A PARALLEL FPGA SVD ACCELERATOR WITH OPTIMIZED ON-CHIP DATA REUSE

A Thesis

by

YU WANG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Peng Li
Co-Chair of Committee,	Jeyavijayan Rajendran
Committee Member,	Eun Jung Kim
Head of Department,	Miroslav M. Begovic

December 2019

Major Subject: Computer Engineering

ABSTRACT

In recent years, the emerging of new machine learning algorithms and the fast development of available hardware allow people to perform complex recognition and generation tasks of images, natural languages and speaking voices. Although the general-purpose CPUs are continuously being optimized, it still takes massive time to finish training or even perform recognition with the growing size of datasets. The pressing need for fast, energy and area efficient hardware platforms make research community and the industry focus on developing dedicated hardware accelerators for running computationally intensive data processing algorithms.

Singular value decomposition (SVD) is a fundamental computation kernel which has been wildly applied in pattern recognition, matrix compression and signal processing. However, limited by the high computation complexity, it usually is the most time-consuming part in many data pre-processing schemes and machine learning algorithms.

While many existing works have proposed their parallel computing architectures for SVD, these works are either limited by the strict requirement of matrix sizes, or the flexibility and scalability of their architectures. Moreover, the data movement issue, which serves as a critical bottleneck in parallel computation architectures for SVD, has rarely been discussed.

In this thesis, we propose a new Maximum Data Sharing ordering (MDS ordering) and corresponding Field Programmable Logic Array (FPGA) architecture to maximize the data reuse on-chip and can significantly reduce the bandwidth requirement. The proposed

reconfigurable SVD engine can decompose matrices with arbitrary sizes much larger than existing solutions and can reach a speed up of 80X to 300X compared with the results of Eigen package running on high performance CPU.

DEDICATION

Alea jacta est!

ACKNOWLEDGEMENTS

At the very first beginning, I would like to thank Dr Peng Li, my committee chair and Dr Jeyavijayan Rajendran as well as Dr Eun Jung Kim, my committee member for the guidance in completing the research work and suggestions in my graduate years. Learning how to start a research, complete a research and most importantly, improve the research is the most precious treasure I found in A&M.

I also want to thank my friends, both in AMS group and CE group. I'm fortunate enough to meet you and share the meaningful days together in this flashing two years. Your enthusiasm and sincerity often make me feel flattered.

Finally, I must thank my parents for their unconditional love and support, the unlimited power in fighting the loneliness and difficulties in this arduous journey of learning abroad. Without you, I will never have been able to go this far.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee of Professor Peng Li, Professor Jeyavijayan Rajendran of the Department of Electrical and Computer Engineering and Professor Eun Jung Kim of Computer Science and Engineering.

Part of the work is collaborated with Jeong-Jun Lee, a Ph.D. candidate at Texas A&M University. The rest of the work conducted for the thesis was completed by the student independently.

There are no outside funding contributions to acknowledge related to the research and compilation of this document.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

1. INTRODUCTION

1.1. Background

Since its first appearance in 1940s, the performance of computers has been continuously improved. Computer engineers have been trying to push the throughput of single core processors by utilizing pipelining, instruction level parallelism (ILP) and caching techniques. However, performance improvement starts to degrade in the early 21st century.

Meanwhile, engineers started to look for alternative ways to further improve the throughput. Some of them focused on thread level parallelism and developed the multi-core processors. The others utilized the data level parallelism and proposed the idea of heterogeneous computing.

In recent days, heterogeneous computing is being frequently discussed mainly for the following two reasons. First, application that require parallel computation for acceleration is increasing, from PC gaming, to the big data center analysis, and the emerging machine learning field. Multicore processors are not able to achieve decent performance in these application specific tasks. Second, the development in hardware devices and techniques allow integration of different co-processors. Additionally, the development of general high-speed buses, e.g. PCIE and SATA, makes the integration and data communication more efficient. For users and developers, it is convenient to be not restricted to the CPU-GPU architecture, but also to be able to choose the CPU-FPGA for flexible design or the CPU-ASIC for good application specific performance.

