigenvector-processing, respectively. Once all the rotations are completed, SFSM proceeds to INX and OUTX states. Data residing in each CP is exchanged, first within the processor itself, then among neighboring processors, in INX and OUTX states, respectively, such that all processors are ready for the next step in accordance with the chess tournament (CT) algorithm. Note that CPs located on the top row and far left column of SA perform inner exchange differently than the rest due to the fact that the first index is kept static in CT (see Fig. 2). Outer-exchange scheme is displayed in Fig. 1. Finally, whenever

	$N = 16$	$N = 64$	$N = 256$	$N = 512$	$N = 1024$
CPU	0.97	17.25	1,326.88	11,570.31	110,761.36
GPU	2.55	10.79	65.43	248.29	1,610.13
FPGA	0.17	*2.79*	*44.58*	*178.32*	*713.29*

Table 1. Computation time in milliseconds for CPU, GPU, and FPGA. Values in italics are estimated via (16).

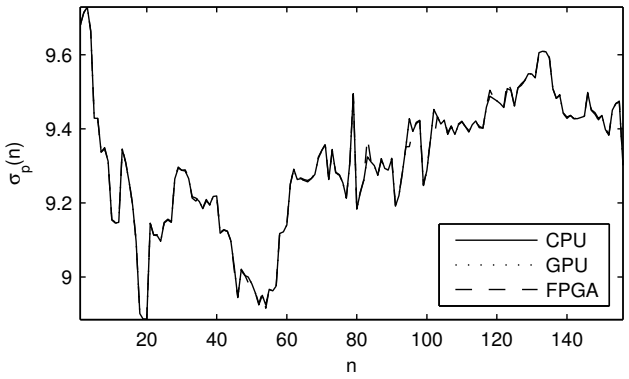

Fig. 3. Portfolio risk in bps (0.01%) as a function of discrete-time, n, calculated with CPU, GPU, and FPGA.

total number of sweeps is reached, SFSM proceeds to FINISH state that results in MFSM to switch to the OUTPUT state.

4.4. Computation Time

At each step of a sweep, a CP uses 14 cycles in operations necessary for initialization and exchange of the data. Moreover, CORDIC algorithm is used for 4 times within a processor at each step. In the worst-case scenario, each run of CORDIC algorithm takes maximum number of iterations (cycles), i.e. M, to be completed. Given that there are $N-1$ steps in a sweep, number of cycles required for a complete sweep is $C_s = (N-1)(14+4M)$. As discussed earlier, N^2 cycles are required for both feeding in and taking out the data from SA. Therefore, total number of cycles required for the design to provide the result is

$$C_t = 2N^2 + SC_s = 2N^2 + S(N-1)(14+4M) \quad (15)$$

where S is the pre-determined number of sweeps and larger than zero.

5. PERFORMANCE COMPARISONS

A real-time risk analysis system that supports empirical correlation matrix sizes up to $N_{max} = 16$ is implemented on Altera Stratix IV FPGA coded in VHDL [18] with the values of $M = 24$ in (15), $B = 32$ and $G = 16$ in (14). Area and f_{max}, the maximum frequency the circuit can be clocked, are 334,252 ALUTs (78% of total available) and 56.85 MHz, respectively. For comparison purposes, we cite performance of CPU and GPU implementations of Jacobi algorithm reported in [5]. We re-tested with faster CPU and GPU we currently have, Intel® Core™ i7-3960X CPU and an NVIDIA GeForce™ GTX 580, respectively. For GPU, time required for memory I/O is not measured. Computation times required to perform $S = 6$ sweeps of JA are tabulated in Table 1 for various matrix sizes, N. FPGA results for $N > N_{max}$ are estimated by using the rationale that it would take $(N/N_{max})^2$ times more to solve a larger matrix by using the block JA [3]. More specifically, the worst-case computation time for FPGA is calculated by using the equation (15)

$$\widehat{t}_{FPGA} = \left(\frac{N}{N_{max}}\right)^2 \frac{1}{f_{max}} S(N_{max}-1)(14+4M), \quad (16)$$

where time required for I/O, i.e. $2N^2$, is omitted for a fair comparison with GPU. Risk analyses for a portfolio comprised of 16 largest market capitalization stocks listed in Dow Jones Industrial Average (DJIA) index as of Nov 14, 2012 are implemented on CPU, GPU, and FPGA. The portfolio rebalancing period is 5 minutes. The historical market data utilized for experiments runs from Nov 14, 2012 9:30 EST to Nov 15, 2012 16:00 EST. Capital is allocated among assets of the portfolio in equal amounts, and empirical correlation matrix is estimated over a day long time window (78 samples per day). Calculated risks according to (5) using empirical correlation matrices eigenfiltered with CPU, GPU and FPGA implementations of JA are displayed in Fig. 3. RMS value of discrepancy between risks measured by CPU and GPU implementations is 55×10^{-6} bps. It is 11.5×10^{-3} bps between CPU and FPGA implementations.

