Adaptive Scalable SVD Unit for Fast Processing of Large LSE Problems

Iñaki Bildosola and Unai Martinez-Corral
Grupo de Diseño en Electrónica Digital (GDED)
University of the Basque Country (UPV/EHU)
Bilbao, Basque Country, Spain
inaki.bildosola@ehu.es and umartinez012@ikasle.ehu.es

Koldo Basterretxea
Dept. Electronics Technology
University of the Basque Country (UPV/EHU)
Bilbao, Basque Country, Spain
koldo.basterretxea@ehu.es

***Abstract*—Singular Value Decomposition (SVD) is a key linear algebraic operation in many scientific and engineering applications. In particular, many computational intelligence systems rely on machine learning methods involving high dimensionality datasets that have to be fast processed for real-time adaptability. In this paper we describe a practical FPGA (Field Programmable Gate Array) implementation of a SVD processor for accelerating the solution of large LSE problems. The design approach has been comprehensive, from the algorithmic refinement to the numerical analysis to the customization for an efficient hardware realization. The processing scheme rests on an adaptive vector rotation evaluator for error regularization that enhances convergence speed with no penalty on the solution accuracy. The proposed architecture, which follows a data transfer scheme, is scalable and based on the interconnection of simple rotations units, which allows for a trade-off between occupied area and processing acceleration in the final implementation. This permits the SVD processor to be implemented both on low-cost and high-end FPGAs, according to the final application requirements.**

***Index Terms*—Singular Value Decomposition, adaptive threshold, selectable accuracy, scalable architecture, FPGA**

I. Introduction

This work was originally motivated by the need for acceleration of large least square estimation (LSE) problem solving in embedded processors for computational intelligence applications. The learning or adaptation algorithms involved in such information processing-schemes generally give rise to linear systems defined by large-scale, rank-deficient, and ill-conditioned matrices. To numerically solve the LSE problems associated with such matrices various factorization algorithms can be applied, but the SVD is the most accurate and numerically robust, on balance. This is especially so in situations when the matrix may be rank-deficient or close to being rank-deficient as it allows us to determine the so-called pseudoinverse matrix, which generalizes for arbitrary matrices the notion of the inverse of a square, invertible matrix [1]. Due to these properties, the SVD is commonly used in the solution of unconstrained linear least square problems, matrix rank estimation, and canonical correlation analysis. In computational science, it is commonly applied in domains such as information retrieval, seismic reflection tomography, and real-time signal processing. As mentioned, it is also a key linear algebraic operation at the heart of many machine learning methods, which often involve on-line learning and/or massive dataset dimensionality and size [2]. However, when it comes to fast or real-time operation and when available computational resources are limited (embedded systems), the required processing time of SVD algorithms on regular computing architectures may be unacceptably long for many applications.

In order to fast process the SVD algorithm, the design of specifically tailored processing architectures implemented on FPGAs is an increasingly common approach in contemporary literature. FPGA-based processor implementation allows for the use of advanced design methods such as pipelining, parallelization, and HW/SW codesign to achieve higher processing performance for unit area and power consumption [3]. Several papers have been published where the SVD is implemented on an FPGA for the factorization of small-scale matrices. In this sense, most of them are focused on Brent, Luk and Van Loan's idea of an expandable square systolic array of simple 2×2 processors, based on the two-sided Jacobi algorithm [4]. An interesting approach described in [5] is based on the implementation of a customized architecture, not based on a systolic array, which is validated for medium-scale square matrix sizes up to 150×150.

Different methods have been proposed to improve parallelization in order to process larger matrices or achieve higher processing speed. In [5] the theoretical development of a stream algorithm for the SVD of non-square $m \times n$ matrices, based on a square array of processors, is described. Making use of the former theoretical approach, in [6] the one-sided Jacobi algorithm is implemented in a slightly modified form (named JRS) and tested with large $n \times n$ square matrices up to 1000×1000. However the convergence and the architecture are not explained thoroughly. An FPGA-based SVD unit prototype implementation for large, non-square $m \times n$ matrix decomposition is presented in [3], although tests on matrices of up to only 32×127 are performed. In this work, the convergence accuracy criterion and the solution error linked to it is firstly considered and set as an important user-defined parameter. Lately, high-level synthesis flows have been proposed for the automatic

This work was supported by the Spanish Ministry of Economy and Competitiveness, and European FEDER funds (grant TEC2010-15388), and by the Basque Government (grants IT733-13, S-PC12UN016, and S-PC13UN034).

implementation of 'cone-based propagation' architectures to process iterative stencil loop algorithms [7]. In brief, since many published architectures attempt to theoretically reach full-parallelization they are based on regular fixed schemes, usually only small-scale implementation examples are described, and no practical optimized performance designs have been published considering large matrices.

In this work, we present a scalable SVD processing unit architecture specifically tailored to the factorization of large matrices, but which can be adapted to process square and non-square matrices of any size. It can be implemented on both small and large FPGAs for the best cost/speed trade-off in each target application. The selected method to perform the SVD has been firstly optimized by introducing some modifications to the well-known one-sided Jacobi algorithm with the aim of speeding up the factorization process by reducing the total amount of required rotations for a given desired accuracy. In particular, an adaptive threshold-based decision unit is proposed, which individually optimizes the convergence threshold (also named rotation threshold) of performed rotations. Moreover, some internal arithmetic operations have been carefully designed to achieve a computing scheme that makes the most of available HW resources in FPGAs. Finally, a scalable parallel processing architecture has been developed based on a linear array of processing-units (PUs) and a double data-flow paradigm (FIFO memories and a shared bus) for efficient data transfer. The performance of the system has been verified through several study cases and matrix decompositions.

