Hardware architectures for eigenvalue computation of real symmetric matrices

Y. Liu  C.-S. Bouganis  P.Y.K. Cheung

Department of Electrical and Electronic Engineering, Imperial College London, UK
E-mail: christos-savvas.bouganis@imperial.ac.uk

Abstract: Computation of eigenvalues is essential in many applications in the fields of science and engineering. When the application of interest requires the computation of eigenvalues of high throughput or real-time performance, a hardware implementation of an eigenvalue computation block is often employed. The problem of eigenvalue computation of real symmetric matrices is focused upon. For the general case of a symmetric matrix eigenvalue problem, the approximate Jacobi method is proposed, where for the special case of a $3 \times 3$ symmetric matrix, an algebraic-based method is introduced. The proposed methods are compared with various other approaches reported in the literature. Results obtained by mapping the above architectures on a field programmable gate array device illustrate the advantages of the proposed methods over the existing ones.

1 Introduction

The computation of eigenvalues for symmetric matrices is essential in many applications. In video processing, for example, the computation of eigenvalues for symmetric matrices is required in feature detection [1], optical flow [2] and motion analysis [3] algorithms, just to name a few. The inevitable trend of escalating the number of pixels within images is clear in commercial cameras. As the number of pixels increases, it is a necessity to have a dedicated hardware module for the computation of eigenvalues that can achieve high throughput for real-time operation of the aforementioned algorithms.

Current hardware approaches rely mostly on Jacobi-like methods for the computation of eigenvalues because of their inherent parallelism, which makes them particularly suitable for distributed resource systems. A systolic array of processors [4–6] was originally proposed for a VLSI implementation of Jacobi-like methods and the advent of field programmable gate array (FPGA) popularised this architecture [7–9]. Although previous implementations employed the exact Jacobi method, Gotze et al. [10] proposed an approximation to the exact jacobi method and claimed that it is theoretically superior to its exact counterpart without experimental evidence. Besides the Jacobi-based method, the algebraic method [11], which is often used in software approaches for small matrices (e.g. $3 \times 3$ matrix), offers an alternative approach that may be more suitable for some restricted classes of eigenvalue problems.

This work focuses on the implementation issues regarding the approximate Jacobi method and proposes a new architecture for mapping the algebraic method into hardware. Moreover, it provides a thorough and systematic comparison of the above mentioned approaches in terms of resource utilisation and throughput in order to enable a system designer to choose between them with confidence. As a target device, a modern FPGA has been selected.

In [12], we have presented some initial results regarding the mapping of the above algorithms into hardware. This paper extends that work by comparing the proposed architectures with the window Jacobi method, a variant of the exact Jacobi method, by performing a comparison study between systolic and dual processor architectures, and finally by investigating the impact of resource sharing on the overall throughput of the system and area usage.
In summary, the novel contributions of this paper are:

- A detailed hardware implementation of the approximate Jacobi method.
- A novel architecture for the implementation of the algebraic method that handles the special case of computing the eigenvalues of a $3 \times 3$ symmetric matrix.
- A detailed comparison of the proposed architectures with various other architectures from the literature in terms of area and throughput in the cases where these architectures are mapped in a modern FPGA device.

2 Jacobi-based methods

It is well known that if $A$ is a symmetric matrix, then there exists a real-valued orthogonal matrix $Q$ such that post- and pre-multiplication with matrix $A$ results in a diagonal matrix $D$ (1). The elements of the main diagonal of $D$ are the eigenvalues of matrix $A$

$$Q^T A Q = D = \text{diag}(\lambda_1, \ldots, \lambda_n)$$

Current methodology for eigenvalue computation relies on the above property and computes the eigenvalues of the matrix under investigation by converting it into a diagonal matrix. QR and Jacobi methods are two of the most widely used methods that are based on the above property.

QR methods rely on the QR decomposition where a matrix is decomposed to a product of an orthogonal matrix $Q$ with an upper triangular matrix $R$.

Most of the QR methods first use a sequence of Householder transformations to reduce an original matrix $A$ to a tridiagonal form, and then apply a version of the QR decomposition to obtain all the eigenvalues of the resulting tridiagonal matrix.

Jacobi methods rely on Jacobi transformations. This work is focused on the Jacobi methods, as they are inherently more parallel than QR methods. As it will be demonstrated later in the paper, operations in Jacobi methods are hardware friendly avoiding the use of square root and division operations, which are common in QR decomposition methods.

2.1 Exact Jacobi method

Given a symmetric $n \times n$ matrix $A$, Jacobi methods [13] work by systematically reducing the norm of the off-diagonal elements as defined in (2)

$$\mathcal{F}(A) = \sqrt{\sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} a_{ij}^2}$$

This is accomplished by performing a sequence of orthogonal similarity transformations $J_{pq}$ to matrix $A$

$$A^{(k+1)} = J_{pq} A^{(k)} J_{pq}, \quad k = 0, 1, 2, \ldots$$

with $A^{(0)} = A$

$J_{pq}$ is the transformation matrix governing the Jacobi rotation and is defined by the parameters $(c, s, -s, c)$ in the $(pq, pq, qp, qq)$ entries of an $n \times n$ identity matrix, where $p < q$, $c = \cos(\theta)$, and $s = \sin(\theta)$. $\theta$ is the rotation angle.

By applying (3) iteratively to matrix $A$ to make the off-diagonal elements zero, the diagonal elements converge to the eigenvalues of $A$, denoted as $\lambda_i$.

