A Scalable FPGA Engine for Parallel Acceleration of Singular Value Decomposition

Yu Wang[1], Jeong-Jun Lee[1], Yu Ding[2], and Peng Li[1]
[1]Department of Electrical and Computer Engineering, University of California Santa Barbara
[2]Department of Industrial and Systems Engineering, Texas A&M University
Email: [1]{yu95, jeong-jun, lip}@ucsb.edu, [2]{yuding}@tamu.edu

Abstract

Singular value decomposition (SVD) is a fundamental computational kernel and tool wildly used in data analytics such as least squares regression, principle components analysis (PCA), and pattern recognition. While a number of dedicated hardware processors have been proposed to accelerate the computationally intensive SVD computation, these designs suffer from poor flexibly and scalability, and/or lack full consideration of compute and data movement challenges associated with SVD. This paper presents a scalable parallel SVD FPGA engine based on the Hestenes-Jacobi method. We propose a so-called Maximum Data Sharing (MDS) ordering, which maximizes on-chip data reuse, and significantly reduces the expensive off-chip data movements and bandwidth requirement. Our SVD engine can flexibly decompose rectangular matrices with variable sizes and speed up SVD computation by 80X to 300X when compared with software SVD solvers such as the Eigen package running on high-performance CPUs. It can process much larger matrices than the previously reported FPGA designs.

Keywords

Singular Value Decomposition, Hestenes-Jacobi Method, FPGA, Hardware Acceleration, Data Reuse

1. Introduction

As a fundamental computational kernel and tool, Singular Value Decomposition (SVD) has been widely adopted in data analysis such as pattern recognition [14]. However, SVD is computationally intensive and most SVD algorithms have a complexity cubic in problem size, rendering SVD is a key bottleneck, particularly for real-time data processing [15].

Among all SVD algorithms, the family of Jacobi methods is most accurate and numerically stable. Despite the Jacobi methods have a low theoretical convergence rate, they are amenable to parallel processing and have been targeted extensively for hardware acceleration. Two kinds of Jacobi methods have been implemented for dedicated hardware SVD processors. A systolic array approach based on the classic two-sided Jacobi algorithm was proposed in [2]. Modified systolic SVD architectures have been implemented on recent FPGAs [3][13]. However, the two-sided Jacobi algorithm strictly requires the input matrix to be square. Moreover, systolic arrays suffer from poor scalability.

In comparison, the one-sided Hestenes-Jacobi method [1] is more appealing for hardware acceleration, e.g. on GPU [4][5], as it overcomes the key limitations of the two-sided Jacobi algorithm and is applicable to general rectangular matrices. Nevertheless, these designs fail to provide large speedups over SVD solvers on general-purpose CPUs, which may be attributed to frequent thread synchronizations as discussed in [4]. On FPGA, a fixed-point design is demonstrated in [7]. Its many limitations include low dynamic range, limited on-chip, and small problem scale with the maximum demonstrated matrix size of 32×128. [8] presents an FPGA architecture that only computes the singular values without singular vectors for relatively large matrices from 100×100 to $2{,}000 \times 2{,}000$. The key issue there is that it does not explore on-chip data reuse, leading to severe performance degradations once the on-chip cache is used up. While bandwidth and data reuse have been studied under a different context of distributed systems [9], these issues have been rarely focused on in recent SVD FPGA processors.

We present a scalable parallel SVD FPGA engine based on the Hestenes-Jacobi method. Unlike the existing FPGA SVD processors, we present an in-depth analysis of data reuse opportunities and propose a new data ordering scheme maximizing on-chip data reuse, and significantly reducing the expensive off-chip data movements and bandwidth requirement by $2k$, where k is the number of parallel processing components. Furthermore, we explore not only the common column-based parallel processing but also the additional opportunity from row-based parallel processing, further improving throughout and relaxing the need for on-chip caching. Our FPGA architecture and SVD engine can flexibly decompose rectangular matrices with variable sizes, speed up SVD computation by up to two orders of magnitude compared with software SVD solvers such as the Eigen package running on high-performance CPUs, and process much larger matrices than previously reported FPGA designs.

2. Background

2.1 Singular Value Decomposition (SVD)

SVD factorizes a given $m \times n$ $\boldsymbol{A}_{m\times n}$ matrix into:

$$\boldsymbol{A}_{m\times n} = \boldsymbol{U}_{m\times n}\boldsymbol{\Sigma}_{n\times n}\boldsymbol{V}_{n\times n}^{\boldsymbol{T}} \quad (1)$$

where both $\boldsymbol{U}$ and $\boldsymbol{V}$ are unitary matrices and $\boldsymbol{\Sigma}$ is a diagonal matrix. The diagonal elements of $\boldsymbol{\Sigma}$ $(\sigma_1, \sigma_2 \cdots \sigma_n)$ are called the singular values of $\boldsymbol{A}$.