The paper is organized as follows: in Section II a brief description of different alternatives and parallelization degrees in the SVD algorithm is given. In Section III, the influence of matrix conditioning, together with the available computing precision and obtainable accuracy in the solution are discussed. Here the concept of convergence threshold is presented as a user defined parameter, which allows us to obtain a trade-off between desired accuracy and computing effort. In Section IV, the concept of Adaptive-Threshold is introduced and its application to the enhancement of the SVD computation is exposed. Several results are presented supporting the achieved improvements in the algorithm's behaviour. In Section V, the proposed architecture and the nuclear PU design are described, along with both theoretical performance and practical implementation results.

II. SVD ALGORITHM AND PARALLELIZATION

The Singular Value Decomposition (SVD) of an $m \times n$ matrix A is defined by

$$A = U\Sigma V^T \tag{1}$$

where Σ is a diagonal matrix such that $\Sigma = diag(\sigma_1, \sigma_2, \ldots, \sigma_n)$ where $(\sigma_1, \sigma_2, \ldots, \sigma_n)$ are the singular values of A. The U and V matrices are orthogonal matrices of size $m \times m$ and $n \times n$ respectively. When the matrix is not invertible, a pseudo-inverse matrix A^+ such that $(A^+A)^T = A^+A$ (also known as the Moore-Penrose matrix) may be determined by

$$A^+ = V\Sigma^+U^T \tag{2}$$

where Σ^+ is the inverse of Σ, i.e. it is a diagonal matrix formed by inverted (when non-zero) values of $\sigma_1, \sigma_2, \ldots, \sigma_n$. Thus, when it comes to solving the LSE problem $Ax = b$, the solution will be obtained by

$$x' = A^+b \tag{3}$$

Among existing SVD algorithms, the Jacobi algorithm in its two main variants, two-sided and one sided, provides the best opportunities for parallelization. The original two-sided algorithm exploits (2) to generate the matrices U and V by performing a sequence of orthogonal two-sided plane rotations to the input matrix $J_i^l A_i J_i^r = A_{i+1}$. Jacobi rotations affect only two columns/rows i and j of the matrix A, thus allowing a parallel computation scheme. However, this parallelization, considering data dependence, is not purely non-conflicting. Briefly stated, a parallel processing attempt would suppose that at least two processors would be tackling a different pair of columns/rows, which means both trying to update some common elements at the same time. Even though the convergence of the algorithm can be proved [5], it is not the best option from a hardware implementation point of view, since each processor would have to manage two rows and two columns; this is a complex issue when it comes to processing large matrices.

The one-sided Jacobi or Hestenes-Jacobi algorithm, provides a satisfying solution to the above-mentioned problems. Since it is based on the orthogonalization of pairs of columns, data sharing is purely non-conflicting [8]. Given an $m \times n$ non-symmetric matrix, the Hestenes-Jacobi method generates an orthogonal matrix W as a product of plane rotations $AV = W$. Thus, the Euclidean norms of the columns of W are the singular values of A:

$$\sigma_i = ||W_{(:,i)}|| \tag{4}$$

By normalizing W by its Euclidean length $||W_{(:,i)}||$ the U matrix is obtained:

$$U_{(:,i)} = \frac{W_{(:,i)}}{\sigma_i} \tag{5}$$

The SVD of the matrix is then obtained by $AV = W \rightarrow AV = U\Sigma \rightarrow A = U\Sigma V^T$, as defined in (1). Plane rotations represented by $A^{k+1} = Q^k A^k$, affect only the column-pairs $(A^k_{(:,i)}, A^k_{(:,j)})$. In each rotation the angle θ^k_{ij} is chosen in such a way that the new column-pairs are orthogonal. This is done by applying (6) (the formula of Rutishauser [9]), and performing Givens' rotations as defined in (7).

$$tan(2\theta^k_{ij}) = \frac{2 \cdot (A^k_{(:,i)} * A^k_{(:,j)})}{||A^k_{(:,j)}||^2 - ||A^k_{(:,i)}||^2} \tag{6}$$

$$\begin{aligned} A^{k+1}_{(:,i)} &= A^k_{(:,i)} \cdot cos(\theta^k_{ij}) - A^k_{(:,j)} \cdot sin(\theta^k_{ij}) \\ A^{k+1}_{(:,j)} &= A^k_{(:,i)} \cdot sin(\theta^k_{ij}) - A^k_{(:,j)} \cdot cos(\theta^k_{ij}) \end{aligned} \tag{7}$$

Note that exactly the same rotation is performed with the V matrix.

Based on the Hestenes-Jacobi method, we have implemented a parallelizable algorithm that receives as input arguments the matrix A and a threshold value, which defines when two columns are to be considered as orthogonalized. This is an essential parameter for evaluating the convergence of the algorithm that is of particular relevance when computing with finite precision processors, as we will see in the next section. When convergence is verified, the algorithm returns the three matrices as the result of the factorization.

III. Matrix conditioning, computing precision, and obtainable accuracy: the threshold value

The parameterization of the modified Hestenes-Jacobi algorithm presented here by setting the value of the rotation threshold will provide the SVD unit with a certain measure of control over the computing effort to reach a desired level of accuracy in the solution. However, when dealing with numerical linear algebraic problems, the condition number of the matrix, $\kappa(A)$, plays a central role in the analysis of the obtainable solution accuracy. At the same time, since finite-precision processors are used to process the SVD, the effects of round-off and its derived computing errors will intimately interact with the aforementioned convergence parameters and $\kappa(A)$. For instance, there is no point in setting a very demanding rotation threshold value when computing with very low precision. In the same manner, we cannot expect a very accurate solution for an ill-conditioned matrix, even if the algorithm is computed with high precision. In consequence, it is worth analyzing these numerical issues to have a picture of what can be expected in terms of solution accuracy before designing a SVD processing unit.