Each update in (3) affects only the $p$ and $q$ rows and columns of $A^{(k)}$ and thus, the norm of the updated matrix is given by (4). The aim of the method is to reduce the norm of the updated matrix $A$ to zero in the fewest possible transformations

$$\left[\mathcal{F}(A^{(k+1)})\right]^2 = \left[\mathcal{F}(A^{(k)})\right]^2 - 2[c_{pq}^2 - c_{pp}^2] (4)$$

In the exact Jacobi method, exact Jacobi rotations are performed, that is, the coefficients $c$ and $s$ at each iteration are computed using (5) and (6) such that $d_{pq}^{(k+1)}$ becomes zero and, hence, maximal reduction of $\mathcal{F}(A^{(k+1)})$ is achieved

$$\theta = \frac{1}{2} \tan^{-1}\left[\frac{2d_{pq}^{(k)}}{d_{pp}^{(k)} - d_{qp}^{(k)}}\right]$$

$$c = \cos(\theta), \quad s = \sin(\theta)$$

2.2 Hardware considerations of exact Jacobi method

The exact Jacobi method comprises the calculation of angle $\theta$ from (5) followed by the orthogonal similarity transformations in (3). The latter process is equivalent to the multiplications of the elements of matrix $A$ with $c$ and $s$ in (6). These trigonometric operations can be performed with only addition, shifting and multiplication via the Coordinate Rotation Digital Computer (CORDIC) algorithm [14]. Performing rotations using the CORDIC algorithm in FPGAs can be found in [15], where the use of CORDIC in the context of the eigenvalue computation is described in [16]. In fact, all the methods of eigenvalue computation depicted in this paper involve the CORDIC algorithm to a certain extent, and so the following paragraphs are devoted to a brief explanation of the algorithm.

The CORDIC algorithm rotates a vector $[x, y]$ by an arbitrary angle $\theta$ in a hardware friendly manner. The idea revolves around the use of the elementary angles $\theta_i$, defined as

$$\theta_i = \pm \arctan(2^{-i}) \quad \text{with} \quad i = 0, 1, \ldots, b - 1$$
where $b$ is the wordlength of the hardware system. Note that a rotation by any of these elementary angles requires only shifts, additions and multiplications by a scaling factor. Since any arbitrary angle can be composed of a combination of the elementary angles, the rotation by any arbitrary angle can be obtained by performing successive iterations by elementary angles. More importantly, the individual scaling can be removed from each iteration and the accumulative scaling factor is applied only once after the last iteration. As a result, only one multiplication by a constant is involved regardless of the number of the iterations. Each CORDIC iteration corresponds to a rotation by an elementary angle. As the iterations progress, the elementary angle $\pm \arctan 2^{-i}$ decreases and the net rotation converges to the desired rotation of the target arbitrary angle. The accumulative scaling factor, which needs to be multiplied with the resulting vector after $b$ iterations, is given by (8). This scaling can be executed with $\sim b/4$ shifts and additions [8]

$$
1/K_b = \prod_{i=b}^{1} \frac{1}{\sqrt{1 + 2^{-2i}}} \quad (8)
$$

By convention, a ‘step’ in Jacobi methods consists of finding the angle $\theta$ which satisfies (5), followed by a rotation by $\theta$. A more thorough implementation of the exact Jacobi method can be found in [7, 8]. The angle $\theta$ can be calculated without the division operation by setting the initial CORDIC condition to appropriate values.

### 2.3 Window Jacobi method

Although each step of the exact Jacobi method causes one off-diagonal element to be annihilated, that is, made zero, this result can be reversed in the next step. Hence, the effort of the previous step is wasted to some extent and precise annihilation of elements is an overkill but for the very last steps [17, 18]. Thus, the accuracy of annihilation may be coarse in the early steps and refined as the algorithm proceeds. This can be regarded as having a window of iteration indexes chosen for the CORDIC algorithm. This method is referred to as the window Jacobi method in this work.

### 2.4 Hardware considerations of window Jacobi

The methodology of the window Jacobi method is similar to that of the exact Jacobi method. Both methods compute $\theta$ from (5) and rotate the matrix using $c$ and $s$ from (6) in each step. The difference of the window Jacobi from the exact Jacobi method is that the CORDIC algorithms applied for these operations have variable resolution. Instead of having CORDIC rotations with the index in $\theta_i = \pm \arctan 2^{-i}$ running from 0 to $b - 1$, the index can run from $m$ to $n$, where $m \geq 0$ and $n \leq b$, in the window Jacobi method. Experiments in [19, 20] indicate that with a fixed window width $n - m + 1$ equal to 3 or 4, the total number of steps barely exceeds the number of steps in the exact Jacobi method, reducing, however, the total number of CORDIC operations. However, the window Jacobi method requires more complex control logic than the exact Jacobi method since the appropriate CORDIC indexes at each iteration have to be selected.

### 3 Approximate Jacobi method

Several works, both software and hardware related, have been done on the exact and window Jacobi methods. On the other hand, the literature lacks information on the hardware implementation of the approximate Jacobi method. This work focuses on the detailed implementation of this method into hardware.

