2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

An FPGA Implementation of the Hestenes-Jacobi Algorithm for Singular Value Decomposition

Xinying Wang and Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University, Ames, Iowa, USA
Email: {xinying, zambreno}@iastate.edu

***Abstract*—As a useful tool for dimensionality reduction, Singular Value Decomposition (SVD) plays an increasingly significant role in many scientific and engineering applications. The high computational complexity of SVD poses challenges for efficient signal processing and data analysis systems, especially for time-sensitive applications with large data sets. While the emergence of FPGAs provides a flexible and low-cost opportunity to pursue high-performance SVD designs, the classical two-sided Jacobi rotation-based SVD architectures are restricted in terms of scalability and input matrix representation. The Hestenes-Jacobi algorithm offers a more parallelizable solution to analyze arbitrary rectangular matrices; however, to date both FPGA and GPU-based implementations have not lived up to the algorithm's potential. In this paper, we introduce a floating-point Hestenes-Jacobi architecture for SVD, which is capable of analyzing arbitrary sized matrices. Our implementation on an FPGA-based hybrid acceleration system demonstrates improved efficiency of our architecture compared to an optimized software-based SVD solution for matrices with small to medium column dimensions, even with comparably large row dimensions. The dimensional speedups can be achieved range from $3.8\times$ to $43.6\times$ for matrices with column dimensions from 128 to 256 and row sizes from 128 to 2048. Additionally, we also evaluate the accuracy of our SVD process through convergence analysis.**

***Keywords*-Architecture, FPGA, Singular Value Decomposition, Hestenes-Jacobi Algorithm.**

I. Introduction

In many real-world applications, data dimensionality is rapidly growing. Principal Component Analysis (PCA) is widely employed to reduce the dimensions of data in high-dimensional spaces. Among the classical solutions for PCA, Singular Value Decomposition (SVD) is the most popular technique to approximate high-dimensional data through orthogonal transformations. SVD-based PCA has been used in many signal processing applications such as image processing, computer vision, pattern recognition and remote sensing [1]–[3]. However, SVD is a computationally-expensive procedure, which makes its use unlikely to meet the requirements of many time-sensitive designs, especially when it is processed iteratively in those applications. For instance, in the application of video surveillance [4], it takes 185.2 seconds to recover the square matrix with the dimensions of 3000 through running partial SVD 15 times, which makes it difficult to satisfy stringent real-time performance requirements. As data dimensionality is increasing continuously, the runtime of SVD is likely to have further substantial growth.

The SVD operation diagonalizes an arbitrary $m \times n$ matrix through a series of orthogonal transformations [5]. Optimized software implementations (e.g., MATLAB, LAPACK) employ the Householder transformation [6] to perform SVD computation; however, their performance is restricted by their inherent computational complexity and high data dependency. Highly parallel accelerators such as Graphic Processing Unit (GPUs) and multi-core platforms have been employed to explore parallel implementations, although these previous works only achieved speedups when the input matrices have dimensions greater than 1000 [7], [8].

In the reconfigurable architecture community, systolic-arrays have been implemented on modern FPGAs to compute the classic two-sided Jacobi rotations [9], [10]. Although improved performance has been demonstrated, the scalability of those implementations are limited, and the designs are restricted to only handle square input matrices.

Hestenes [10] discovered the equivalence between zeroing out an off-diagonal a_{ij} and orthogonalizing the ith and jth vectors through plane rotation. Instead of annihilating every non-zero off-diagonal element by rotating 2×2 matrices, the Hestenes-Jacobi method is capable of decomposing an arbitrary $m \times n$ non-square matrix through vector computations. GPUs and FPGAs have been employed to evaluate parallel Hestenes-Jacobi designs; however, the performance has suffered from the iterative thread synchronizations (in the case of GPUs [11]) and repeated calculations (in the case of FPGA implementations [12]).

In this paper, we present an FPGA-based hardware design of the Hestenes-Jacobi algorithm for SVD with floating-point arithmetic, which attempts to analyze an arbitrary $m \times n$ matrix. Compared to a previous FPGA-based Hestenes-Jacobi implementation [12], our architecture optimizes the calculations through improving data reuse, and employs IEEE-754 double precision floating-point operators to provide a wider dynamic range. Also, off-chip memory is employed to break the restriction of the analyzable matrix dimensions. Our experimental results have demonstrated the efficiency of our design for matrices with small to medium column dimensions, even when they have comparably large row dimensions. The dimension-dependent speedups that can be achieved range from $3.8\times$ to $43.6\times$ for matrices with column sizes from 128 to 256 and row dimensions from 128 to 2048. Compared to other GPU-based and FPGA-based implementations of

Hestenes-Jacobi SVD, our architecture is currently the fastest in terms of overall performance. We have also evaluated the accuracy of our approach through analysis of the convergence properties.

The remainder of this paper is organized as follows. Section II introduces the theoretical background of SVD, while related work is discussed in Section III. Section IV presents the modified Hestenes-Jacobi algorithm, and Section V introduces our novel hardware architecture for Hestenes-Jacobi algorithm, including detailed descriptions of individual components. The evaluation of performance and accuracy for our architecture is presented in Section VI. Finally, the paper is concluded in Section VII with a preview of future planned work.

II. Theoretical Background

A. Singular Value Decomposition (SVD)

The Singular Value Decomposition transforms an $m \times n$ matrix into a product of an $m \times m$ orthogonal matrix, an $m \times n$ diagonal matrix with singular values and the transpose of an $n \times n$ orthogonal matrix [5] in the form of eq. (1).