2.2 Hestenes-Jacobi Algorithm

The Hestenes-Jacobi method rewrites SVD as:

$$\boldsymbol{B} = \boldsymbol{AV} = \boldsymbol{U\Sigma} \quad (2)$$

Thus, the decomposition of $\boldsymbol{A}$ is equivalent to applying a right-hand side unitary transformation and orthogonalizing the matrix. Since $\boldsymbol{B}$ is orthogonal, the right-hand side transformation $\boldsymbol{V}$ can be split into a series of iterative orthogonal transformation as:

$$\boldsymbol{V} = \boldsymbol{J}_1\boldsymbol{J}_2\boldsymbol{J}_3\boldsymbol{J}_4 \cdots$$

$$\text{where: } \boldsymbol{J}_{p,q} = \begin{array}{c|ccccc|} & & p & & q & \\ p & 1 & & \cdots & & 0 \\ & & c & \cdots & -s & \\ & \vdots & \vdots & \ddots & \vdots & \vdots \\ q & & s & \cdots & c & \\ & 0 & & \cdots & & 1 \end{array}_{n\times n} \tag{3}$$

In each $\boldsymbol{J}$ matrix, by choosing the correct rotation parameters, two columns in $\boldsymbol{A}$ can be orthogonalized:

$$\boldsymbol{A}'_p, \boldsymbol{A}'_q = [\boldsymbol{A}_p, \boldsymbol{A}_q] \cdot \begin{bmatrix} c & -s \\ s & c \end{bmatrix}, \ where\ \boldsymbol{A}'_p \cdot \boldsymbol{A}'_q = \boldsymbol{0} \tag{4}$$

Rotation parameters can be calculated with eq. (5) in [10]:

$$let\ Q = n_p - n_q, P = 2cov, V = \sqrt{Q^2 + P^2}$$
$$if\ Q \geq 0: c = \sqrt{\frac{V+Q}{2V}},\ s = \frac{P}{\sqrt{2V(V+Q)}}$$
$$if\ Q < 0: s = sign(P) \cdot \sqrt{\frac{V-Q}{2V}},\ c = \frac{P}{\sqrt{2V(V-Q)}} \tag{5}$$

The process of orthogonalizing all possible column combinations is called a *sweep*. When convergence is reached after certain sweeps, i.e. all columns are closely orthogonal to each other, $\boldsymbol{\Sigma}$ and $\boldsymbol{U}$ matrix can be extracted by:

$$\boldsymbol{\Sigma} = \sqrt{\boldsymbol{B^T B}},\ \boldsymbol{U} = \boldsymbol{B}/\boldsymbol{\Sigma} \tag{6}$$

Without losing generality, we assume all matrices have a size of $m \times n, m \geq n$. The original Hestenes-Jacobi algorithm without data ordering optimization is shown in algorithm 1:

Algorithm 1 Hestenes Jacobi Algorithm

```
Input: $\boldsymbol{A}_{m\times n}, error$
Output: $\boldsymbol{U}_{m\times n}, \boldsymbol{\Sigma}_{n\times n}, \boldsymbol{V}_{n\times n}$
 1: repeat
 2:    for $i = 1$ to $numofcol$ - 1 do
 3:       for $j = i + 1$ to $numofcol$ do
 4:          Caculate: $n_i, n_j, cov$
 5:          if $|cov| \geq error$ then
 6:             Calculate: $c, s$
 7:             Update: $\boldsymbol{A}_i, \boldsymbol{A}_j, \boldsymbol{V}_i, \boldsymbol{V}_j$
 8:          end if
 9:       end for
10:    end for
11: until Convergence is reached
12: for $i = 1$ to $numofcol$ do
13:    $\boldsymbol{\Sigma}_{ii} = \sigma_i = \sqrt{\boldsymbol{A}_i * \boldsymbol{A}_i}$
14:    $\boldsymbol{U}_i = \boldsymbol{A}_i / \sigma_i$
15: end for
```

3. Data Ordering

3.1 Data Reuse in Hestenes-Jacobi Architecture

At each sweep in the Hestenes-Jacobi algorithm, different column pairs can be processed simultaneously with multiple processing units (PU). In existing architectures, each PU is fully pipelined to maximize throughput [8] while the number of PUs implementable is severely limited by the available bandwidth as shown in Fig. 1. Data caching and reuse shall be exploited to minimize the frequent data movements between the on-chip processing units and off-chip memory. To exploit the parallel computation and data reuse, data ordering is introduced to determine which two columns of the matrix shall be processed by each PU at a given processing step to allow *column-based* parallel processing.