The approximate Jacobi method takes a further step forward than the window Jacobi method targeting an efficient use of the hardware resources. As has been already highlighted, the required computational effort to find the exact $J_{pq}$ in (3) that provides a maximum reduction of $F(A)$ in (2) is not completely justifiable since this annihilation effort is undermined in the next step. It is sufficient to approximate the $J_{pq}$ as long as the orthogonality is preserved [10]. Although the window Jacobi method performs a ‘window’ of CORDIC iterations whose resultant angle is close to the $\theta$ in (5), and whereas the exact Jacobi method performs a full CORDIC rotation, the approximate Jacobi method finds the CORDIC elementary angle in (7) that is closest to the desired rotation angle $\theta$. Rotating by one elementary angle requires one single CORDIC iteration, as opposed to a rotation by the ‘exact’ $\theta$, which requires $b$ number of iterations.

#### 3.1 Methodology of the approximate Jacobi method

The approximate Jacobi method does not annihilate the off-diagonal element $a_{pq}^{(k+1)}$ in each iteration, but reduces it by a factor $d$ as in (9)

$$
a_{pq}^{(k+1)} = d a_{pq}^{(k)}, \text{ where } d = \frac{1 - 2\pi t - t^2}{1 + t^2} \quad (9)
$$

The maximal value of $|d|$, denoted as $|d|_{\max}$, is a measure of the quality of the approximation since a smaller $|d|_{\max}$ gives rise to a larger reduction of $J_{pq}^{(k+1)}$. In comparison, the exact Jacobi method, with $t$ computed as in (5), yields $d = 0$ at each update. In contrast, the approximate Jacobi method approximates $t$ with hardware efficient operators, notably adders and shifters. The approximated $\tilde{t}$ is shown in (10), where $b$ denotes the wordlength of the hardware system again

$$
\tilde{t} = \text{sign}(t) \cdot 2^{-l} \approx t = \tan \theta, \quad l \in [0, 1, 2, \ldots, b] \quad (10)
$$

The rotation that approximates the Jacobi rotation is described by $J_{pq}(\tilde{t})$, which is an $n \times n$ identity matrix
with its four entries at \((p, p, q, q)\) replaced as shown in (11)

\[
J_{pq}(I) = \frac{1}{\sqrt{1 + 2^{-2l}}} \begin{bmatrix}
1 & \text{sign}(\tau)2^{-l} \\
-\text{sign}(\tau)2^{-l} & 1
\end{bmatrix}
\] (11)

It is essential for the approximate Jacobi method to choose \(l\) such that the approximate angle \(\tilde{\theta} = \arctan(2^{-\tilde{l}})\) is the closest to the exact rotation angle \(\theta\) and \(|d|_{\text{max}}\) is minimised. Gotze et al. [10] derived the following algorithm which requires at most three comparisons to find \(l\) while guaranteeing \(|d|_{\text{max}} \leq 1/3\).

- Compute \(k\) according to (12), where \(a_{D} = a_{pp} - a_{pq}\) and \(\exp\) denotes the exponent of a number. In binary number representation, \(\exp\) means finding the position of the most significant ‘1’ bit in an expression

\[
k = \exp(|a_{D}|) - \exp(|a_{pq}|)
\] (12)

- Determine a set of possible values of \(l\) using Table 1.

- When \(k > -2\), \(l\) is chosen as the one that satisfies the following expression

\[
(2^l - 2^{-l+1})|a_{pq}| \leq |a_{D}| + 2^{-1}|a_{D}| < (2^{l+1} - 2^{-l})|a_{pq}|
\]

The core idea that contributes to the efficiency of the approximate Jacobi method is that the angle of rotations is designed to be equal to a CORDIC elementary angle. It should be noted that the scaling factor in the approximate Jacobi method varies with \(l\). The corresponding scaling factor is given in (13)

\[
\frac{1}{K_l} = \frac{1}{\sqrt{1 + 2^{-2l}}}
\] (13)

As \(l\) increases, the scaling factor converges to unity. The implication is that \(1/K_l\) with large \(l\) need not to be stored. Experiments have shown that the number of \(1/K_l\) values that are needed to be stored is around \(b/2\).

### 3.2 Systolic architecture of the approximate Jacobi method

A systolic array architecture [5, 6], which offers a high degree of parallelism, is proposed for the hardware implementation of the approximate Jacobi method. The architecture is in line with previous work in the literature for the implementation of the exact Jacobi method [4, 7, 8], permitting a direct comparison between the two methodologies.

Each processor within the systolic array has bilateral communication with its nearest neighbours to avoid broadcasting of signals. One of the benefits of this short distance communication is that as the size of the systolic array increases, the complexity of routing between processors does not increase. To take further advantage of the symmetry of the matrix, an upper triangular array as suggested in [10], rather than a full square array of processors, was adopted. The adopted architecture is illustrated in Fig. 1. The system consists of a set of diagonal processors that are responsible for the reduction of the non-diagonal elements, and a set of off-diagonal processors that are responsible to perform the necessary updates to the remaining elements of the matrix, which result from the transformations that have been applied by the diagonal processors.

The input matrix is divided into \(2 \times 2\)-size blocks and the elements in each block are mapped to a processor. Because of the symmetry of the input matrix, each diagonal processor operates on three elements of the matrix and each off-diagonal processor operates on four. Since the \(J_{pq}(I)\) in (11) differs from the identity matrix at four locations \((p, p, q, q)\), the approximate Jacobi rotation affects only the \(p\) and \(q\) columns and rows of the matrix. This underlying characteristic ensures that different \(pq\) pairs have independent operations, which allows the sequential processing of orthogonal transformation in (3) to be mapped into our parallel architecture. In our implementation, without any loss of generality, \(p\) are odd integers and \(q\) are even integers. For example, the pair \(p = 1, q = 2\) involves the first column, second column, first row and second row. \(a_{pp}, a_{pq}\) and \(a_{pq}\) are fed into the diagonal processor where the approximate Jacobi rotation is executed and gives rise to the reduction of \(a_{pq}\) according to (9). This reduction eventually leads to a diagonal resultant matrix that has the eigenvalues of the original matrix in its main diagonal. The

![Figure 1](image-url)
orthogonal transformation affects all the elements in the same rows and columns. These elements are fed into the off-diagonal processors for corresponding updating of the matrix.