A. Matrix Conditioning and Computing Precision

The solution accuracy of an overdetermined system $Ax = b$ is given by the residual vector $r = ||Ax' - b||$, with x' being the least linear square method solution which minimizes this vector. Thus, as it is well known, $\kappa(A)$, which defines how sensitive the problem is to perturbations in the data (A and b), will directly influence the achievable final solution accuracy. As a rule of thumb, with $\kappa(A) \approx 10^k$, the computed solution of $Ax = b$ should usually be at least as accurate as $q - k$ significant digits, where A and b are accurate to q significant digits. Apart from general theoretical studies, some papers have analyzed its impact when solving least linear square problems by iterative methods [10]. In addition, there is also a limit imposed by the available computing precision due to the truncation caused by the limited word length or round-off error. Finally, linked with both the $\kappa(A)$ and computing precision, the matrix size (ultimately the number of unknowns), is also a limiting factor of the achievable solution accuracy [11] since computing inaccuracies may aggravate as a consequence of the cumulative propagated error on iterative algorithms.

Some experimental numerical results are graphically given below in Figs. 1 and 2, in order to expose how the Hestenes-Jacobi algorithm in particular is affected by the combination of the $\kappa(A)$, the computing precision, and the matrix size. The matrices were randomly generated in Matlab with a fixed value of the $\kappa(A)$ (*sprand* function) and with elements normalized in the range $[-1, 1]$. The results obtained with the Matlab *svd* function computed with double-precision floating point accuracy were taken as 'true' or 'exact' solutions. For comparison purposes, a single-precision version of the Hestenes-Jacobi algorithm was programmed and applied to the same example matrices. This testing is particularly meaningful to us since we have been implementing the SVD algorithm by software in several previous SoC designs using embedded single-precision soft-processors in FPGAs (Xilinx Microblaze32-bit RISC Harvard). Three meaningful errors are specified: the Singular Error (SE), as the maximum normalized error in the computed non-null singular values; the Inverse Error (IE), measured as the normalized l-2 norm of the error in the elements of the pseudoinverse matrix relative to the double precision solution; and the Remainder (RE), as the l-2 norm of the difference between A and USV^T. The reflected values were obtained as the normalized mean errors for 20 randomly generated matrices.

From Fig. 1, it can be appreciated that the SE will be close

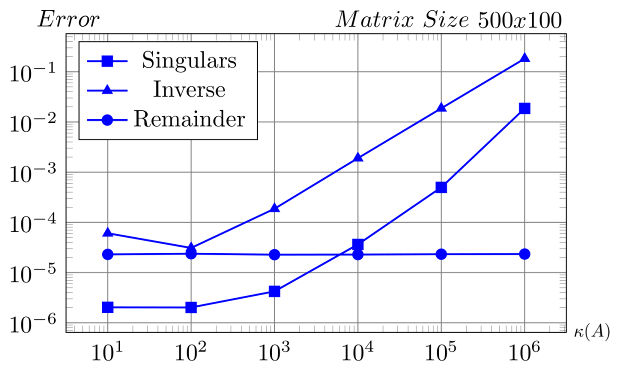

Fig. 1. Singular Error, Inverse Error and Remainder Error for increasing condition number in 500×100 matrices.

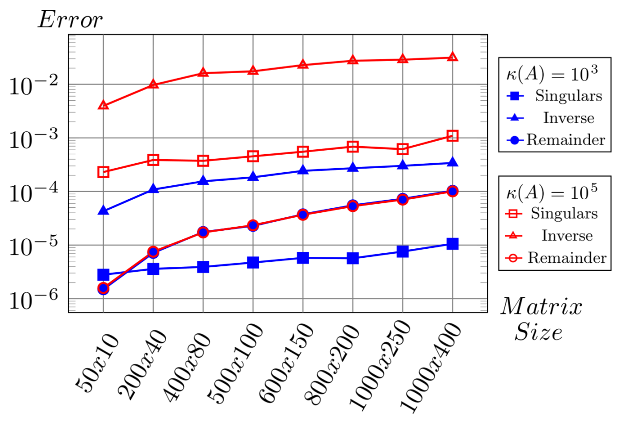

Fig. 2. Singular Error, Inverse Error and Remainder Error for increasing matrix sizes and $\kappa(A) = (1E3, 1E5)$

to the computing precision (10^{-7}) for small values of $\kappa(A)$, while it increases one order of magnitude when the $\kappa(A)$ increases as well. A similar behavior is inferred for the IE, but always 10^2 worse, as it is a more sensitive error. As said before, if A and b are accurate to q significant bits, assuming this accuracy is better than the IE and taking into account expression (3), the error in x' should not differ too much from those error magnitudes. Fig. 2 shows the matrix size impact on the computed errors for two different values of $\kappa(A)$. As it can be seen, the larger the $\kappa(A)$ the more noticeable the error degradation.

B. The rotation threshold

Once the potential accuracy of the solution has been estimated for a given problem and a given computation precision, there is a key parameter in the SVD algorithm which will determine how accurate the solution is actually going to be and, at the same time, what the required total computing effort (processing time) will be in order to obtain that accuracy: this is the (rotation) threshold value. As an iterative algorithm, the one-sided Jacobi needs a threshold value to decide when the orthogonalization process is finished, and this value must be set to find a compromise between the desired accuracy and the processing time. In this sense, Fig. 3 shows the impact of the threshold value on the factorization error and on the computing effort for the programmed algorithm (single-precision).

As expected, a more demanding threshold value produces a higher level of accuracy (a lower error) and calls for more rotations (more processing time). Notwithstanding, an error saturation phenomenon can be observed since no appreciable improvement in the accuracy is obtained beyond a threshold value determined by the already analyzed factors, even when the number of rotations continues increasing. This phenomenon can be described as an 'error saturation limit' and it depends on the factors analyzed in Section III-A. Based on these results, and before setting the threshold value of the SVD algorithm for a target application, it would be sensible to estimate this saturation limit based on the available information on the expected values of $\kappa(A)$, the matrix sizes, and the available computing precision. Once that limit is known, more relaxed threshold values could be used to achieve shorter processing times for those target applications that require fast computation, should a lower accuracy be acceptable.