3.2 Existing Round-Robin and Ring Ordering

Round-robin ordering and ring ordering are two popular orderings adopted in many Hestenes-Jacobi applications [9]. In this paper, we adopt these two data ordering as the basis for comparison. With the input matrix with 8 columns and 2 PUs, an example of round-robin ordering and ring ordering is shown in Fig. 2 and Fig. 3, respectively.

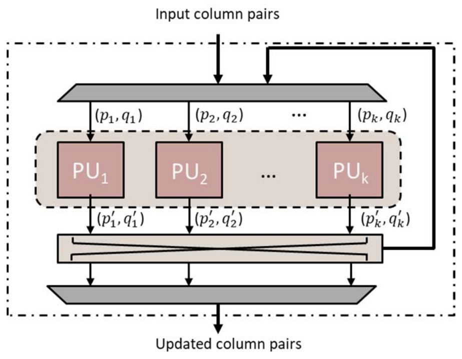

Fig. 1: General Hestenes-Jacobi architecture.

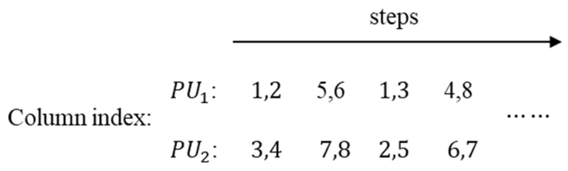

Fig. 2: Round-robin ordering: the pair "a,b" present the indexes of the two columns processed by a PU at a given step.

Round-robin ordering is easy to generate but there is no sharing of column data between different PUs or different processing steps. Ring ordering can reuse data within the same PU by caching the columns orthogonalized in the previous step. But the data reuse between different PUs requires complex interconnection network on-chip.

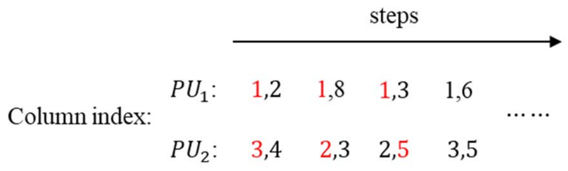

Fig. 3: Ring ordering: columns in red are cached and reused in the same PU at the next step.

4. Proposed Data Ordering and SVD Architecture

4.1 Proposed Maximum Data Sharing Ordering

We make two key observations on the possible data reuse in Jacobi like algorithms by caching processed columns in local PU memory:

a. Column processed in the previous step may be reused by the same PU in the next step.
b. Column processed in the previous step can be reused by another PU in the next step.

Unlike the existing orderings, we propose a new Maximum Data Sharing (MDS) ordering which simultaneously explores the above two opportunities, as shown in algorithm 2. Furthermore, the MDS ordering not only maximizes the data reuse between on-chip PUs, but also simplifies the required architecture, i.e. the communication network between PUs. We show an example of this ordering for a matrix with 8 columns and 4 PUs in Fig. 4.

4.2 Proposed Linearly-Connected (LC) PU Array

Important to note that the data sharing between different PUs in the proposed ordering has a fixed pattern with no requirement for a complex interconnection network. To take advantage of this property, a linearly-connected (LC) PU array architecture is designed as in Fig. 5.

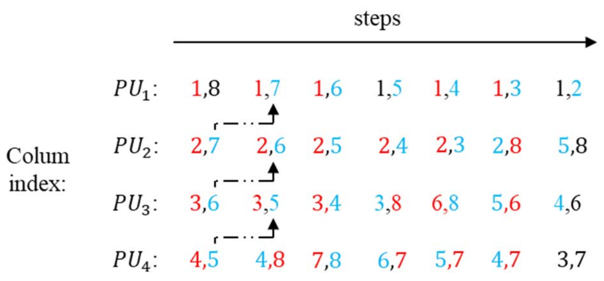
steps
Colum index:
PU_1: 1,8 1,7 1,6 1,5 1,4 1,3 1,2
PU_2: 2,7 2,6 2,5 2,4 2,3 2,8 5,8
PU_3: 3,6 3,5 3,4 3,8 6,8 5,6 4,6
PU_4: 4,5 4,8 7,8 6,7 5,7 4,7 3,7

Fig. 4: Proposed data ordering. Columns with red index are reused by the same PU while the ones with blue index are reused by another PU in the following steps.