3.2.1 Diagonal processor: The block diagram of a diagonal processor is shown in Fig. 2. Each diagonal processor accomplishes two tasks: (a) computation of $l$ and (b) updating the elements of the matrix through the approximate Jacobi rotation.

On receiving an input matrix, all the diagonal processors compute the quantity $l$ in parallel as illustrated in Fig. 3a. $l$ and $\text{sign}(\tau)$ computed by each diagonal processor are transmitted to the neighbouring off-diagonal processors in both the horizontal and vertical directions shown in Fig. 3b. Following analogous derivation to the update equations of the exact Jacobi method [4], the approximate rotation blocks execute the update operations for $a_{pp}$, $a_{pq}$ and $a_{qq}$ shown below, where $k$ denotes the current iteration:

$$a_{pp}^{(k+1)} = a_{pp}^{(k)} - \text{sign}(\tau)2^{l}a_{pp}^{(k)} + 2^{l}a_{pp}^{(k)}$$

$$a_{pq}^{(k+1)} = 2^{-2l}a_{pq}^{(k)} + \text{sign}(\tau)2^{l}a_{pq}^{(k)} + a_{pq}^{(k)}$$

$$a_{qq}^{(k+1)} = \text{sign}(\tau)2^{-l}(a_{pp}^{(k)} - a_{pq}^{(k)}) - 2^{-2l}a_{pq}^{(k)} + a_{pq}^{(k)}$$

After the updating, every diagonal processor reduces its individual $a_{pq}$ element, contributing to the progressive creation of the diagonal matrix that contains the eigenvalues of the original matrix. Therefore four reductions happen in parallel in the case illustrated in Fig. 3 where an $8 \times 8$ symmetric matrix is assumed.

In the diagonal processor, the scaling factor $1/K_l$ has to be applied twice and, hence, the square root in (13) is removed. The appropriate scaling can be calculated recursively using only shifts and additions as shown in [10]. Given $b$ as the wordlength of the signals in the diagonal processor, the recursion terminates when $2^{l+1}l \leq b$, that is, after $\log_2 \left[ b/2l \right]$ recursions since the accuracy of the system is reached. After the scaling, the matrix elements are updated and exchanged for those in the adjacent processors and the whole process is repeated.

3.2.2 Off-diagonal processor: The block diagram of an off-diagonal processor is shown in Fig. 4. Each off-diagonal processor has to wait for the arrival of $l$ and $\text{sign}(\tau)$ signals from the adjacent processors in both the horizontal and vertical directions. In Fig. 3b, the off-diagonal processors closest to the diagonal processors receive the $l$ and $\text{sign}(\tau)$ signals. In the next clock cycle shown in Fig. 3c, they pass on the signals to their vertical and horizontal neighbouring off-diagonal processors and in the meantime they start the updating of the elements stored in them.

The input matrix elements are subject to approximate rotation determined by the horizontally transmitted $l_h$ and $\text{sign}(\tau)_h$ signals, which have been generated by the diagonal processor on the same row. The update operations consist of...
of right-shifts and additions/subtractions, which are governed by the equations shown below

\[ a_{11} = a_{11} - \text{sign}(\tau)2^{-b}a_{21} \]
\[ a_{12} = a_{12} - \text{sign}(\tau)2^{-b}a_{22} \]
\[ a_{21} = a_{21} + \text{sign}(\tau)2^{-b}a_{11} \]
\[ a_{22} = a_{22} + \text{sign}(\tau)2^{-b}a_{12} \]

The scaling is carried out by multiplying with one of the precomputed \( 1/K_l \) as in (13), which is selected by \( l_b \) via a multiplexer. Then, the scaled results are subject to similar operations controlled by vertically transmitted signals in order to complete the update of matrix elements.

Finally, the matrix is updated only after the updating operations are completed in the entire array of processors, including both diagonal and off-diagonal processors. In the following clock cycle, the updated matrix elements are exchanged among adjacent processors such that other elements in the matrix can be sent to the diagonal processors for reduction.

### 3.3 Dual processor architecture of approximate Jacobi method

The systolic architecture proposed in the previous section targets applications that require massive throughput of eigenvalue computation. On the other end of the spectrum, there are systems that can only spare limited hardware resources to a dedicated hardware module for eigenvalue computation. Thus, an alternative architecture is proposed which employs just one diagonal processor and one off-diagonal processor to carry out the approximate Jacobi method. Since the method only requires two processors to compute eigenvalues regardless of the matrix size, this architecture will be referred to as dual processor architecture in this paper.

In comparison with the systolic array architecture, the diagonal processor in the dual processor architecture sequentially performs all the operations that are performed in parallel by the multiple diagonal processors in the systolic architecture, where the off-diagonal processor performs all the operations that are performed in parallel by the multiple off-diagonal processors.

The dual processor architecture is illustrated in Fig. 5. The matrix elements are stored in registers and the appropriate elements are multiplexed to the two processors. Both processors are identical to those in the systolic architecture. While the diagonal processor computes \( I \) and \( \text{sign}(\tau) \) signals, the off-diagonal processor stays idle and it commences computation as soon as the signals are generated and received from the diagonal processor. The control unit oversees the registers, the multiplexer and the two processors.