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n} \tag{1}$$

B. Classic Two-sided Jacobi Rotations

Jacobi rotations are widely used in diagonalizing matrices. To perform Jacobi rotation, Jacobi rotation matrices J^l and J^r are applied to the matrix from both sides as shown in eq. (2). By applying the Jacobi rotation matrices to a 2×2 matrix, the off-diagonal elements are annihilated as in eq. (3).

$$A_{i+1} = J^l_i A_i J^r_i \tag{2}$$

$$J^{l'} \cdot \begin{pmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{pmatrix} \cdot J^r = \begin{pmatrix} A^{"}_{pp} & 0 \\ 0 & A^{"}_{qq} \end{pmatrix} \tag{3}$$

The Jacobi rotation matrices are generated through eq. (4) and eq. (5), where θ represents plain rotation angles α or β [9].

$$\begin{cases} J_{pp} = cos(\theta); & \\ J_{pq} = sin(\theta); & (p < q) \\ J_{qp} = -sin(\theta); & (p < q) \\ J_{qq} = cos(\theta); & \\ J_{ii} = 1; & (i \neq p,q) \\ J_{ij} = 0,\ Others. & \end{cases} \tag{4}$$

$$\beta{+}\alpha{=}\arctan\left(\frac{A_{qp}+A_{pq}}{A_{qq}-A_{pp}}\right) \qquad \beta{-}\alpha{=}\arctan\left(\frac{A_{qp}-A_{pq}}{A_{qq}+A_{pp}}\right) \tag{5}$$

To process SVD, Jacobi rotations are calculated on every 2×2 matrix to zero out all the non-zero off-diagonal elements. The calculation of an independent 2×2 Jacobi rotation only affects two rows and columns of a matrix, which provides an opportunity for parallel designs through simultaneously performing independent 2×2 Jacobi rotations. Due to the nature of 2×2 Jacobi rotations, the input matrix is restricted to square dimensions.

C. Hestenes-Jacobi Method

In [10], Hestenes observed that the annihilation of a matrix element is equivalent to orthogonalizing two column vectors. Instead of directly annihilating non-zero off-diagonal elements, the Hestenes-Jacobi algorithm (also known as the one-sided Jacobi method) performs the matrix decomposition through iterative orthogonal transformations between every pair of vectors. In the Hestenes-Jacobi method, the matrix is orthogonalized by columns through post-multiplying an orthogonal matrix, which is generated through a product of plane rotations as in eq. (6).

$$A \cdot V = B, \qquad where\ b^T_i \cdot b_j = 0 \tag{6}$$

Next, matrix B is further normalized through the equation $B = B \cdot \Sigma^{-1} \cdot \Sigma$, in which Σ is a diagonal matrix with the squared column norms as diagonal elements. Then, by setting $U = B \cdot \Sigma^{-1}$, eq. (6) can be rewritten as eq. (7), which is the result form of SVD.

$$A \cdot V = U \cdot \Sigma \quad \longleftrightarrow \quad A = U \cdot \Sigma \cdot V^T \tag{7}$$

Compared to the classic two-sided Jacobi rotation approach, the Hestenes-Jacobi method is capable to analyze an arbitrary rectangular matrix.

III. Related Work

In recent years, the significant surge of data dimensionality has made the application of SVD seem ubiquitous [13]. To compute SVD, the Householder transformation-based method and the Jacobi rotation-based method have demonstrated satisfied stability and accuracy [14], [15]. The Householder transformation [6], [16] is capable of efficiently bi-diagonalizing matrices through vector computations, which is then followed by iterative implicit QR factorization [17] or divide-and-conquer iterations [18] for generating singular values. In Householder transformation-based method, the SVD process is dominated by the calculations of Householder vectors and their respective updates, whose performance improvement is challenged by the inherent data dependency. To parallelize the Householder transformation, implementations have been demonstrated on GPUs [7], [11] and multi-core platforms [8], in which possible accelerations of GPU-based designs are achieved only for matrices with significantly large dimensions due to the iterative thread synchronization, while, the performance of implementation on multi-core platform is dominated by the task splitting and time consumption of communications.

The emergence of reconfigurable fabrics such as FPGAs introduces low-cost solutions to parallelize the algorithm at the operand-level granularity. To perform SVD, 1-dimensional or 2-dimensional systolic arrays have been employed to parallelize the classic two-sided Jacobi rotation algorithm [9], [19]–[21]. With the featured independent 2×2 rotations, a highly parallel 2-dimensional systolic array is employed to map the classic two-sided Jacobi rotation algorithm into FPGA devices with the computational complexity of O($n \log n$) for an *n*-by-*n* square matrix. However, to fit the architecture on a single chip, the scalability is limited, as n^2 processing elements (PEs) is needed by the systolic array implementation.

Compared to the classical Jacobi rotation approach, the Hestenes-Jacobi algorithm provides a more flexible solution to analyze the rectangular matrices. To explore the high performance SVD design, FPGAs and GPUs have been employed to demonstrate the parallel implementations of the Hestenes-Jacobi SVD algorithm [10]; however, the performance has suffered from iterative thread synchronizations for the implementation on GPUs [11], and the iterative design with duplicated computations in the case of FPGA implementation [12].

IV. Modified Hestenes-Jacobi Algorithm

As previously mentioned, the Hestenes-Jacobi algorithm computes the SVD through orthogonalizing every pair of vectors. Instead of directly performing element-wise operations to annihilate an off-diagonal, the Hestenes-Jacobi method applies orthogonal transformation between the two vectors whose indexes are equal to the row and column indexes of that off-diagonal element. To orthogonalize a pair of vectors, Jacobi rotation is computed with the squared 2-norms of the vectors and the covariance between them.