Algorithm 2 Proposed Data Ordering Generation

Input: Matrix column number: n, number of PU: k
Output: Column pair indices for ith step: $(p_1^i, q_1^i), (p_2^i, q_2^i), ...(p_k^i, q_k^i)$

```
1:  for $l = 0$ to $\frac{n}{2k} - 1$ do
2:    for $j = 1$ to $k$ do
3:      $p_j^0 = j + k * l, q_j^0 = n + 1 - j - k * l$
4:      $flag_j = 1$
5:    end for
6:    for $i = 1$ to $n - 1$ do
7:      for $j = 1$ to $k$ do
8:        if $flag_j == 1$ then
9:          if $p_j^{i-1} == q_j^{i-1} - 1$ then
10:           $p_j^i = p_j^{i-1}, q_j^i = n, flag_j = 2$
11:         else
12:           $p_j^i = p_j^{i-1}, q_j^i = q_i^{i-1}, flag_j = 1$
13:         end if
14:       else if $flag_j == 2$ then
15:         $p_j^i = p_j^{i-1} + \frac{n}{2k} - 1, q_i^i = q_j^{i-1}, flag_j = 3$
16:       else
17:         $p_i^j = p_i^{i-1} - 1, q_j^i = q_j^{i-1}$
18:       end if
19:     end for
20:   end for
21: end for
```

In each PU, two local memories cache the two columns to be orthogonalized and are configured as either *private* or *shared* during the process. The memory in which any column to be reused by the same PU in the next step resides is configured as *private*. Otherwise, the memory is configured as *shared*. By the end of each step when the columns in each PU have been updated, the column from the private memory (ones with red indices in Fig. 4) will be written back to the same memory while the column from the shared memory (ones with blue indices in Fig. 4) will be sent to the shared memory of the "next" PU. In the next step, the memories of each PU will be configured again and the same process continues.

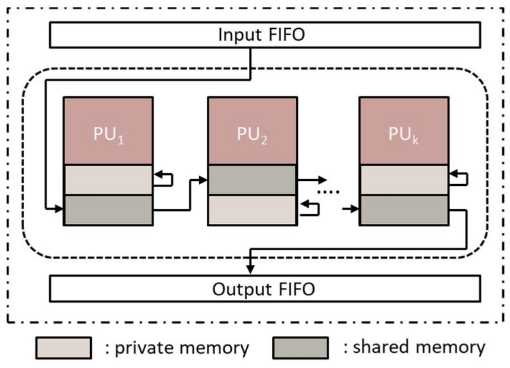

Fig. 5: Proposed linearly-connected (LC) PU array.

As an example, data sharing in MDS ordering for a matrix with 8 columns in an array of two PUs, e.g. the first two steps of PU₁ and PU₂ in Fig. 4, is shown in Fig. 6.

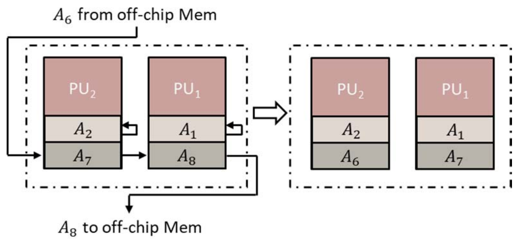

Fig. 6: Data sharing in the proposed LC PU array.

In general, we assume the target matrix has n columns and the PU array has a size of k. In our MDS ordering, during a sweep, there will be $\frac{n}{2k}$ steps where the columns processed in current step cannot be reused in the next step. In this case, all PUs have to flush the cached data and refill new data. The new data loading will be resolved sequentially as shown in Fig. 7. In all other steps, only two PUs (as shown in Fig. 5) need to communicate with off-chip memory and the required bandwidth is reduced by a factor of $2k$.

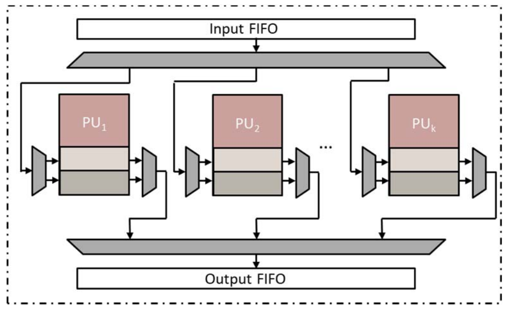

Fig. 7: Data flow for PU array when data sharing is not available.