### 3.4 Discussion on fixed-point implementation

In the proposed hardware implementation, the elements of the input matrix are assumed to be in the range of \(-1 \leq I_1 \leq 1 - 2^{-b+1}\), where \( b \) is the wordlength of the input. Hence, the most significant bit is the sign and the rest of the bits represent the fractional part of the number. If the range assumption is violated, that is, if any element in the input matrix exceeds the range, the matrix must be normalised to the range through dividing every element in the matrix by a constant ensuring that the largest positive number is \( \leq 1 - 2^{-b+1} \) and the smallest negative number is \( \geq -1 \). The computed eigenvalues should be scaled back by multiplying with the same constant to reflect the eigenvalues of the input matrix at the original scale.

From [21], the absolute value of any eigenvalue \( \lambda \) of a matrix \( A \) is bounded from above by a function of its elements as shown in (14), where \( \text{ABS}(\cdot) \) returns the absolute value of every element in the matrix. \( \text{SumOfRow} \) returns a vector formed by summing the rows of the matrix

\[ |\lambda| \leq \max (\text{SumOfRow}(\text{ABS}(A))) \quad (14) \]

The input matrix element range of \(-1 \leq I_1 \leq 1 - 2^{-b+1}\) imposes an upper bound to the value of the matrix eigenvalues. For example, for an input matrix \( 4 \times 4 \), the absolute value of any of its eigenvalues is \( \leq 4 \). Therefore two extra bits need to be assigned in the data-path of the hardware system to avoid overflow or underflow in the \( 4 \times 4 \) case, and in general \( \lceil \log n \rceil \) extra bits are required for an \( n \times n \) input matrix.

To clarify the notation, the operations involved in reducing the entire set of matrix elements is customarily called a sweep. With finite fixed-point accuracy, experiments demonstrated that although \( F(A) \) in (2) decreases to a certain minimum with approximate Jacobi rotations, the eigenvalues computed with excessive sweeps depart from the actual
eigenvalues. This is because of the propagation of the truncation error to the next sweep. The instability of eigenvalues can be rectified by executing rounding instead of truncation. This improvement can be seen from Fig. 6, where the maximum absolute error is plotted against the number of sweeps by simulating 3000 randomly generated 4 × 4 matrices in a 16-bit system. The matrix elements are scaled to lie in the range [−1, 1]. From the graph, it is clear that the rounding scheme stabilises the eigenvalue result. However, the overhead in terms of area incurred by the rounding operation is considerable. For 4 × 4 16-bit systems, the area increases by 27% over to a design that uses truncation, for the proposed architecture.

For some hardware devices, for example FPGAs, there are available hard-core multipliers which can be exploited by the proposed architecture. The availability of these blocks is beneficial since the scaling in the off-diagonal processors is achieved through multiplication. For example, experiments have demonstrated that in a 4 × 4 16-bit system, two 18 × 18 embedded multipliers are traded for 1343 slices.

4 Algebraic method

For the special case of a 3 × 3 real symmetric matrix \( \mathbf{M} \), the eigenvalues can be calculated analytically. Given

\[
\mathbf{M} = \begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix}
\]

the eigenvalues can be expressed as roots of a third-degree characteristic polynomial \( p(\lambda) = \text{det}(\mathbf{M} - \lambda \mathbf{I}) \) [13]. Let this characteristic equation be

\[
\lambda^3 + k\lambda^2 + l\lambda + m = 0 \tag{15}
\]

where

\[
k = -(a + b + c) \tag{16}
\]

\[
l = ab + bc + ac - f^2 - e^2 - d^2
\]

\[
m = af^2 - abe - 2fde + bce^2 + cd^2
\]

By substituting \( \lambda = x - k/3 \), we remove the \( \lambda^2 \) term in (15)

\[
x^3 + px + q = 0 \tag{17}
\]

where

\[
p = -\frac{1}{3}k^2 + l, \quad q = \frac{2}{27}k^3 - \frac{1}{3}lk + m \tag{18}
\]

The condition of symmetric matrix \( \mathbf{M} \) assures that all eigenvalues are real numbers [13]. Following [11], we substitute \( x = \sqrt{-4p/3}y \), yielding

\[
4y^3 - 3y = \frac{3q}{p\sqrt{-4p/3}}
\]

Using the trigonometric relation \( \cos 3\psi = 4\cos^3 \psi - 3\cos \psi \) and setting \( y = \cos \psi \), we have

\[
\cos 3\psi = \frac{3q}{p\sqrt{-4p/3}}
\]

The three real solutions to (17) are computed and the eigenvalues of \( \mathbf{M} \) are

\[
\lambda_1 = x_1 - \frac{k}{3} = \beta \cos \psi - \frac{k}{3}
\]

\[
\lambda_2 = x_2 - \frac{k}{3} = \beta \cos \left( \psi + \frac{2\pi}{3} \right) - \frac{k}{3}
\]

\[
\lambda_3 = x_3 - \frac{k}{3} = \beta \cos \left( \psi + \frac{2\pi}{3} \right) - \frac{k}{3}
\]

where

\[
\beta = \sqrt{-\frac{4p}{3}}, \quad \psi = \frac{1}{3} \left( \pi - \arcsin \frac{3q}{p\beta} \right) \tag{19}
\]

The range of \( \psi, \psi \in (0, \pi/3) \), imposes an ordering of the acquired eigenvalues which is \( \lambda_1 \geq \lambda_2 \geq \lambda_3 \). This provides the designer of the system the flexibility to select only the desired eigenvalues.