Among them, the CPU-PFGA architecture is attracting more and more attention in heterogeneous computation. It can be quickly developed and deployed for a specific application scenario and achieve acceptable performance boost without the huge investigation needed as that of AISC circuits design. Although compared with GPUs, FPGA has fewer available computation resource on-chip, the programmable characteristic gives it the flexibility to well organize the data flow and makes the implementation be more efficient. Noticeably, in the current data science and machine learning area, new algorithms are developing fast and some of them are difficult for GPU users to program as GPU programmers have less control in data sharing between threads. For this reason, for architecture research, as well as commercial data centers, CPU-FPGA architecture is gaining increasing popularity.

Singular value decomposition (SVD) is a fundamental computation method that is applied to many machine learning algorithms. It is wildly used in data processing, matrix compression and many recognition tasks [15][16]. However, most SVD algorithms have a complexity which is cubic in problem size and severely limits the application of SVD in many real time tasks [17]. As a result, many architecture researches have been done on SVD of utilizing parallel algorithms and implementing them on FPGA or GPUs.

Among the existing algorithms, the Jacobi algorithm has the highest accuracy and numerical stability while the convergence rate is slow. However, the inner parallelism of Jacobi algorithm makes it well suitable for parallel acceleration of SVD. Two kinds of Jacobi methods, the two-sided Jacobi method and the Hestenes-Jacobi method have been extensively targeted for developing parallel architectures. The first systolic array

architecture utilizing two-side Jacobi method is reported in [2]. Modified systolic arrays designed for modern FPGAs have also been implemented in [3][14]. However, these architectures are limited by a strict requirement for square input matrices and they are difficult to scale up with respect to problem size.

In contrast, the Hestenes-Jacobi method, which is proposed in [1] is more flexible for hardware acceleration as it can decompose rectangular matrices. GPU application of the algorithm has been reported in [4][5][6]. Nevertheless, the designs are not able to show reasonable speedups compared with SVD engines on CPU due to the frequent data synchronization as discussed in [4]. On the reconfigurable family side, a fixed-point implementation is shown in [8], with a low dynamic range and restriction on computation of matrices no large than 32×128. In [9], an architecture that computes only singular values for matrices with a size from 100×100 to $2k \times 2k$ is presented. However, the design does not give a solution to the on-chip data reuse and faces a severe performance degradation when the on-chip cache is fully utilized. In [10], the data reuse for the Hestenes-Jacobi method implemented on distributed memory systems have been discussed, but related studies have not been explored in recent FPGAs.

In this thesis, we present a scalable SVD engine utilizing the Hestenes-Jacobi method and our proposed Maximum Data Sharing ordering (MDS ordering). Our architecture can maximize the data reuse in the Hestenes-Jacobi algorithm and can significantly reduce the bandwidth requirement by $2k$, while k is the number of parallel processing elements. Meanwhile, to further improve the throughput and reduce the on-chip cache requirement of the design, we also explore the row-based parallelism and

integrate it into our SVD architecture. With the application of the MDS ordering and utilization of both column & row-based parallelism, our architecture can reach up to 300X speed up compared with the solvers in the Eigen package running on high performance CPU. Compared with existing FPGA implementation, our SVD engine can give a full solution to much larger matrices.

1.2. Organization of thesis

The dissertation can be divided into the following parts. In Section 2, we will give a brief introduction to SVD and the Jacobi method. In Section 3, parallelism of the Hestenes-Jacobi method and data reuse possibility will be discussed. We will also give a definition of data ordering in the Hestenes-Jacobi method and present two existing data orderings. In Section 4, we will demonstrate our proposed MDS ordering and the corresponding hardware architecture. In Section 5, detailed implementation of different parts of the system will be discussed. In Section 6, we will show the experiment setup and analyze the experiment results. In the final part, we will conclude the dissertation and discuss about the future works.

2. JACOBI METHOD IN SINGULAR VALUE DECOMPOSITION

2.1. Singular Value Decomposition

Singular value decomposition is defined as a factorization of a matrix in linear algebra. Given a matrix $\boldsymbol{A}_{m\times n}$, the decomposition can be written as:

$$\boldsymbol{A}_{m\times n} = \boldsymbol{U}_{m\times n}\boldsymbol{\Sigma}_{n\times n}\boldsymbol{V}^{\boldsymbol{T}}_{n\times n}\ , \tag{2.1}$$

where $\boldsymbol{U}, \boldsymbol{V}$ are both unitary matrices and $\boldsymbol{\Sigma}$ is a diagonal matrix. The diagonal elements of $\boldsymbol{\Sigma}$, i.e. σ_i are called the singular values of $\boldsymbol{A}$.