In the Hestenes-Jacobi process (detailed in Algo. 1), the orthogonalization between two column vectors is started with the calculation of their squared 2-norms and respective covariance. Then, Jacobi rotation is performed with the calculated squared 2-norms and covariance, after which, the elements in those two column vectors are updated by applying the generated rotation angle parameters. At runtime, pairwise orthogonalizations are performed iteratively until the satisfied convergence is reached. The singular values are obtained as the square root of the diagonal elements in the resulted matrix.

To optimize the algorithm by reducing the amount of computations, the squared 2-norms of rotated vectors and their associated covariances are updated directly after each rotation. Thus, the repeated regeneration of squared 2-norms and covariances has become unnecessary. In Algo. 1, matrix D is the covariance matrix, whose diagonal and off-diagonal elements are the squared 2-norms of column vectors and the covariances between them, respectively.

Algorithm 1: Singular value decomposition via Modified Hestenes-Jacobi algorithm

```
Input: matrix $A$
Output: singular value vector $Sig$
1  $R \leftarrow A$
   /* Generating the squared 2-norms of column
      vectors and their associated covariances
      */
2  for $i \leftarrow 1$ to $NumofColumn$ do
3      for $j \leftarrow i$ to $NumofColumn$ do
4          $D_{i,j} \leftarrow R_i^T * R_j$
5  repeat
6      for $i \leftarrow 1$ to $NumofColumn - 1$ do
7          for $j \leftarrow i$ to $NumofColumn$ do
               /* Generating Jacobi rotation angle
                  parameters with squared 2-norms
                  of column vectors and their
                  respective covariance          */
8              $norm_1 \leftarrow D_{j,j}$
9              $norm_2 \leftarrow D_{i,i}$
10             $cov \leftarrow D_{i,j}$
11             $\rho \leftarrow (norm_2 - norm_1)/(2 * cov)$
12             $t \leftarrow sign(\rho)/(|\rho| + \sqrt{1+\rho^2})$
13             $\cos \leftarrow 1/\sqrt{1+t^2}$
14             $\sin \leftarrow cos * t$
               /* Updating the squared 2-norms
                  affected by rotation           */
15             $D_{j,j} \leftarrow D_{j,j} + t * cov$
16             $D_{i,i} \leftarrow D_{i,i} - t * cov$
17             $cov \leftarrow 0$
               /* Updating the covariances
                  affected by rotation           */
18             for $k \leftarrow 1$ to $i - 1$ do
19                 $D_{k,i} = D_{k,i} * \cos - D_{k,j} * \sin$
20                 $D_{k,j} = D_{k,i} * \sin + D_{k,j} * \cos$
21             for $k \leftarrow i + 1$ to $j - 1$ do
22                 $D_{i,k} = D_{i,k} * \cos - D_{k,j} * \sin$
23                 $D_{k,j} = D_{i,k} * \sin + D_{k,j} * \cos$
24             for $k \leftarrow j + 1$ to $NumofRow$ do
25                 $D_{i,k} = D_{i,k} * \cos - D_{j,k} * \sin$
26                 $D_{j,k} = D_{i,k} * \sin + D_{j,k} * \cos$
27 until convergence reached
28 for $i \leftarrow 1$ to $\min(NumofColumn, NumofRows)$ do
29     $Sig_i \leftarrow \sqrt{D_{i,i}}$
```

V. Our Hestenes-Jacobi SVD architecture

The Hestenes-Jacobi SVD process primarily consists of three computations: (1), calculating the squared 2-norms of vectors and the covariances between vector pairs; (2), performing Jacobi rotations with paired squared 2-norms and their respective covariance; (3), updating rotated vector elements and affected covariances.

To implement the Hestenes-Jacobi SVD, we created three components: the *Hestenes preprocessor*, the *Jacobi rotation component* and the *Update operator* (shown in Fig. 1), all of which are pipelined. The Hestenes preprocessor is responsible for computing the squared column 2-norms and the associated covariances. The Jacobi rotation component is used to zero out the covariance through applying plane rotation with its associated vectors. The Update operator is employed to update the affected columns and covariances.

The Hestenes-Jacobi SVD is an iterative diagonalization process, which performs the orthogonal transformations between every pair of columns by numerous iterations to achieve satisfied convergence. To reduce the amount of computations, instead of repeatedly regenerating all the squared 2-norms and covariances in each iteration, our Hestenes-Jacobi SVD architecture calculates all the squared 2-norms and covariances only in the first iteration, and then those squared 2-norms and covariances are directly updated and reused in the subsequent iterations. To reduce the hardware resource usage, the Hestenes preprocessor is reconfigured to function as an

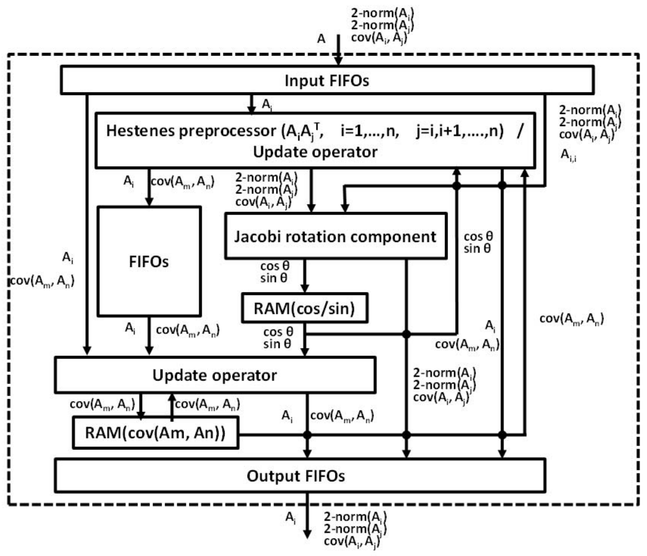

Fig. 1: Block diagram of the Hestenes-Jacobi SVD architecture.