4.1 Hybrid arcsin CORDIC algorithm

The algebraic method for the computation of eigenvalues requires the computation of a wide division followed by an arcsin function shown in (19), which is an expensive block with regard to a hardware implementation.

We propose a new approach based on the CORDIC algorithm that avoids the wide division and performs the arcsin function simultaneously in (19). Specifically, \( \alpha \) in (20) needs to be computed

\[
\alpha = \arcsin \left( \frac{q}{p} \right) \tag{20}
\]

![Figure 6](image.png)

**Figure 6** Maximum absolute error in the eigenvalues using rounding and truncation schemes

78 The Institution of Engineering and Technology 2009
The proposed approach begins by placing a vector of magnitude $p$ on the x-axis as shown in Fig. 7. This vector is rotated according to the rules governed by the CORDIC iteration equations. The rotations finish when the vector's y component is equal to $c$, the choice of which will be determined conveniently to solve the problem laid in (20). We introduce $r_i$ to denote the magnitude of the vector $[x_i, y_i]$, that is, $r_i = \sqrt{(x_i^2 + y_i^2)}$ at each rotation $i$.

The initial conditions depicting the vector on the x-axis are as follows

$$x_0 = p$$
$$y_0 = 0$$
$$z_0 = 0$$
$$r_0 = \sqrt{(x_0^2 + y_0^2)} = p$$

The direction to which the vector is rotated in the next iteration is decided by $d_i$, which is the result of a comparison between $c$ and the y component of the rotated vector at the current iteration $i$ (25)

$$d_i = \text{sign}(c - y_i)$$

After $n$ iterations, $r_0$ has been scaled by $A_n$ and the CORDIC algorithm produces

$$r_n = A_n \cdot r_0 = A_n \cdot p$$
$$x_n = \sqrt{(A_n \cdot p)^2 - c^2}$$
$$y_n = c$$
$$z_n = \arcsin \left( \frac{c}{A_n \cdot p} \right)$$
$$A_n = \prod_{i=1}^{n} \frac{1}{K_i} = \prod_{i=1}^{n} \sqrt{1 + 2^{-2i}}$$

The above steps are illustrated in Fig. 7. By choosing $c$ to be $c = A_n \cdot q$, the result $z_n$ is given by (26), which is the desired function. This concludes the conventional arcsin CORDIC algorithm

$$z_n = \arcsin \left( \frac{A_n \cdot q}{A_n \cdot p} \right) = \arcsin \left( \frac{q}{p} \right)$$ (26)

Effectively, each CORDIC iteration rotates the vector by an angle $\theta_i$ and scales the magnitude of the vector $r_i$ by $K_i$, where $\theta_i = \arctan 2^{-i}$ and $K_i = \sqrt{1 + 2^{-2i}}$, respectively. Because of the scaling in magnitude, this computation can, nonetheless, cause incorrect decision-making, that is, $d_i$ to take the wrong value, if this scaling is not compensated. As a result of such incorrect decision-making, the vector $r_i$ rotates away from the desirable $a$ and the arcsin result in angle accumulation would be incorrect. If the scaling is compensated in each iteration, the aforementioned decision error can be resolved. This is essentially a general rotation, which is defined as vector rotation without changing the vector’s magnitude. In terms of the required computations, general rotations are more expensive, since each general rotation is equivalent to carrying out a conventional shift-and-add CORDIC rotation as well as a multiplication for scaling compensation.

It is observed that the resultant angle computed by the conventional arcsin CORDIC algorithm departs from the true angle more significantly when the incorrect decision is made earlier. Recalling from (7), the $\theta_i$ is getting smaller with an increasing $i$. If correct decision-making is ensured for the first few rotations by resorting to a general rotation and then applying the conventional CORDIC algorithm afterwards, a reasonable trade-off between computation and accuracy can be achieved. This algorithm is referred to as a hybrid arcsin algorithm for the rest of the paper. Fig. 8 illustrates how the average error in the obtained arcsin angle declines by increasing the number of general rotations.

In principle, accuracy is attained from the general rotation and a low computation cost is achieved from the CORDIC algorithm. Fig. 9 demonstrates the greatly reduced error achieved by the hybrid arcsin algorithm compared with the conventional one.
4.2 Architecture of the algebraic method

A pipelined architecture based on the feedforward nature of the algebraic method is proposed. Without the loss of generality, the design computes the smallest eigenvalue of a $3 \times 3$ matrix. The overview of the design is shown in Fig. 10. The ALU module computes the values of $p$, $q$ and $k/3$ from (18) and (16), respectively. This module is the very first stage of the system, and so the wordlengths of output variables are determined such that an early stage truncation error is avoided. To be specific, the wordlength is doubled when there is multiplication and extended by 1 bit when there is an addition or subtraction. The existence of common subexpressions is explored to reduce the number of operators in the ALU.

The square root module adopts the successive approximation algorithm [22], which is an iterative process that achieves 1 bit of accuracy per iteration. The algorithm works by repeatedly halving the range of values in which the square root of a value is known to exist. For example, to find the square root of $N$, the initial range is $N$ and it starts with the mid point $N/2$. It then compares $(N/2)^2$ with $N$ and the result decides in which half the square root lies. The search range is halved after each iteration and, eventually, the square root of desired precision is acquired. In our implementation, the square root module is completely unrolled, so that it can be pipelined to provide one square root result every clock cycle.