4.3 Row-based Parallelism and Vector PU

The design of a Hestenes-Jacobi SVD accelerator requires a careful balancing in use of two types of resources:

a. Logic/compute resources, limiting the number of PUs that can be realized;

b. On-chip memory (BRAM) resources, limiting the maximum column size and number of columns can be cached.

For matrices with a large height, the local memory of each PU shall be sufficiently large to cache one or multiple columns. Due to the high requirement of local memory per PU, the number of PUs that can be supported by the available on-chip memory resources would be fairly small, even though the logic/compute resources may be under-utilized.

Thus, we introduce *row-based* parallelism to add additional degree of design freedom, and to fully utilize the available logic/compute resources and achieve the maximum speedup possible. Vector norm computation and column updating are split into multiple *column segments*, which are processed in parallel in addition to the already adopted column-based parallelism that is discussed before.

In this case, the PU we presented earlier is extended to a vector PU as shown in Fig. 8. Within the vector PU, each PU block is responsible for caching and updating one segment of a column pair.

4.4 Proposed SVD Architecture with Row & Column-based Parallelism

The overview of our proposed Hestenes-Jacobi SVD architecture is shown in Fig. 9. Our system consists of mainly three parts: a vector PU array, a data ordering generator, and a rotation parameter calculator. The data ordering generator is responsible for generating desired column indices based on the MDS ordering for all PUs. The vector PU array has multiple vector PUs. At a given step, each vector PU caches a column pair, calculates the corresponding norms, and updates the two columns. An input FIFO and an output FIFO are used to synchronize data between the PU array and off-chip memory.

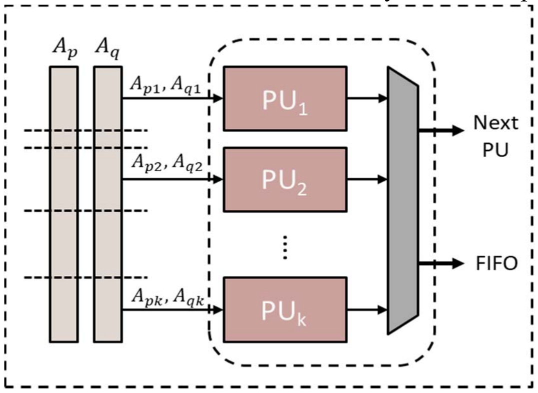

Fig. 8: An example of a vector PU.

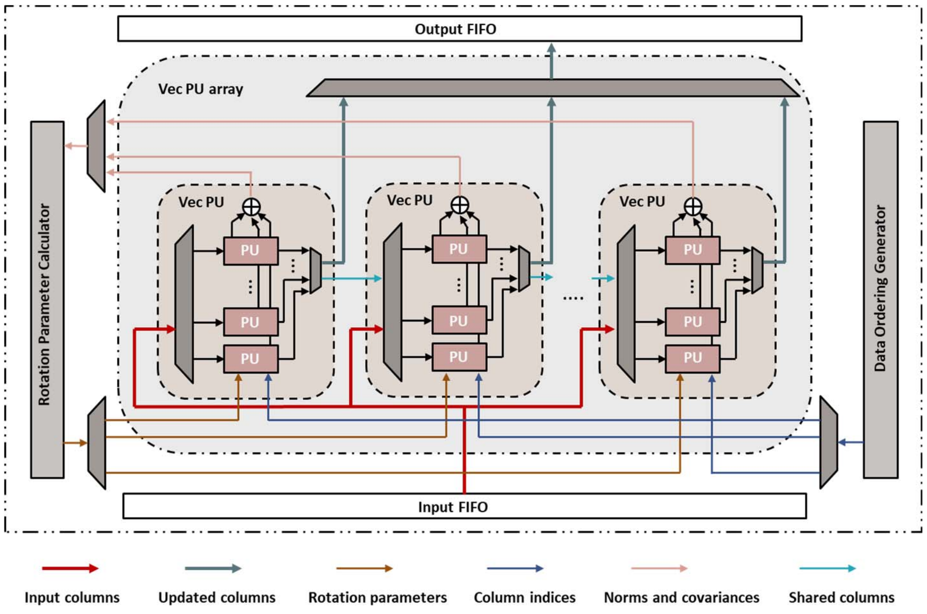

Fig. 9: The proposed SVD architecture with the MDS ordering and both column and row-based parallelisms.