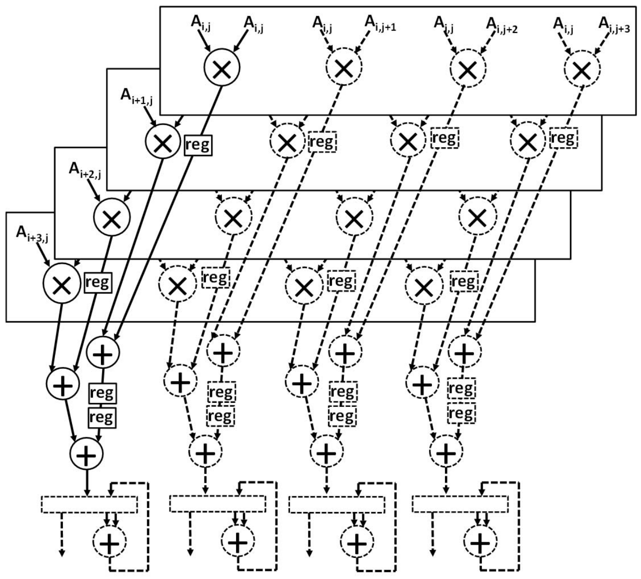

Fig. 2: Example architecture of the Hestenes preprocessor.

additional Update operator after the first iteration. The square-root operator in the Jacobi rotation component is employed to finalize the SVD process, from which the singular values are produced. Besides, as shown in Fig. 1, FIFOs are employed to synchronize the computations and transmit data between the Hestenes preprocessor and the Update operator. Local BRAMs are used to hold the generated rotation angle parameters $\cos$ and $\sin$, and the covariances whose computations have not been completed with the current vector pairing.

A. Hestenes Preprocessor

The Hestenes preprocessor is responsible for calculating the squared column 2-norms and the covariances between column vectors, in which $A_i^T * A_j$ is computed. Considering the overall system performance, we have to balance the amount of parallel computation with the I/O requests. In the Hestenes preprocessor (shown in Fig. 2), multiple layers of pipelined multiplier-arrays are devised, in which operands are reused by all the multipliers successively in a multiplier-array to calculate the partial results of various squared column 2-norms and their related covariances. The resulting product of a multiplier is summed up with the results of its corresponding multiplications across layers, whose operands are the matrix elements from the same columns. For example, in Fig. 2, the matrix element $A_{i,j+3}$ multiplies with $A_{i,j}$ at the first layer, whose product is then added to the product of multiplying $A_{i+1,j+3}$ by $A_{i+1,j}$ at the second layer, the product of multiplying $A_{i+2,j+3}$ by $A_{i+2,j}$ at the third layer, and the product of multiplying $A_{i+3,j+3}$ by $A_{i+3,j}$ at the forth layer, whose sum is the partial result of the covariance between the jth and $(j+3)th$ columns. Meanwhile, $A_{i,j+3}$ moves leftwards to be applied to the adjacent multiplier for multiplication of $A_{i,j+3}$ and $A_{i,j+1}$, whose product is used for computing the covariance between $(j+1)th$ and $(j+3)th$ columns.

The example input for a single multiplier-array with four multipliers is described in Fig. 3, in which the new operand requests for the multiplier array are underlined. The dashed arrows highlights the data movement for reuse and the dashed circles indicate the entered operands, which are reused in later computations. In this case, in a single layer, four double-precision floating-point numbers and at most one double-precision floating-point number are needed as the input for the starting cycle and every subsequent cycle respectively to perform the computations on a sub-vector. Then, the computations on different layers are initialized successively. Thus, 16 cycles are used for the input to obtain the covariance matrix of an 8×8 matrix if 8 layers of multiplier-arrays are equipped. Additional adders are employed to process the accumulations of partial results of covariances and squared 2-norms for vectors with the lengths over 8.

B. Jacobi Rotation Component

Jacobi rotation component performs the orthogonal transformation between two column vectors through a series of operations on their squared column 2-norms and the covariance between them. To calculate the Jacobi rotations, the CORDIC (for COordinate Rotation DIgital Computer) algorithm [22] is a popular choice in the research literature, due to its advantages on efficiently performing complicated trigonometric functions through simple shift-and-add operations. Although CORDIC has been demonstrated as a hardware-efficient algorithm for fixed-point operations, its efficient floating-point implementation is challenged by its inherent bit-width shift-and-add structure. As floating-point arithmetic has become increasingly popular in signal processing applications for its support of a much wider range of values compared to decimal fixed-point

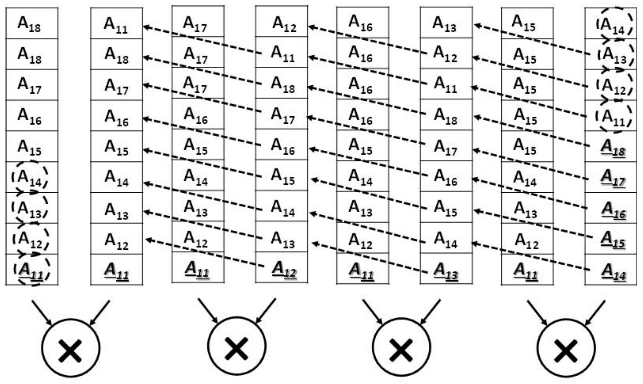
A18 A18 A17 A16 A15 A14 A13 A12 A11
A11 A18 A17 A16 A15 A14 A13 A12 A11
A17 A17 A17 A16 A15 A14 A13 A12 A11
A12 A11 A18 A17 A16 A15 A14 A13 A12
A16 A16 A16 A16 A15 A14 A13 A12 A11
A13 A12 A11 A18 A17 A16 A15 A14 A13
A15 A15 A15 A15 A15 A14 A13 A12 A11
A14 A13 A12 A11 A18 A17 A16 A15 A14

Fig. 3: Example input to a single layer of multiplier-array.