The hybrid arcsin module implements the hybrid arcsin algorithm described in the last subsection.

5 Performance evaluation of architectures

In this section, the methods described in this paper, namely the exact, window, and approximate Jacobi methods implemented in both systolic and dual processor architecture are evaluated. Moreover, the pipelined architectures of the approximate Jacobi method and the algebraic method for a $3 \times 3$ matrix are compared and evaluated. The implementation language is Handel-C [23], which is a high-level language for hardware description. The synthesis tool used is the Xilinx Integrated Software Environment (ISE). The target device is the Xilinx XC2V6000-6 Virtex-II FPGA chip [24]. In the rest of this paper, the $N$-bit system implies that the inputs are $N$-bit fixed-point numbers. Unless stated otherwise, the input matrix is assumed to be $4 \times 4$ with an accuracy of 16-bit. It should be noted that all the methodologies were compared keeping the accuracy of the calculated eigenvalues the same.

Both the exact and window Jacobi methods require a series of CORDIC rotations that have no data dependency on each other. Thus, they can be executed in parallel and each rotation can have a dedicated CORDIC module to take full advantage of the available resources in order to maximise the throughput. On the other hand, these CORDIC modules can be shared within the same processor to save hardware resource (i.e. area). To investigate the impact of CORDIC module sharing, two configurations are applied to the exact and window Jacobi methods: (a) dedicated CORDIC modules and (b) shared CORDIC modules. Note that in the approximate Jacobi method, the rotations are performed with only one CORDIC iteration and, thus, CORDIC module sharing is irrelevant.

5.1 Area

Table 2 summarises the area required for the hardware implementation by the various methods under both architectures. The results demonstrate that the approximate Jacobi method requires the smallest area of all and the window Jacobi method requires a larger area than the exact Jacobi as expected from Section 2.4. However, the areas required by the systems under consideration do not exhibit large variation.

With respect to the approximate Jacobi systolic architecture using dedicated CORDIC modules, Table 3 provides a glimpse of how the area scales with the wordlength of the system as well as with the matrix size. Further experimental data show that the area of the approximate Jacobi systolic architecture scales linearly with the wordlength of the input variables, something that was expected because of the nature of the architecture.

Table 4 summarises the area occupancy for each processor under the different methods and architectures for a 16-bit
system. Thus, given a matrix size of \( n \times n \), an estimate of the total required area of the system can be made by summing the areas of the processors according to (27), where \( \text{Area}_D \) is the area occupied by a diagonal processor and \( \text{Area}_O \) the area occupied by an off-diagonal processor.

\[
\text{Area}(n \times n) = \frac{n}{2} \cdot \text{Area}_D + \frac{n/2((n/2) - 1)}{2} \cdot \text{Area}_O \quad (27)
\]

### 5.2 Maximum frequency

The maximum achieved frequency of each architecture was obtained by Xilinx ISE post-place and route static timing analysis [24] and they are shown in Table 2. The lower maximum frequency of the dual processor architecture compared with that of the systolic architecture is explained because of the more complicated routing that is required. The exact and window Jacobi-based architectures produce similar results, whereas the approximate Jacobi method produced inferior results (around 14% slower).

### 5.3 Number of clock cycles per step

The number of clock cycles required for one step of each method with regard to the systolic architecture with dedicated CORDIC modules is shown in (28), (29) and (30), where \( b \) indicates the input wordlength.

\[
T_{\text{exact}} = 15 + 3 \cdot \frac{1}{2} b \quad (28)
\]

\[
T_{\text{window}} = 16 + 3W + \frac{1}{2} b \quad (29)
\]

\[
T_{\text{approximate}} = 15 + \log_2 \left( \frac{1}{2} b \right) \quad (30)
\]

The exact Jacobi method takes \( b \) CORDIC iterations to compute the cosine–sine pair in (6) and \( 2b \) CORDIC rotations to update the matrix. The approximate Jacobi method needs significantly fewer clock cycles. It takes at the most three comparisons to compute \( i \), which is the cosine–sine counterpart. Only one iteration of CORDIC is needed for every approximate rotation. Thus, while the number of clock cycles per step of the exact Jacobi method scales linearly with wordlength \( b \) as in (28), it scales only logarithmically for the approximate Jacobi method (30). In the window Jacobi method, each step can have a different window size, which results in a different number of CORDIC iterations. In this work, \( W \) is used to denote the average window size throughout the eigenvalue computation. For the 16-bit \( 4 \times 4 \) matrix, where the window is chosen to be 8 for the first six steps and 16 for another three steps, \( W \) would be 10.67. Table 2 summarises the number of required clock cycles per step for this particular case. The results indicate that the approximate Jacobi method requires considerably fewer clock cycles per step.

### 5.4 Area-throughput design space

To render eigenvalues of the same accuracy, the approximate Jacobi method [10] requires more steps than the exact Jacobi Method or the window Jacobi Method. For a 16-bit \( 4 \times 4 \) matrix, the approximate Jacobi method requires 12 steps against nine for the exact and window Jacobi methods. Nevertheless, as shown in Table 2, the time saving per step for the approximate Jacobi architecture is so significant that it outweighs this drawback of slow convergence.