IV. Enhanced SVD Algorithm for Optimized Convergence and Hardware Implementation

A. The adaptive threshold

The rotation threshold in the Hestenes-Jacobi algorithm is not an absolute value. In fact, it is related to the Euclidean norm values of the processed columns. As the algorithm is based on the orthogonalization of pairs of columns, i.e. rotating them until their scalar product falls below a predetermined limit value (rotation threshold), our proposal is to adapt this value to the norm of the columns to be processed at each time. That is to say, use more demanding (smaller) values with columns of smaller norms, and use looser values with large

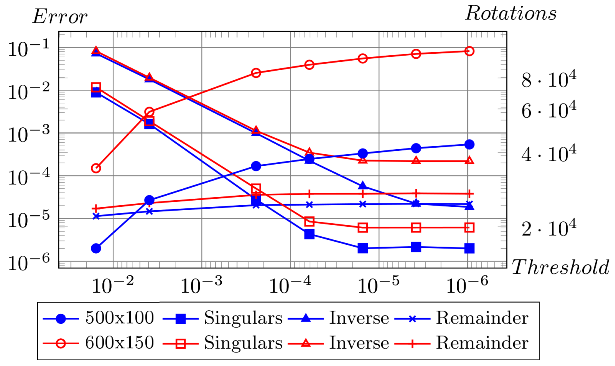

Fig. 3. Number of total rotations until convergence (dots) and obtained IE and SE for decreasing values of the rotation threshold. Two examples: 500×100 matrix $(\kappa(A) = 1E2)$ and 600×150 matrix $(\kappa(A) = 1E3)$.

norm columns. In this way all the columns will be uniformly orthogonalized and no rotations will be performed in vain to achieve a desired accuracy. In fact, since the norms of the columns of the W matrix are the singular values themselves (4), to be fussier with them means to be fussier with smaller singular values. Ultimately, the scalar product to be used has to be normalized. This approach was already used by Brent and Luk (B&L) in [4], where the threshold value is compared with the normalized scalar product:

$$\frac{A_i \times A_j^T}{||A_i|| \cdot ||A_j||} < Threshold \tag{8}$$

The result of performing this evaluation before each rotation is that similar relative error values for all eigenvalues, large and small, are obtained. In addition to that, focusing on achieving a similar error range, fewer rotations are performed as larger columns are not rotated in vain, so the total number of rotations is reduced with no increase in the error magnitude. We propose going a little further than B&L in giving more relevance to small norm columns from earlier in the algorithm execution by remultiplying the threshold value again by the smallest norm of the columns. So this new evaluation method, which we named Adaptive Minimum Norm (AMN), is:

$$\frac{A_i \times A_j^T}{||A_i|| \cdot ||A_j||} < Threshold \cdot min(||A_i||, ||A_j||) \tag{9}$$

The left side in (9) can be generalized as $cos(\alpha_{ij}) = f(A_i * A_j^T, ||A_i||, ||A_j||)$, since $A_i * A_j^T = ||A_i|| \cdot ||A_j|| \cdot cos(\alpha_{ij})$. The Rutishauser angle, which has to be computed if a rotation is required, may also be expressed as $\theta_{ij} = g(A_i * A_j^T, ||A_i||, ||A_j||)$, according to (6). With the aim of optimizing the algorithm execution by avoiding the computation of two different angles which basically contain the same implicit information, one step more is given in our approach. The rotation decision (9) is readapted, bypassing the computation of the scalar product and directly comparing the adaptive threshold value with the Rutishauser angle to make the rotation decision. The new evaluation expression (Adaptive Rutishauser or ARH) is now:

$$\theta_{ij} < Threshold \cdot min(||A_i||^2, ||A_j||^2) \quad (10)$$

In this case, since the Rutishauser angle is more sensitive to the norm's change, instead of remultiplying the threshold by the smallest norm of both columns, the square of this value is used in order to maintain the level of regularization. Moreover, the calculation of the square roots is avoided, since $||A_k||^2 = \sum_1^m (A_k * A_k)$ can be computed in a multiply-accumulate (MACC) unit. Focusing on hardware realization, from (9) to (10) two multiplications and two square roots are saved. Therefore, if powers of two are used in the selected threshold value, evaluating (10) is reduced to a shift operation (apart from the two comparators and a multiplexor, which are needed in any case).

B. The sequence and sorting

The minimum-length 'sweep', denoting any sequence to go through all matrix column-pairs at least once, seems to be a fairly simple combinatorial problem that produces $n\cdot(n-1)/2$ combinations. However, when parallel-computing the SVD, going through all column-pairs is not enough to ensure convergence, and sorting, due to implicit information transference, has to be considered. Minimum convergence time is achieved when (i, j) pairs are computed before any $(i + k_i, j + k_j)$, where $k_i, k_j \in \mathbb{N}$, as in serial cyclic sequences. Column-pair sequence generation and its effects on the algorithm convergence have been discussed in depth in several papers [5], [12], but generally neither the computation of matrix V nor conditional data transfer issues are considered when analysing computation time and resource requirements. Since in our approach the effective computation-time is reduced through fast hardware processing, the relative impact of data transfer times becomes critical.