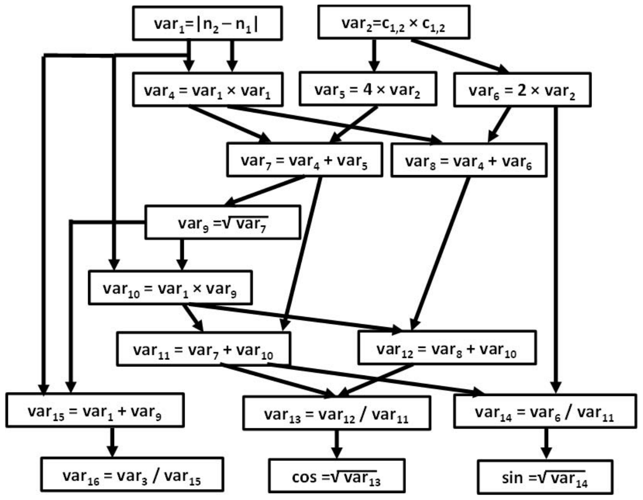

Fig. 4: Dataflow of the Jocobi rotation procedure.

format, our architecture is designed to perform floating-point calculations by using pipelined IEEE-754 double-precision floating-point operators.

As described in Algo. 1, Jacobi rotation of two column vectors is computed with their squared 2-norms and covariance through a series of addition, subtraction, multiplication, division and square-root. The Jacobi rotation equations can be represented as eq. (8), eq. (9), eq. (10), where n_1 and n_2 represents the squared 2-norms of column vectors, while the covariance between them is represented by $c_{1,2}$. The calculated parameter t is then applied to update the squared 2-norms of rotated vectors and zero out their covariance. In Fig. 4, the computations of Jacobi rotation is demonstrated, in which independent calculations can be processed simultaneously. To minimize hardware resource usage, the expensive floating-point computational cores are reused by those calculations. Once all the orthogonal transformations are completed, the square-root operator in the Jacobi rotation component is used to generate the singular values by applying it to the diagonal elements of the processed matrix.

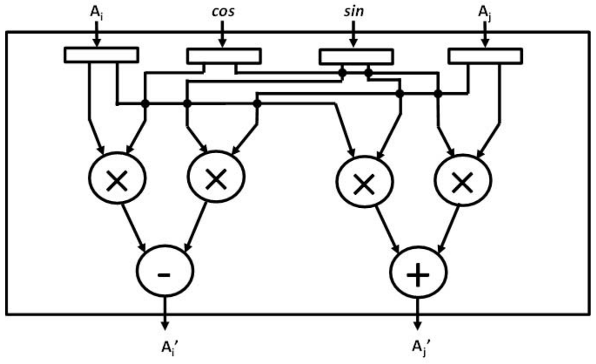

Fig. 5: The architecture of a single update kernel.

$$t = \frac{|2*c_{1,2}|}{|n_2-n_1|+\sqrt{(n_2-n_1)^2+4*c_{1,2}^2}} \tag{8}$$

$$cos = \sqrt{\frac{(n_2-n_1)^2+2*c_{1,2}^2+|n_2-n_1|*\sqrt{(n_2-n_1)^2+4*c_{1,2}^2}}{(n_2-n_1)^2+4*c_{1,2}^2+|n_2-n_1|*\sqrt{(n_2-n_1)^2+4*c_{1,2}^2}}} \tag{9}$$

$$sin = (sign)\sqrt{\frac{2*c_{1,2}^2}{(n_2-n_1)^2+4*c_{1,2}^2+|n_2-n_1|*\sqrt{(n_2-n_1)^2+4*c_{1,2}^2}}} \tag{10}$$

C. Update Operator

$$A_i' = A_i \times cos - A_j \times sin \tag{11}$$

$$A_j' = A_i \times sin + A_j \times cos \tag{12}$$

The Update operator is responsible for updating column elements and covariances which are affected by the processed rotations. Generated rotation angle parameters cos and sin are employed to update the covariances before they are used by later rotations. The update process for a pair of elements contains simple multiplications, additions and subtractions as is shown in eq. 11 and eq. 12. An architecture of a single update kernel is demonstrated in Fig. 5, in which pipelined multipliers, an adder and a subtractor are employed. Multiple update kernels are included in the Update operator. The number of update kernels that can be allocated to a single chip is determined by the resource capacity on the chip. This determines the efficiency of the system, especially for large-scale matrices, where performance is dominated by the amount of updates after each rotation. The convergence of SVD requires the orthogonal transformation of the matrix to be performed in numerous iterations. Both individual column elements and covariances have to be updated in the first iteration, and in the subsequent iterations, only covariances are operated. To optimize the use of hardware resources, the Hestenes preprocessor is able to be reconfigured to function as multiple update kernels.