<table>
<thead>
<tr>
<th>Method</th>
<th>Architecture</th>
<th>Dedicated CORDIC blocks</th>
<th>Shared CORDIC blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Slices</td>
<td>Frequency (MHz)</td>
<td>Clock cycles</td>
</tr>
<tr>
<td>exact Jacobi</td>
<td>systolic</td>
<td>4852</td>
<td>88.6</td>
</tr>
<tr>
<td></td>
<td>dual processor</td>
<td>3872</td>
<td>84.7</td>
</tr>
<tr>
<td>window Jacobi</td>
<td>systolic</td>
<td>6459</td>
<td>85.0</td>
</tr>
<tr>
<td></td>
<td>dual processor</td>
<td>3627</td>
<td>85.0</td>
</tr>
<tr>
<td>approximate Jacobi</td>
<td>systolic</td>
<td>4540</td>
<td>73.1</td>
</tr>
<tr>
<td></td>
<td>dual processor</td>
<td>3382</td>
<td>74.1</td>
</tr>
</tbody>
</table>

The clock cycles columns indicate the number of clock cycles per step for each architecture.
We adopt ‘throughput’ as the speed metric, which indicates the number of eigenvalues generated per second. This metric unifies the maximum frequency, the number of clock cycles per step and the number of steps per eigenvalue calculation. The area-throughput design space for all the three methods in both architecture variants is illustrated in Fig. 11. For a given area, the approximate Jacobi method generates more than twice the eigenvalues per second than the other methods. It unambiguously surpasses the rest of the methods, which have comparable performance.

Moreover, experiments were performed to compare the throughput of the designs under investigation with the performance of a graphical processing unit (GPU). Recently, GPUs have gained a lot of interest from the HPC community because of their simple programming model and high performance that can be achieved because of their many computational cores. The target device was an NVIDIA GeForce 8600 GTS. Fig. 12 shows the achieved throughput (number of eigenvalues per second) against the matrix size. It should be noted that the input matrices were symmetric and tridiagonal, having elements in the same range as the FPGA designs, and the target precision was set to the precision that is achieved by a 16 bit system (approximate Jacobi algorithm). Implementation details can be found in [25–27]. The time measurements do not include the time to transfer the data. From Fig. 12, it can be concluded that even though the required time for transforming a symmetric matrix to the tridiagonal form is not included, the GPU achieves a throughput of around 44 000 eigenvalues per second for a 4 × 4 matrix, which is about 31 times less than the throughput achieved by the approximate systolic architecture in an FPGA.

To achieve a clearer overview of the efficiency of each architecture, we employ an intuitive efficiency metric defined in (31). ‘Efficiency’ is defined as the number of eigenvalues generated every second, given one unit of hardware (one FPGA slice in our implementation)

\[
\text{Efficiency} = \frac{\text{Throughput}}{\text{Area}} \tag{31}
\]

From the point of view of efficiency, the approximate Jacobi method in the systolic architecture has a clear advantage.
over the other methods as demonstrated in Fig. 13. In contrast to the theory, the window Jacobi method is not superior to the exact Jacobi method because the reduction in the number of CORDIC rotations is not substantial enough to justify the extra hardware overhead incurred. In terms of resource sharing, the dual processor architecture offers a better platform than the systolic architecture. For both the exact and approximate Jacobi methods, the ‘dual processor architecture with dedicated CORDIC modules’ is slightly more efficient than the ‘systolic architecture with shared CORDIC modules’. The ‘dual processor architecture with shared CORDIC modules’ requires the least area as shown in Table 2, but it is less efficient than the others because of a larger proportion of hardware that is used for auxiliary functions rather than the core eigenvalue computation.

5.5 Pipelined architectures for 3 × 3 symmetric matrices

In many applications, such as real-time optical flow computation [2], a massive throughput of eigenvalues of 3 × 3 matrices is required. The performance evaluation section concluded that the approximate Jacobi method is the most efficient architecture of the Jacobi-based eigenvalues framework. Thus, it was used as a reference architecture to evaluate the performance of the algebraic method for the special case of a 3 × 3 symmetric matrix. Both architectures have been pipelined and designed to return the smallest eigenvalue of a matrix at every clock cycle.

Table 5 summarises the obtained results from the comparison between the algebraic method and the approximate Jacobi method for 13-bits systems. It should be noted that the approximate Jacobi method is unrolled and pipelined and its behaviour is equivalent to three sweeps of its un-pipelined version.

The results in Table 5 show that the implementation of the algebraic method is considerably smaller than that of the approximate Jacobi method. For applications that can allow this type of error in the computation of the eigenvalues, for example, computer vision applications, the pipelined algebraic architecture provides a good choice compared with the current Jacobi-like based architectures since it provides a solution with similar performance but using considerably less hardware resources. However, there is no straightforward method to extend the algebraic method for the computation of the eigenvalues to matrices with a size larger than 3 × 3.

6 Conclusion

This paper explores the systolic and dual processor architectures for eigenvalue computation based on the approximate Jacobi method. For comparison purposes, the exact and window Jacobi methods are also implemented and evaluated. In terms of area and throughput, experimental results demonstrate that the approximate Jacobi method, when mapped to a systolic architecture, is the most efficient system in terms of throughput per unit area. For certain types of applications that require the computation of the eigenvalues of a 3 × 3 symmetric matrix, a pipelined architecture resorting to the algebraic method is proposed. The evaluation showed that the proposed algebraic method achieves a throughput comparable to the approximate Jacobi method using only 22% of the area. Future work will focus on the error analysis and numerical stability of the eigenvalue computation methods reported in this work.

7 Acknowledgments

This work was funded by the UK Research Council under the Basic Technology Research Programme ‘Reverse Engineering Human Visual Processes’ GR/R87642/02.

8 References