Without losing generality, we assume the matrices discussed in our dissertation are full ranked, i.e. $rank(\boldsymbol{A}) = min\{m, n\}$. Under this assumption, the columns in unitary matrix $\boldsymbol{U}$ form a set of orthogonal bases in the column space of $\boldsymbol{A}$, while columns in $\boldsymbol{V}$ form a set of orthogonal bases in the row space of $\boldsymbol{A}$. Noticeably, although given a specific matrix, the singular values are determined, the decomposition itself is not unique, as the orthogonal bases for row or column space of $\boldsymbol{A}$ can be chosen arbitrarily.

2.2. Jacobian Rotation

In linear algebra, Jacobi rotation is defined as a unitary transformation for a given 2×2 real symmetric matrix:

$$\begin{bmatrix} b_1 & 0 \\ 0 & b_2 \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}^T \begin{bmatrix} a_1 & a_2 \\ a_2 & a_3 \end{bmatrix} \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \tag{2.2}$$

The rotation parameters, c, s can be calculated with eq. (2.3):

$$let\ \beta = \frac{a_3 - a_1}{2a_2}, t = \frac{sgn(\beta)}{|\beta| + \sqrt{\beta^2 + 1}}$$

$$c = \frac{1}{\sqrt{t^2+1}}, s = c \cdot t \tag{2.3}$$

Jacobian rotation is a unitary transformation and can be applied to diagonalize symmetric matrices. Due to its property, it serves as the basic transformation in the two-sided Jacobi method.

2.3. Two-sided Jacobi Method

The Two-sided Jacobi method is proposed in [18] to decompose symmetric square matrices $\boldsymbol{A}_{n\times n}$. By utilizing Jacobian rotation for a submatrix $\boldsymbol{A}_{p,q} = \begin{bmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{bmatrix}$ of $\boldsymbol{A}$, the off-diagonal elements of $\boldsymbol{A}_{p,q}$ can be annihilated.

Jacobian rotations for different sub-matrices in $\boldsymbol{A}$ can be applied simultaneously, i.e. given $\boldsymbol{A}_{p,q}, \boldsymbol{A}_{k,l} (p, q \neq k, l)$, the diagonalizations can be performed at the same time. Thus, parallel acceleration of Jacobi method can be done by performing Jacobian rotations for different sub-matrices.

However, several limitations prevent the wild application of the two-sided Jacobi method. First, the input matrix has to be a square symmetric matrix $\boldsymbol{A}_{n\times n}$. Second, the number of parallel computation units needed is quadratic proportional to n, which severely limits the scalability of the algorithm.

2.4. One-sided Jacobi Method

To overcome the deficiencies in the two-sided Jacobi method, the Hestenes-Jacobi method is proposed in [1]. By rewriting the SVD definition as eq. (2.4):

$$\boldsymbol{B} = \boldsymbol{AV} = \boldsymbol{U\Sigma} \tag{2.4}$$

As $\boldsymbol{U}$ is unitary and $\boldsymbol{\Sigma}$ is diagonal, the left-hand side of the equation must be an orthogonal matrix, i.e. the SVD process becomes applying a unitary transformation to $\boldsymbol{A}$ and makes it become orthogonal. Meanwhile, the right-hand side transformation $\boldsymbol{V}$ can be split into a series of iterative orthogonal transformations as shown in eq. (2.5):

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

where:

$$J_{p,q} = \begin{array}{c} \\ p \\ \\ q \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} & & p & & q \end{array} \\ \begin{bmatrix} 1 & & \cdots & & 0 \\ & c & \cdots & -s & \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ & s & \cdots & c & \\ 0 & & \cdots & & 1 \end{bmatrix}_{n\times n} \end{array} \tag{2.5}$$

In every $\boldsymbol{J}$ matrix, with proper rotation parameters c, s, two columns in $\boldsymbol{A}$ can be rotated and orthogonalized as shown in eq. (2.6).