As described in [12], index ordering is not sufficient to guarantee minimum convergence time, and sorting has to be considered for efficient implementation. When using cyclic sequences ($i < j \ \forall \ i, j$) we can imagine convergence as the information structure contained in data virtually redistributing, placing the most of it in the first columns and less as we approach the last ones. Since $||A_i||$ is incremented and $||A_j||$ is reduced in each rotation, convergence can be accelerated if we swap columns before orthogonalization when $||A_i|| < ||A_j||$ [12]. Then, if we include this 'Active Sorting' concept, expressions (9) and (10) are simplified to:

$$\frac{A_i \times A_j^T}{||A_i|| \cdot ||A_j||} < Threshold \cdot ||A_j|| \quad (11)$$

and

$$\theta_{ij} < Threshold \cdot ||A_j||^2 \quad (12)$$

In reality, computing (12) requires the same hardware resources as evaluating (10), since the minimum square norm has to be checked in any case. However, sorting as soon as data is available allows us to compute an angle θ_{ij} which will always reduce the norm of smaller columns, thus allowing for less aggressive threshold values. As a result of the larger norm always being $||A_i||$, and since $i < j$, besides accelerating convergence, the computed singular values will be sorted in a mainly decreasing order as soon as the first sweep is completed. The columns with a null norm will always be last and, since these do not contribute to enhancing the accuracy of the final solution, we can take advantage of the sorting to ignore them in the following sweeps (see dotted boxes in Fig. 4).

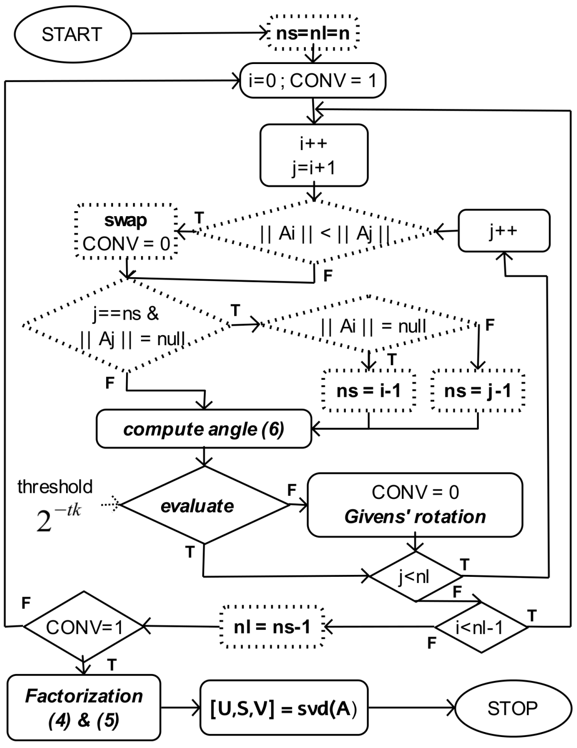

Fig. 4. SVD Active Adaptive Threshold algorithm for $m \times n$ matrices.

The proposed enhanced SVD algorithm including the Active Adaptive Rutishauser (AARH) threshold concept (12) is shown in Fig. 4. When a sweep is completed and the convergence flag is *true*, the rotation phase is finished. Regarding the factorization, matrix V is directly obtained and matrices U and Σ will be obtained by (5) and (4).

C. Comparative tests

Several tests have been made to compare the performance of the above-described rotation evaluation strategies and verify whether AARH is able to actually produce a smaller number of rotations without loss of accuracy. Illustrative results are summed up in Table I where the AARH strategy (12) is compared to other alternatives: the fixed threshold [3], the B&L evaluation (8) [4], the Active Sorting approach applied to B&L (ABL), and the Active Adaptive Minimum Norm (AAMN) (11). It has to be taken into account that reaching the saturation error through each approach requires establishing different threshold values. However this is the fairest way to make a comparison since the processing effort to obtain a given solution accuracy is measured. The matrix size is

TABLE I
COMPARATIVE OF ALGORITHM PERFORMANCE

	Inverse Error | $2^{-Threshold}$			
Fixed	4.19E-6 | 24	2.11E-5 | 24	2.54E-4 | 24	1.89E-3 | 12
B&L	3.98E-6 | 22	1.93E-5 | 18	1.62E-4 | 16	1.87E-5 | 16
ABL	3.68E-6 | 22	1.83E-5 | 18	1.63E-4 | 16	1.78E-3 | 12
AAMN	5.60E-6 | 20	1.76E-5 | 16	1.68E-4 | 10	1.34E-3 | 8
AARH	6.77E-6 | 20	1.87E-5 | 16	1.76E-4 | 10	1.41E-3 | 8
$\kappa(A)$	1E1	1E2	1E3	1E4
Fixed	33,696 | 11.3	33,530 | 10.50	33,905 | 10.50	34,004 | 10.75
B&L	32,303 | 9.70	30,911 | 10.20	29,858 | 10.15	26,482 | 10.13
ABL	26,096 | 8.20	24,356 | 8.00	23,121 | 8.05	19,927 | 7.73
AAMN	25,275 | 8.13	23,209 | 8.05	18,353 | 8.05	15,327 | 8.30
AARH	25,512 | 8.40	23,345 | 8.50	18,960 | 8.53	16,078 | 8.35
	Rotations | Sweeps			

TABLE II
TEST MATRICES RESULTS
Inverse Error | $2^{-Threshold}$; Rotations | Sweeps

Size | κ	201 × 47 | 7.5E1	235 × 216 | 1.7E3	200 × 200 | 2.4E3
Name	JGD_Kocay/Trec9	JGD_Forest/TF11	Bai/bwm200
ABL	1.99E-6 | 22	1.4E-3 | 12	2.96E-4 | 14
	5,004 | 7	97,281 | 9	95,902 | 9
AAMN	1.96E-6 | 18	5.3E-4 | 6	1.43E-4 | 8
	4,903 | 8	75,433 | 9	61,981 | 9
AARH	2.21E-6 | 16	8.03E-4 | 4	5.64E-4 | 6
	4,908 | 9	76,854 | 9	57,675 | 10

500×100 and the range of $\kappa(A)$ is that for which the error level is acceptable $(IE \leq 1\%)$ for single precision computing.