5. Implementation Details

5.1 Data Ordering Generator

The generation of data ordering is done by the data ordering generator. Algorithm 2 is utilized to generate the MDS ordering for arbitrary matrix sizes with simple sequential logic. The data ordering generator can also be reconfigured to generate the round-robin ordering and the ring ordering, allowing us to make comparison between different data orderings.

5.2 Processing Unit

Covariance calculation and column updating in algorithm 1 are done in each PU which consists three modules: covariance calculator, update kernel, and local memory. Fig.10 shows a single PU (w/o row-based parallelism).

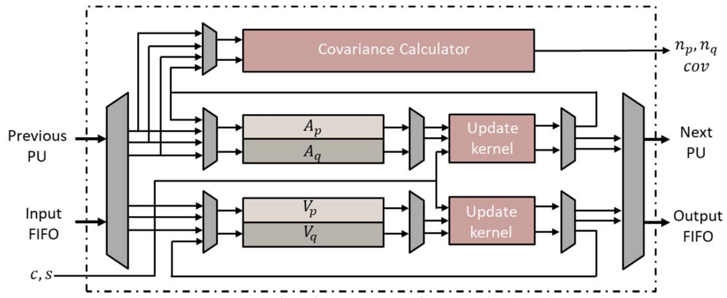

Fig.10: A single processing unit (PU).

The covariance calculator consists three floating point multiplier and three floating point accumulators (Fig. 11 (left)). The accumulated results will be used to calculate rotation parameters. The update kernel with four floating point multipliers and two floating point adders performs Jacobi rotation with eq. (4), and is shown in Fig.11 (right).

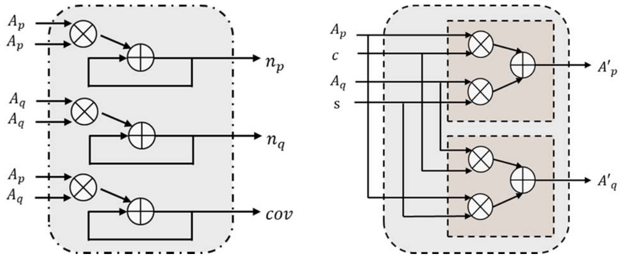

Fig. 11: Covariance calculator (left) and update kernel (right).

5.3 Rotation Parameter Calculator

The calculation of rotation parameters is performed in rotation parameter calculator with eq. (5), which has a comparator, five adders/subtractors, three multipliers, three square root operators and a divider. Two shift registers are utilized to hold the intermediate value P and Q (Fig. 12).

6.2 Resource Utilization and Power Dissipation

Resource utilizations (in percentages of total amount available on the FPGA) and power dissipations of the three designs are summarized in Table 1, which are approximately proportional to the total number of single PUs in each design.

Table.1: Resource utilizations and power dissipations.

Utilization: %	20 Single PU Array	10 Single PU Array	10 Vector PU Array
LUTs	41.8	21.8	42.2
FFs	33.6	17.4	33.81
DSP48	79.1	40.4	79.3
BRAMs	52.1	26	26
Power(W)	3.886	1.964	3.91

6.3 Performance Analysis

Comparison of execution time is made between our FPGA designs and two software SVD routines in the Eigen library.

Table 2: Execution times (seconds) of the Eigen-Jacobi routine, Eigen-BDC routine, and our FPGA designs.

Matrix	Eigen-Jacobi	Eigen-BDC	10 PU Array	20 PU Array	10 Vector PU Array
100×100	0.3067	0.2367	0.0067	0.0038	0.0040
200×200	2.1533	1.0000	0.0349	0.0191	0.0201
500×500	32.7433	7.5500	0.3684	0.1943	0.2005
$1k \times 1k$	271.2830	41.7800	2.4738	1.2770	1.3020
$2k \times 2k$	2220.7900	260.0030	17.8956	9.1080	9.2080
$4k \times 4k$	20933.3197	1784.7700	135.5832	68.4320	68.8319

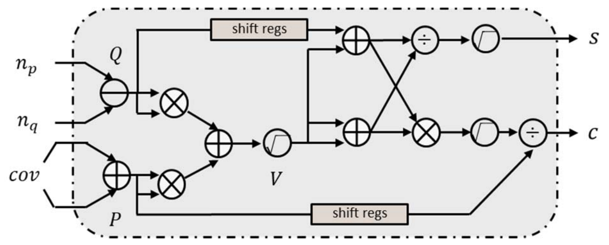

Fig. 12: Design of rotation parameter calculation.