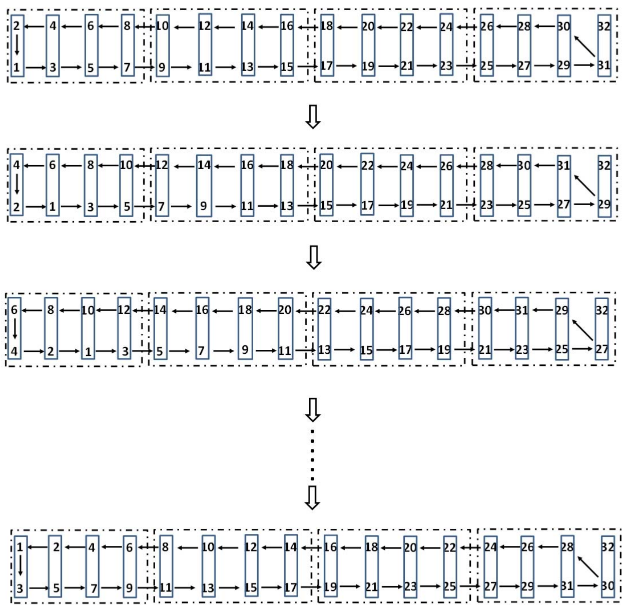

Fig. 6: Demonstration of employed cyclic order for vector pairing.

D. The Cyclic Order for Vector Pairing

The order of vector pairing determines the speed and feasibility of the convergence. In our design, we employ the cyclic ordering, which was demonstrated with the capability of achieving convergence efficiently [9]. In Fig. 6, cyclic ordering is demonstrated with 32 vectors, in which the numbers represent the column indexes, and the arrows indicate the direction for the movement of indexes to form the new vector pairs. In the cyclic ordering, each column has to be paired with every other column. The paired vectors are highlighted by solid boxes in Fig. 6. Besides, due to the limited hardware resources on a single chip, a limited number of vector pairs can be operated simultaneously. In Fig. 6, a dashed box highlights a group of vector pairs, whose computations can be performed in parallel. All the vector-pair groups enter our Hestenes-Jacobi architecture successively.

VI. Experiments and Evaluations

A. Implementation and experimental setup

To evaluate the performance of our Hestenes-Jacobi design, a single Xilinx Virtex-5 XC5VLX330 FPGA on our Convey HC-2 system [23] is used to implement our architecture. In our implementation, double-precision floating-point computational cores are generated by using Xilinx Coregen generator [24]. In the Hestenes preprocessor, four layers of multiplier-array are implemented, in which 16 multipliers and 16 adders are used. To improve the computational intensity on the limited hardware resources, the Hestenes processor calculates all the squared 2-norms and covariances in the first iteration of orthogonalization, and it is then reconfigured as four update kernels with 16 multipliers and 8 adders in the remaining iterations. To perform Jacobi rotation, 1 multiplier, 2 adders, 1 divider and 1 square-root calculators are used, which can start 8 independent Jacobi rotations in every 64 clock cycles. Additionally, an array of eight update kernels are implemented in the Update operator, which contains 32 multipliers and 16 adders or subtractors. The IP core generated computational cores are configured with default latencies as 9, 14, 57, 57 clock cycles for multiplier, adder or subtractor, divider and square-root calculator respectively. Two groups of eight 64-bit width FIFOs are programmed to synchronize the input and output, while a group of eight 127-bit width FIFOs are used for the data transmissions between the Hestenes processor and the Update operator. Simple dual port RAMs are employed to temporarily cache the rotation angle parameters and some covariances. The whole covariance matrix can be stored in the local memory for matrices of column dimension no greater than 256. The system is tested by executing at 150Mhz for 6 iterations, which is believed sufficient for achieving convergence with certain thresholds. Also, a software model is implemented using Matlab to conduct the convergence evaluation.

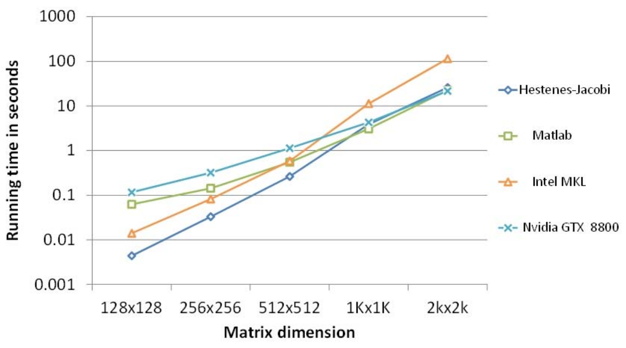

Fig. 7: SVD computation time (in seconds) for square matrices by our Hestenes-Jacobi architecture, Matlab, Intel MKL and GPU

B. Performance analysis

We experimented with both square and rectangular matrices with various dimensions, the performance of which has been summarized in Table I. The experimental results demonstrate that the execution time grows significantly as the number of matrix columns increases, which determines the amount of covariances, whose computation dominates the overall performance. Comparably, the number of rows, which only affects the execution time of the Hestenes preprocessing, has smaller impact on the performance. When the matrix column size grows over 256, the performance is increasingly affected by the I/O bandwidths due to the increased covariance communications have to be made between our Hestenes-Jacobi architecture and off-chip memory.

Comparisons of execution times have been made between our implementation and experimental results from the published literature [7] as well as a Matlab SVD routine. In

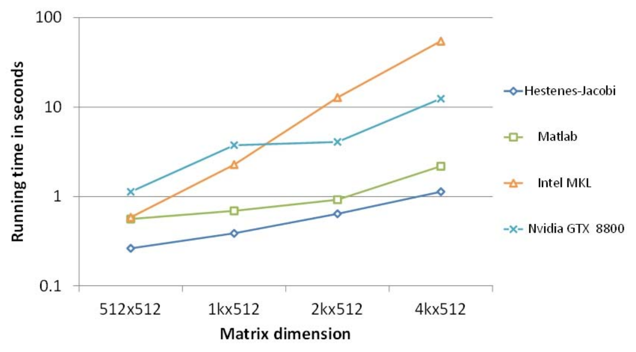

Fig. 8: SVD computation time (in seconds) for rectangular matrices by our Hestenes-Jacobi architecture, Matlab, Intel MKL and GPU