These results show how the error saturation is reached quite evenly with all approaches, but the required sweeps and the total number of rotations are significantly fewer for the adaptive strategies, as less restrictive threshold values are required. Indeed, the proposed new approaches incorporating optimized evaluation expressions demand generally fewer rotations and evenly fewer sweeps than the ABL, particularly for large condition numbers. Some additional tests have been performed with matrices that arise in real applications [13]. Accuracy figures and rotations savings (see some examples in Table II) are in accordance with the experimentation performed with randomly generated matrices.

As explained above, AARH has been finally selected as the best implementation option since multiplications and square roots are avoided and silicon area and computation time will be saved with no loss of accuracy.

V. System architecture

The proposed architecture for the parallelization of the enhanced Hestenes-Jacobi algorithm is based on the interconnection of various basic processing units or PUs. Each PU performs column-pair evaluations according to (6) and (12), and rotates A and V column-pairs as in (7) when orthogonalization is required. Regularity and size of the architecture, which are critical due to memory and routing requirements, rely on the nuclear PU design, since performance is directly related to the number of implemented PUs ($\#PU$). Scalability of the architecture is aimed both at being able to fit minimum size SVD units in low-cost FPGA devices and at pushing performance to the limits in high-end chips with many available resources.

Regarding memory requirements when managing large matrices, in the case that the distributed resources (LUTs) in the target FPGA are not enough to hold the complete set of data during matrix processing, embedded memory blocks or external memory banks are necessary, and data must be transferred sequentially. The main system memory, which holds matrices A and V during computation, is a notorious bottleneck, especially when off-chip modules are used [14]. Since one-sided Jacobi matches up better with systems with small random access memories [5], in order to reduce the impact of low main memory bandwidth and/or high access-time, fine-grained linear array implementations show the best trade-off between scalability and computing-time [15], [16]. Moreover, due to the adaptability of the AARH algorithm, our design has been conceived with the aim of moving data only when required. In other words, columns of V are transferred to/from each PU only when a rotation is to be performed and columns of A are saved back only if rotated and/or swapped.

A. Work scheduling

As mentioned above, cyclic sequences guarantee minimum convergence times of the Hestenes-Jacobi algorithm [5], [12]. Beyond that, it is the most suitable ordering strategy when seeking minimum data transfers: considering the time needed to process a column-pair (i, j) as a unitary step (T_{step}), a column i may be kept in a PU while computing $n - i$ steps, that is from $j = i + 1$ to $j = n - 1$ (see Fig. 5, left). At the same time, cyclic scheduling is well suited to variable PU implementations (scalability) because each PU can handle a

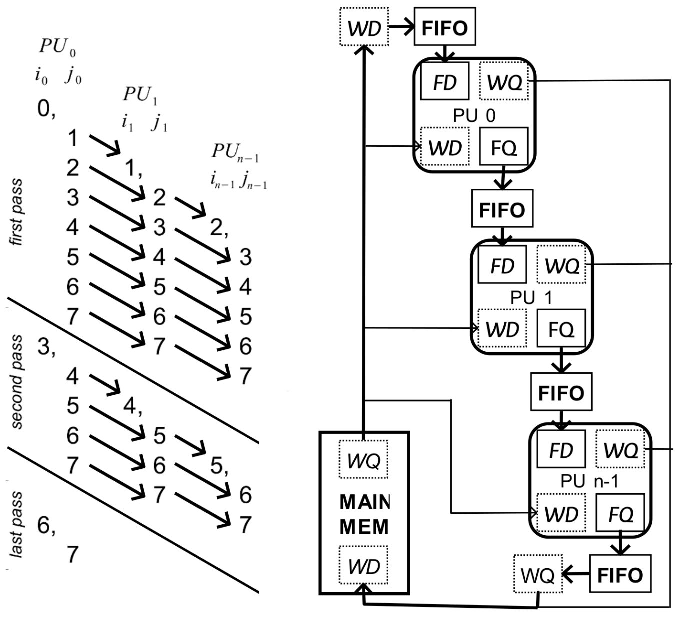

Fig. 5. Proposed processing architecture for $\#PU = 3$ (right) and column-pair schedule for $n = 8$ (left).

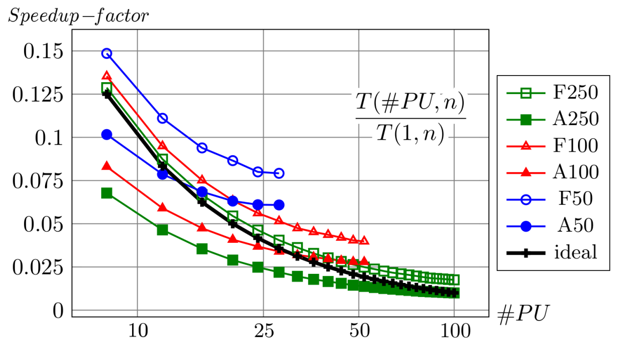

Fig. 6. Obtained speedup-factors in a sweep with fixed (F) and adaptive (A) step-times, compared to an ideal scheduling for $n = 50, 100, 250$.

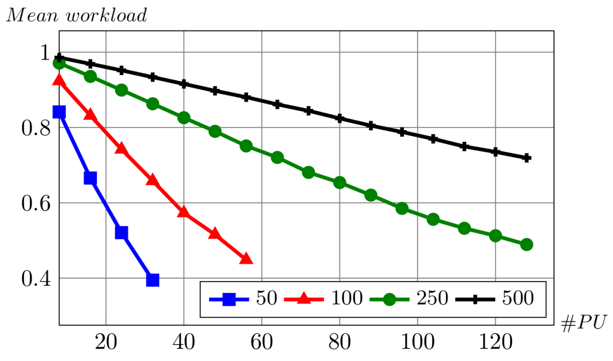

Fig. 7. Average workload of the PUs for a fixed step-time according to the implemented number of PUs ($\#PU$) for different matrix sizes.

whole column-block in a pass $[i+1, n-1]$, by itself, regardless of the available number of PUs [12].