Remark*:* Smarter I/O handling may be employed in both FPGA and GPU implementations. For example, GPU might use RDMA. Similarly, I/O may be handled by fast memory with a faster clock adjacent to FPGA chip, and data may be fed into SA in parallel form. This topic is beyond the scope of this paper and deserves further study.

6. CONCLUSIONS

We forwarded an FPGA implementation of parallel Jacobi algorithm for real-time eigenanalysis of a matrix. It is used for eigenfiltering of noisy empirical correlation matrix of an investment portfolio. We compared performance of the proposed FPGA, CPU and GPU implementations under the same test conditions. It is shown that FPGA implementation of eigenfiltering with JA significantly outperforms the others. It offers fast and scalable risk analysis. We predict that such an affordable technology will be embedded in risk management systems of the future.

7. REFERENCES

[1] H. M. Markowitz, *Portfolio Selection: Efficient Diversification of Investments*. Wiley, New York, 1959.

[2] M. U. Torun, A. N. Akansu, and M. Avellaneda, "Portfolio risk in multiple frequencies," *IEEE Signal Processing Magazine, Special Issue on Signal Processing for Financial Applications*, vol. 28, pp. 61 – 71, Sep. 2011.

[3] G. H. Golub and C. F. V. Loan, *Matrix Computations*. Johns Hopkins University Press, 1996.

[4] M. U. Torun, O. Yilmaz, and A. N. Akansu, "Novel GPU implementation of Jacobi algorithm for Karhunen-Loève transform of dense matrices," in *Proc. IEEE 46th Annual Conference on Information Sciences and Systems (CISS)*, pp. 1 – 6, Mar 2012.

[5] M. U. Torun and A. N. Akansu, "A novel GPU implementation of eigenanalysis for risk management," in *Proc. IEEE 13th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pp. 490 – 494, Jun 2012.

[6] J. Demmel and K. Veselic, "Jacobi's method is more accurate than QR," *SIAM J. Matrix Anal. Appl.*, vol. 13, pp. 1204 – 1245, 1992.

[7] R. Schreiber, "Systolic arrays for eigenvalue computation," *Proceedings of the SPIE*, vol. 341, pp. 27 – 34, 1982.

[8] R. P. Brent, F. T. Luk, and C. V. Loan, "Computation of the singular value decomposition using mesh-connected processors," *Journal of VLSI and Computer Systems*, vol. 1, pp. 242–270, 1985.

[9] J. Gotze, S. Paul, and M. Sauer, "An efficient Jacobi-like algorithm for parallel eigenvalue computation," *IEEE Transactions on Computers*, vol. 42, pp. 1058 – 1065, Sep. 1993.

[10] M. Kim, K. Ichige, and H. Arai, "Design of Jacobi EVD processor based on CORDIC for DOA estimation with MUSIC algorithm," *Proc. PIMRC*, vol. 1, pp. 120 – 124, 2002.

[11] A. Ahmedsaid, A. Amira, and A. Bouridane, "Improved SVD systolic array and implementation on FPGA," in *Proc. IEEE International Conference on Field-Programmable Technology*, pp. 35 – 42, Dec. 2003.

[12] I. Bravo, P. Jimenez, M. Mazo, J. Lazaro, and G. A., "Implementation in FPGAs of Jacobi method to solve the eigenvalue and eigenvector problem," in *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 1 – 4, Aug. 2006.

[13] Y. Liu, C.-S. Bouganis, and P. Y. K. Cheung, "Hardware efficient architectures for eigenvalue computation," *Computers & Digital Techniques, IET*, vol. 3, pp. 72–84, January 2009.

[14] J.-M. Delosme, "CORDIC algorithms: theory and extensions," *Proceedings of the SPIE*, vol. 1152, pp. 131 – 145, 1989.

[15] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proc. ACM/SIGDA sixth international symposium on field programmable gate arrays*, pp. 191 – 200, 1998.

[16] H. T. Kung, "Why systolic architectures?," *Computer*, vol. 15, pp. 37–46, January 1982.

[17] H. T. Kung and C. E. Leiserson, *Systolic Arrays for (VLSI)*. CMU-CS, Carnegie-Mellon University, Department of Computer Science, 1978.

[18] V. Pedroni, *Circuit Design and Simulation With VHDL*. MIT Press, 2010.