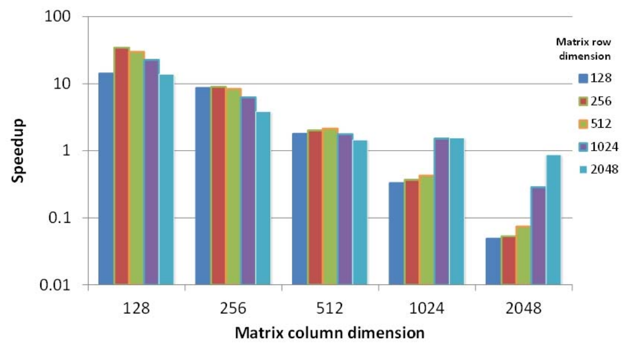

Fig. 9: Speedup of our Hestenes-Jacobi SVD compare to Matlab SVD

Fig. 7 and Fig. 8, the performance of our design, the Matlab 7.10.0 SVD routine running on a 2.2 GHz dual core Intel Xeon processor, SVD solutions with Intel MLK 10.0.4 and NVIDIA 8800 GPU with 128 stream processors [7] have been demonstrated. By analyzing those data points in Fig. 7, our architecture has better efficiency than other software solutions when matrix with dimensions under 512, and our execution slows down when the dimensions over 512 due to the limits of our chosen platform's I/O throughput. In Fig. 8, the comparison is made between matrices with identical column dimensions but various row sizes, which indicates the growth of row number causes a relatively slow increase of the execution time due to the quantity of covariances is determined by the column size.

In Fig. 9, the dimensional speedups of our FPGA-based Hestenes-Jacobi SVD compared to the Matlab SVD solution running on an Intel platform are presented, in which our Hestenes-Jacobi architecture shows better efficiency in analyzing matrices with small to medium column dimensions compared to the standard software solution, even when they are with comparably large row dimensions. The dimensional speedups that can be achieved range from $3.8\times$ to $43.6\times$ for matrices with column sizes from 128 to 256 and row

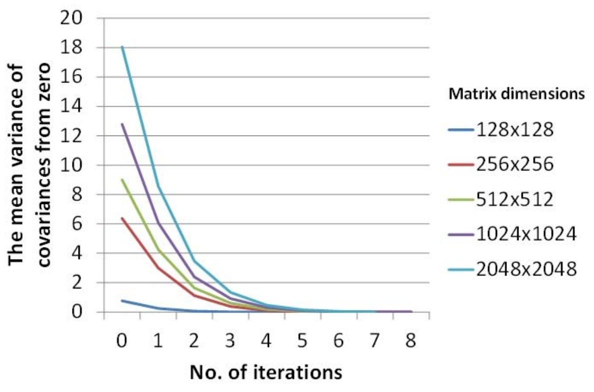

Fig. 10: Convergence process of different dimensional matrices.

dimensions from 128 to 2048. Table II shows the resource utilization by our Hestenes-Jacobi architecture.

Compared to the experimental results of the latest GPU-based and FPGA-based Hestenes-Jacobi implementations, our architecture shows the best performance [11], [12]. In [12], the GPU-based implementation, which ran 106.90ms and 1022.92ms to decompose a 128×128 and a 256×256 matrix respectively, failed to achieve any speedup compared to a conventional software solution. The FPGA-based design [11] was devised to perform fixed-point operations, which can only analyze the matrices with the size up to 32×128 due to the limitation of on-chip memory. Although the better performance has been demonstrated compared to their software execution with Matlab SVD for matrices with dimensions range from 2×2 to 32×127, our Matlab SVD routine runs 100 times faster than their Matlab SVD, and shares comparable speeds with their FPGA-based design. In further comparison to [11], in which 24.3143ms is needed to decompose the largest analyzed matrix with the dimensions of 32×127, the execution time of operating a 128×128 matrix by our architecture shows more than 5 times speedup.

TABLE I: Execution time in seconds.

$m\backslash n$	128	256	512	1024
128	4.39×10^{-3}	6.30×10^{-3}	1.01×10^{-2}	1.79×10^{-2}
256	2.52×10^{-2}	3.30×10^{-2}	4.84×10^{-2}	7.94×10^{-2}
512	1.70×10^{-1}	2.01×10^{-1}	2.63×10^{-1}	3.87×10^{-1}
1024	1.23	1.35	1.61	2.01

TABLE II: Resource consumption of our Hestenes-Jacobi architecture.

$Resource$	Slice LUT	$BRAM$	DSPs
$Our Architecture$	89%	91%	53%

C. Convergence Analysis

SVD is a process of diagonalizing matrix through iterative rotations; to evaluate the correctness and accuracy of the

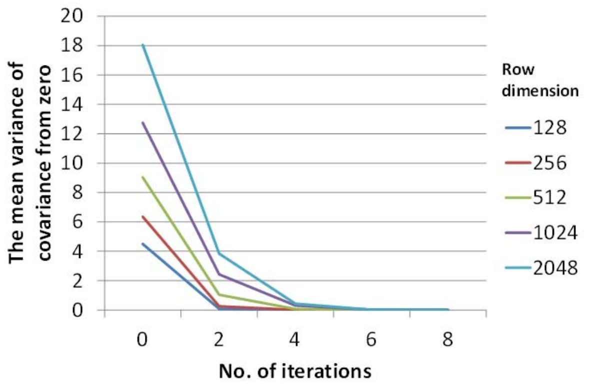

Fig. 11: Convergence process of matrices with column size of 1024 and various row sizes.