In a minimal scheme with a unique dual-port main memory bank (see Fig. 5, right) the main data-flow goes from the reading port (WQ), through the PU-FIFO array and back to the writing port (WD). In order to move data only when necessary, without crossing the whole array of PUs, a secondary data-flow is adopted: a crossbar-switch that connects all the PUs to the main memory, providing asynchronous full-duplex communication (not explicitly shown, for the sake of simplicity). Columns i of the V matrix are transferred to each PU just once, if needed, for each pass (see Fig. 5, left) while columns V_j are pulled and pushed every time it is necessary, which is less and less frequent as sweeps go on. To further reduce access conflicts to the main memory, A_i columns are only sent back if rotated and/or swapped.

In a single PU architecture, the minimum computation time to complete a sweep can be defined as $T_1 = T_{step} \cdot n \cdot (n-1)/2$. The ideal full-parallel computation time would be $T_{PU} = T_1/\#PU$, assuming a fixed T_{step}, full-bandwidth (no memory conflicts), full-parallelization ($2\times\#PU$ columns are computed in T_{step}) and that the same number of total steps is needed for convergence. In our design, however, the last PUs in the array have to wait for a certain delay time or offset until the previous ones have processed their respective columns (see arrows in Fig. 5, left). Hence, assuming a fixed T_{step} and full-bandwidth, at the beginning of every pass, an offset, whose value depends on its position in the array of processors, is introduced in each PU. Although full parallelization is achieved just $2 \cdot \#PU - 1$ steps after loading the first column in a pass, when reaching $i = n-(\#PU-1)$ the parallelization level begins to fall as the remaining possible column-pair combinations get drastically reduced.

Nevertheless this lag in completing the sweeps becomes negligible as the matrix size increases, since increasing the values m or n improves the speedup-factor $T(\#PU, n)/T(1, n)$, asymptotically reaching $1/\#PU$ (see F-datasets in Fig. 6). Moreover, when threshold adaptability is implemented (AARH), the achieved speedup-factor outperforms the ideal full-parallel scheme reference (see A- datasets). Indeed, less data transfer requirements reduce true final computing time and relax routing requirements. Of course, speedup gets saturated at $\#PU = n/2$ since there are only n columns available. However, as a result of the offsets introduced, performance and the average workload of all of the PUs decrease as $\#PU$ increases (see Figs. 6 and 7). As a rule of thumb, the adaptive threshold approach equals the ideal performance when $\#PU = n/4$ with at least a 50% average workload.

B. The Processing Unit

Some arithmetic operations have been carefully chosen to achieve a hardware-friendly scheme in the design of the PUs. Fixed-point arithmetic has been used and all the fixed shifts have been hardwired to avoid barrel-shifters. Embedded DSP blocks are directly used in MACC mode to compute the square Euclidean norms and vector product $(||A_i||^2, ||A_j||^2, A_i * A_j^T)$ (see Fig. 8, top). As in [17] CORDIC is used in vectoring mode to compute the angle of Rutishauser (6) and then it is directly compared to the threshold value (12) (see Fig. 8, left). Threshold values meet 2^{-tk}, thus multiplications feeding the decision comparator are also hardwired shifts with no hardware cost. If orthogonalization is required, the same CORDIC core is used in rotating mode to perform Givens' rotations. Thus all the computations are reduced to add/subs or fixed shifts, except for the computation of the square Euclidean norms and vector products.

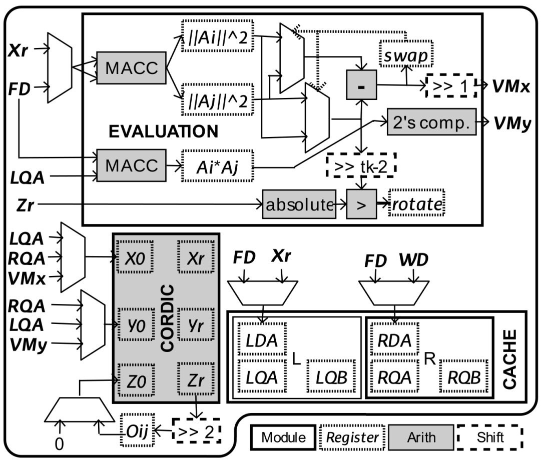

Fig. 8. Processing Unit (PU) core design.

Regarding latency, after loading A_i in m clock cycles, a column-pair is evaluated every $m+k_{MACC}+k_{CORDIC}$ cycles until $j = n-1$, where k_{MACC} is the latency of the DSP modules and k_{CORDIC} depends on the desired precision in (12). Then A_i is sent to the main memory through the bus and the following pair is loaded from the previous FIFO. If orthogonalization is required, high throughput is achieved with an ad-hoc optimized unrolled-pipelined realization. Since the logic blocks available in most FPGAs have registered outputs, pipelining has nearly no hardware cost. This allows rotating A_i,A_j, updating $||A_i||^2$, and sending A_{jr} to the following FIFO in $m + k_{CORDIC} + k_m + k_{MACC}$ cycles, where k_m is the latency for module compensation. Only $k_{CORDIC} + k_m$ additional cycles are required to rotate V_i, V_j, since loading and sending are completed while computing A.

C. Implementation and Tests

The proposed architecture has been implemented in both Spartan-6 and Kintex-7 devices by Xilinx. Resource utilization and system specifications for the orthogonalization, i.e. to obtain W and V, are exposed in Table III. Processing times have been measured considering $A_{m\times n}$ and $I_{n\times n}$ are already loaded into the main memory.