6. Performance Evaluation

6.1 Implementation and Experimental Setup

Our proposed Hestenes-Jacobi accelerators are implemented on a Xilinx ZC706 evaluation board. All data in the architecture is represented in IEEE754 single precision floating point format while the computation components utilize the Xilinx Floating point generator [11]. The on-chip logic is functioning at a frequency of 150MHz. The input matrix and SVD results are stored in the DRAM at the Cortex-A9 ARM core side. The data communication is based on the Xilinx direct memory access (DMA) IP [12].

To demonstrate the proposed architecture, we implement three designs. The first design has an array size of 20 with no row-based parallelism. Each single PU has 0.5Mb local memory and is able to cache columns of a size up to 4,096. The second design is a 10 single PU array system with no row-based parallelism, the local memory size of each PU is also 0.5 Mb. The third design is a vector array containing 10 vector PUs where each vector PU consists of two single PUs.

The Jacobi algorithm and the Bidiagonal-Divide-and-Conquer (BDC) algorithm for SVD in the Eigen package are evaluated on a Xeon E5-2680 CPU clocked at 2.8GHz. The execution times of different implementations are reported in Table 2. Normalized execution times with respect to those of the 20 single PU array clocked at 150MHz are shown in Fig. 13. Compared with the methods in the Eigen library, our accelerator significantly speeds up the runtime by up to 300X.

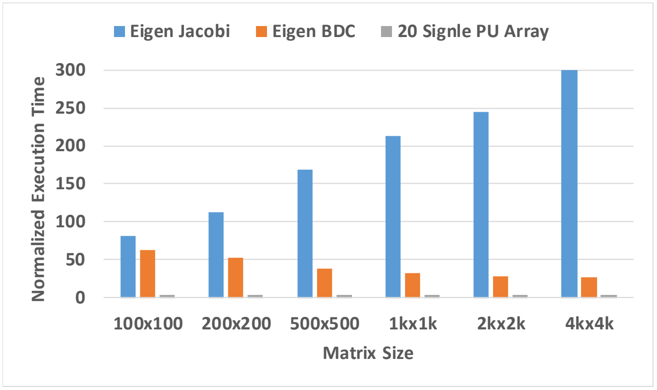

Fig. 13: Normalized execution times of Eigen Jacobi & Eigen BDC @2.8GHz and 20 Single PU Array @150MHz.

Comparison between the three FPGA designs are shown in Fig. 14. The 10 Vector PU Array and 10 Single PU Array both run at 150MHz and have the same total utilized BRAM resources while the introduction of the proposed row-based parallelism in the former speeds up the runtime by about 2X. More generally, when on-chip memory is the dominant limiting factor, use of row-based parallelism allows one to explore available logic resources for additional speedups without requiring more on-chip memory.

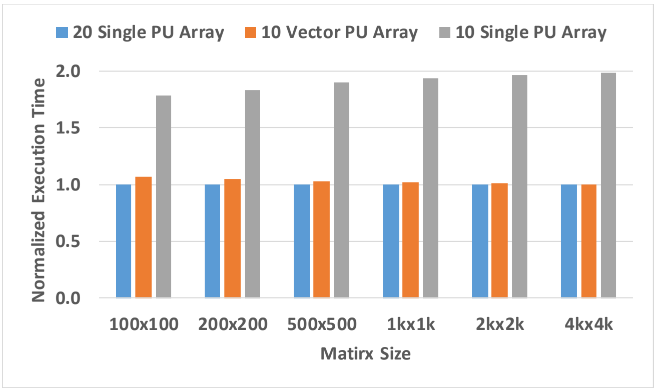

Fig. 14: Normalized execution times of the 20 Single PU Array, 10 Vector PU Array and 10 Single PU Array, all running at 150MHz on the FPGA.

To analyze the MDS ordering, matrices with various dimensions are tested with the round-robin ordering, the ring ordering, and the proposed MDS ordering with the same available bandwidth based on the 20 single PU array design. The execution times of three different data ordering are shown in Table 3. The normalized execution times are shown as Fig.15. When bandwidth is the limiting factor, the MDS ordering is able to achieve significant performance boosts as the data reuse on-chip is well utilized. While in the round-robin and the ring ordering, PUs have to poll for the ownership of the bus to communicate with off-chip memory and cannot function at maximum throughput. With the MDS ordering, we are able to implement more PUs and achieve higher level of parallelism under the same limiting bandwidth, and can achieve close to 20X speedups over the round robin ordering.

Table.3: Execution times (s) of three data orderings.