Hestenes-Jacobi produced singular values, the convergence speed needs to be analyzed. In our evaluation, randomly generated datasets have been applied to the implemented software model of our Hestenes-Jacobi design. The mean absolute deviations from zero of the covariances after being processed by a number of iterations are shown in Fig. 10, in which covariances between column vectors are rapidly converged to zero as the number of processing iterations increase. Reasonable convergence can be achieved within 6 iterations of operations for matrices of dimensions no greater than 2048. Also, similar observations can be obtained from the convergence performance evaluation of $m \times n$ matrices (shown in Fig. 11), in which the applied datasets are with identical column size of 1024 but various row dimensions.

VII. Conclusion and Future Work

An FPGA-based hardware architecture is proposed and implemented to perform Singular Value Decomposition with Hestenes-Jacobi approach; which is capable to analyze arbitrary $m \times n$ rectangular matrix with double-precision floating-point arithmetic. The performance analysis demonstrates the dimensional-dependent efficiency of our design compared to standard software solutions, and the better performance compared to other Hestenes-Jacobi implementations on GPUs and FPGAs. Also, convergence is evaluated by applying random generated datasets with various dimensions. Our proposed framework will be extended to perform principal component analysis for latent semantic indexing as the future work.

Acknowledgements

This work is supported in part by the National Science Foundation (NSF) under awards CNS-1116810 and CCF-1149539.

References

[1] H. Xu, C. Caramanis, and S. Sanghavi, "Robust PCA via outlier pursuit," *IEEE Transactions on Information Theory*, vol. 58, no. 5, pp. 3047–3064, April 2012.

[2] Y. Mu, J. Dong, X. Yuan, and S. Yan, "Accelerated low-rank visual recovery by random projection," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2011, pp. 2609–2616.

[3] R. Liao, Y. Fernandess, K. Tavernier, G. Taylor, and M. Irving, "Recognition of partial discharge patterns," in *Proceedings of IEEE Power and Energy Society General Meeting*, July 2012, pp. 1–8.

[4] E. J. Candes, X. Li, Y. Ma, and J. Wright, "Robust principal component analysis?" *The Computing Research Repository*, vol. 0912.3599, 2009.

[5] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[6] G. Golub and W. Kahan, "Calculating the singular values and pseudo-inverse of a matrix," *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, vol. 2, no. 2, pp. 205–224, 1965.

[7] S. Lahabar and P. Narayanan, "Singular Value Decomposition on GPU using CUDA," in *Proceedings of IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–10.

[8] A. Haidar, J. Kurzak, and P. Luszczek, "An improved parallel singular value algorithm and its implementation for multicore hardware," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*, New York, NY, USA, Nov. 2013, pp. 90:1–90:12.

[9] F. T. Brent, Richard P. Luk and C. V. Loan, "Computation of the Singular Value Decomposition using mesh-connected processors," *Journal of VLSI Computer Systems*, pp. 243–270, 1985.

[10] M. Hestenes, "Inversion of matrices by biorthogonalization and related results," *Journal of the Society for Industrial and Applied Mathematics*, vol. 6, no. 1, pp. 51–90, 1958.

[11] C. Kotas and J. Barhen, "Singular Value Decomposition utilizing parallel algorithms on graphical processors," in *Proceedings of OCEANS 2011*, Sept. 2011, pp. 1–7.

[12] L. Ledesma-Carrillo, E. Cabal-Yepez, R. de J Romero-Troncoso, A. Garcia-Perez, R. Osornio-Rios, and T. Carozzi, "Reconfigurable FPGA-Based unit for Singular Value Decomposition of large m x n matrices," in *Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Nov.-Dec. 2011, pp. 345–350.

[13] C. D. Martin and M. A. Porter, "The extraordinary SVD," *The American Mathimatical Monthly*, vol. 119, no. 10, pp. 838–852, 2012.

[14] T. F. Chan, "An improved algorithm for computing the Singular Value Decomposition," *Journal of ACM Transactions on Mathematical Software*, vol. 8, no. 1, pp. 72–83, 1982.

[15] Z. Drmac, "Implementation of Jacobi Rotations for accurate singular value computation in floating point arithmetic," *SIAM Journal on Scientific Computing*, vol. 18, no. 4, pp. 1200–1222, 1997.

[16] G. Golub and C. Reinsch, "Singular Value Decomposition and least squares solutions," *Numerische Mathematik*, vol. 14, no. 5, pp. 403–420, 1970.

[17] J. Demmel and W. Kahan, "Accurate singular values of bidiagonal matrices," *SIAM Journal on Science and Statistical Computing*, vol. 11, no. 5, pp. 873–912, Sep. 1990.

[18] M. Gu and S. C. Eisenstat, "A divide-and-conquer algorithm for the bidiagonal SVD," *SIAM Journal on Matrix Analysis and Applications*, vol. 16, no. 1, pp. 79–92, Jan. 1995.

[19] R. P. Brent and F. T. Luk, "A systolic architecture for the Singular Value Decomposition," Ithaca, NY, USA, Tech. Rep., 1982.

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

[21] W. Ma, M. E. Kaye, D. M. Luke, and R. Doraiswami, "An FPGA-Based Singular Value Decomposition processor," in *Proceedings of Canadian Conference on Electrical and Computer Engineering (CCECE)*, May 2006, pp. 1047–1050.

[22] P. Meher, J. Valls, T.-B. Juang, K. Sridharan, and K. Maharatna, "50 years of CORDIC: algorithms, architectures, and applications," *IEEE Transactions on Circuits and Systems I*, vol. 56, no. 9, pp. 1893–1907, 2009.

[23] "The convey hc-2 computer architecture overview." [Online]. Available: http://www.conveycomputer.com/

[24] "Logicore IP floating-point operator data sheet," March 2011. [Online]. Available: http://www.xilinx.com/