VI. Conclusion and future work

Two new adaptive rotation threshold approaches to the one-sided Jacobi algorithm have been proposed, AAMN and AARH, which outperform previous proposals in terms of required sweeps/rotations to achieve a desired solution accuracy. A parallel processing scheme based on a linear array of processing units and a double data-flow paradigm has been developed. The designed architecture is fully scalable between 2 PUs, for minimum hardware requirements, and $m/2$ PUs, for maximum performance. The adaptive data-flow optimizes data transfers by avoiding unnecesary transmisions and computations. Moreover, some internal arithmetic operations have been carefully chosen to better map into commonly available resources in FPGAs, i.e. DSPs cores and BRAMs, and to save computing time by eliminating square root and successive multiplication operations, as a consequence of using the Rutishauser angle directly in both, evaluation (6) and rotation (7) expressions.

Regarding future work, we are considering to take advantage of the implicit sorting in the AARH algorithm to perform the processing of bigger $\kappa(A)$ matrices with no need of higher computing precision. This could be achieved by discarding the columns below a previously set minimum norm value so the resulting adaptive threshold value would change in a smoother way.

Although both, the latency of evaluations/rotations in each PU and the offsets introduced at the beginning of each pass, get negligible as the matrix size is increased, there is room for improvement. We are studying optimal sequences and sorting alternatives to achieve smaller computation latencies with a minor impact on occupied area and algorithm convergence rate. Redundant arithmetic-based CORDIC designs, the use of square root and division free Givens' rotations, and designing ad-hoc estimators to compute Euclidean norms and the *atan* function are some of the means for upgrading we would like to explore. Performance of the double data-flow scheme may also be improved by implementing a multichannel shared bus along with a main memory divided into several modules.

TABLE III
Spartan-6 and Virtex-7 resource utilization and elapsed time

Model	xc6slx45-3fgg484	xc7k160t-3fbg484
DSPs | RAMs | Area (%)	96 | 86 | 30	33 | 80 | 38
Max. Freq.	$53.232MHz$	$151.236MHz$
Matrix Size | $\#PU$	200×80 | 4	800×200 | 25
Processing time	$74.1ms$	$77.18ms$

References

[1] G. Allaire and S. M. Kaber, *Numerical Linear Algebra*. Springer, 2008.

[2] M. Russo, "Genetic Fuzzy Learning," in *IEEE Transactions on Evolutionary Computation*, vol. 4, no. 3, 2000.

[3] L. M. Ledesma-Carrillo, E. Cabal-Yepez, R. d. J. Romero-Troncoso, A. Garcia-Perez, R. A. Osornio-Rios, and T. D. Carozzi, "Reconfigurable FPGA-Based Unit for Singular Value Decomposition of Large m x n Matrices," in *Reconfigurable Computing and FPGAs (ReConFig), 2011 International Conference on*, 2011, pp. 345–350.

[4] R. P. Brent, F. T. Luk, and C. Van Loan, "Computation of the Singular Value Decomposition Using Mesh-Connected Processors," *Journal of VLSI and Computer Systems*, vol. 1, no. 3, pp. 242–270, 1985.

[5] V. Strumpen, H. Hoffmann, and A. Agarwal, "A Stream Algorithm for the SVD," Computer Science and Artificial Intelligence Laboratory, MIT, Tech. Rep. Technical Memo 641, 2003, computer Science and Artificial Intelligence Laboratory, MIT.

[6] S. Rajasekaran and M. Song, *A Novel Scheme for the Parallel Computation of SVDs*, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 4208, pp. 129–137.

[7] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza, "A High-Level Synthesis Flow for the Implementation of Iterative Stencil Loop Algorithms on FPGA Devices," in *Proc. of DAC*, 2013.

[8] M. R. Hestenes, "Inversion of Matrices by Biorthogonalization and Related Results," *J. Soc. Ind. Appl. Math*, vol. 6, no. 1, pp. 51–90, 1958.

[9] H. Rutishauser, "The jacobi method for real symmetric matrices," *Numerische Mathematik*, vol. 9, no. 1, pp. 1–10, 1966.

[10] M. Arioli and F. Romani, "Relations Between Condition Numbers and the Convergence of the Jacobi Method for Real Positive Definite Matrices," *Numerische Mathematik*, vol. 46, pp. 31–42, 1985.

[11] A. Pyzara, B. Bylina, and J. Bylina, "The influence of a matrix condition number on iterative methods' convergence," in *Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2011, pp. 459–464.

[12] B. B. Zhou and R. P. Brent, "A parallel ring ordering algorithm for efficient one-sided jacobi svd computations," *Journal of Parallel and Distributed Computing*, vol. 42, no. 1, pp. 1–10, 4/10 1997.

[13] Davis, T. A. and Hu, Y., "The University of Florida Sparse Matrix Collection," in *ACM Transactions on Mathematical Software*, vol. 38, 2011, pp. 1–25.

[14] M. Rahmati, M. S. Sadri, and M. A. Naeini, "FPGA based singular value decomposition for image processing applications," in *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, 2008, pp. 185–190, iD: 1.

[15] K. Kota and J. R. Cavallaro, "Pipelining Multiple SVDs on a Single Processor Array," in *Proc. SPIE Conference on Advanced Signal Processing Algorithms, and Implementations V*, vol. 2296, 1994, pp. 612–623.

[16] J. Zhou, D. Y., Z. J., X. F., Y. Lei, and Y. Tang, *A Fine-Grained Pipelined Implementation for Large-Scale Matrix Inversion on FPGA*, ser. Lecture Notes in Computer Science. Springer, 2009, vol. 5727, pp. 110–122.

[17] J. R. Cavallaro and F. T. Luk, "CORDIC Arithmetic for an SVD Processor," *Journal of parallel and distributed computing*, vol. 5, no. 3, pp. 271–290, 1988.