Matrix	Round-robing	Ring	MDS
100×100	0.021	0.013	0.004
200×200	0.165	0.093	0.019
500×500	2.533	1.330	0.194
$1k \times 1k$	20.133	10.319	1.277
$2k \times 2k$	160.529	81.276	9.108
$4k \times 4k$	1282.115	645.104	68.432

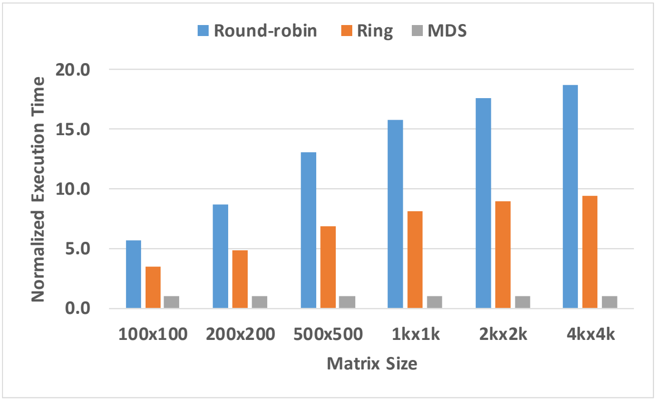

Fig. 15: Normalized execution times of three orderings.

7. Conclusion

In this paper, we propose a new MDS data ordering and SVD accelerator architecture on FPGA. Our design is able to achieve up to 300X speedup compared with the Eigen library. Our experiments also show that the proposed ordering is able to allow for higher degree of parallelism for the Jacobi like algorithms and can reach better performance when bandwidth is limited. Compared with some existing architectures for SVD acceleration, our approach improves at the following aspects: first, the design provides a full solution for matrices of arbitrary dimensions; second, the MDS ordering maximizes the data reuse and allows for higher parallelism of the system; third, the proposed vector PU architecture explores additional row-based parallelism by exploring available logic resources without demanding more on-chip memory.

8. Acknowledgement

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. 1741173. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

9. References

[1] Hestenes, M.R., "Inversion of matrices by biorthogonalization and related results", J. Soc.Indus. Appl.Math.6 (1958), 51-90.

[2] Brent, R.P., and Luk, F.T., "The solution of and symmetric eigenvalue problems on multiprocessor arrays", SIAM J. Sci. Statist. Comput. 6 (1985), 69–84.

[3] W. Ma, et.al, "An FPGA-based singular value decomposition processor", IEEE CCECE, Ottawa, 2006.

[4] C. Kotas, J. Barhen, "Singular value decomposition utilizing parallel algorithm on graphical processors", IEEE OCEANS, 2011.

[5] J. An, D. Wang, "Efficient One-Sided Jacobi SVD Computation on AMD GPU using OpenCL", IEEE ICSP, Chengdu, 2016.

[6] R. Monhanty, et.al, "Design and Performance Analysis of Fixed-Point Jacobi SVD Algorithm on Reconfigurable System", International Conference on Applied Computing, Computer Science, and Computer Engineering (ICACC), 2013.

[7] L. Ledesma-Carrillo, et.al, "Reconfigurable FPGA-Based unit for Singular Value Decomposition of large m x n matrices," Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2011.

[8] X. Wang, J. Zambreno, "An FPGA Implementation of the Hestenes-Jacobi Algorithm for Singular Value Decomposition", 2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshop.

[9] P.J. Eberlein, H. Park, "Efficient implementation of Jacobi alorithms and Jacobi sets on distributed memory architectures", Journal of Parallel and Distributed Computing, Vol. 8. April 1990, 358-366.

[10] L. zhao, Q. Guo, "A Variable Relaxation Parameter for The Parallel One-Sided JRS SVD Algortihm", 7th International Conference on Computer Science & Education (ICCSE), 2012, Melbourne.

[11] Xilinx LogiCORE IP Floating-Point Operator v7.1 Product guide (PG060), https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v7_0/pg060-floating-point.pdf

[12] Xilinx AXI DMA IP Product Guide (PG021), https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

[13] A. Ahmedsaid, A. Amira, A. Bouridane, "Improved SVD systolic array and implementation on FPGA", IEEE International Conference on Field-Programmable Technology (FPT), 2003 Tokyo.

[14] Md Sahidullah, Tomi Kinnunen, "*Local spectral variability features for speaker verification*," Digital Signal Processing, Vol. 50, March 2016, 1-11.

[15] M. V. Athi, S. R. Zekavat, "Real-Time Signal Processing of Massive Sensor Arrays via a Parallel Fast Converging SVD Algorithm: Latency, Throughput, and Resource Analysis", IEEE Sensor Journal, Vol. 16, No. 8, April 15, 2016.