$$\boldsymbol{A}'_p, \boldsymbol{A}'_q = [\boldsymbol{A}_p, \boldsymbol{A}_q] \cdot \begin{bmatrix} c & -s \\ s & c \end{bmatrix},\ where\ \boldsymbol{A}'_p \cdot \boldsymbol{A}'_q = \boldsymbol{0} \tag{2.6}$$

Rotation parameters can be calculated with eq. (2.7) as presented in [11].

$$let\ Q = n_p - n_q, P = 2cov, V = \sqrt{Q^2 + P^2}$$

$$if\ Q \geq 0: c = \sqrt{\frac{V+Q}{2V}},\ s = \frac{P}{\sqrt{2V(V+Q)}}$$

$$if\ Q < 0: s = \sqrt{\frac{V-Q}{2V}},\ c = \frac{P}{\sqrt{2V(V-Q)}} \tag{2.7}$$

In the Hestenes-Jacobi method, all possible column pairs will be orthogonalized sequentially. This process is called a *sweep*. After certain sweeps, convergence can be reached [1] and the $\boldsymbol{\Sigma}$ and $\boldsymbol{U}$ matrix are then calculated by eq. (2.8).

$$\boldsymbol{\Sigma} = \sqrt{\boldsymbol{B}^T\boldsymbol{B}},\ \boldsymbol{U} = \boldsymbol{B}/\boldsymbol{\Sigma} \tag{2.8}$$

The basic Hestenes-Jacobi algorithm without any optimization in data ordering is shown in algorithm 1.

Algorithm 1 Hestenes Jacobi Algorithm

Input: $\boldsymbol{A}_{m\times n}, error$
Output: $\mathbf{U}_{m\times n}, \boldsymbol{\Sigma}_{n\times n}, \boldsymbol{V}_{n\times n}$

repeat
  for $i = 1$ **to** *numofcol* - 1 **do**
    for $j = i + 1$ **to** *numofcol* **do**
      /* *Calculating 2-norms and covariances* */
      $n_p = \boldsymbol{A}_p * \boldsymbol{A}_p$
      $n_q = \boldsymbol{A}_q * \boldsymbol{A}_q$
      $cov = \boldsymbol{A}_p * \boldsymbol{A}_q$
      /* *Neglect column pairs that are already orthogonal* */
      if $|cov| \geq error$ **then**
        /* *Calculating rotation parameters* */
        $Q = n_p - n_q$
        $P = 2 * cov$
        $V = \sqrt{P^2 + Q^2}$
        if $Q \geq 0$ **then**
          $c = \sqrt{\frac{V+Q}{2V}}, s = \frac{P}{\sqrt{2V(V+Q)}}$
        else
          $c = \frac{P}{\sqrt{2V(V-Q)}}, s = \sqrt{\frac{V-Q}{2V}}$
        end if
        /* *Updating column pairs* */
        $\boldsymbol{A}_p^{'} = \boldsymbol{A}_p * c + \boldsymbol{A}_q * s$
        $\boldsymbol{A}_q^{'} = \boldsymbol{A}_q * c - \boldsymbol{A}_p * s$
        $\boldsymbol{V}_p^{'} = \boldsymbol{V}_p * c + \boldsymbol{V}_q * s$
        $\boldsymbol{V}_q^{'} = \boldsymbol{V}_q * c - \boldsymbol{V}_p * s$
      end if
    end for
  end for
until *Convergence is reached*
for $i = 1$ **to** *numofcol* **do**
  $\boldsymbol{\Sigma}_{ii} = \sigma_i = \sqrt{\boldsymbol{A}_i * \boldsymbol{A}_i}$
  $\mathbf{U}_i = \boldsymbol{A}_i / \sigma_i$
end for

3. DATA ORDERING IN JACOBI METHOD

3.1. Data Reuse and Data Ordering in the Hestenes-Jacobi Architecture

The parallelism of the Hestenes-Jacobi algorithm allows processing of different column pairs by different processing units (PU) at each step, where each processing unit is responsible for calculating Euclidean norms, covariances and updating the column pairs as shown in Algorithm 1. To maximize throughput, each processing unit is designed to be fully pipelined [9]. However, the increasing of problem size results in storing the whole matrix in off-chip memory. To achieve reasonable speedup, the required bandwidth grows dramatically when the number of PUs increases (as shown in Figure 3.1). Thus, proper data cache technique should be introduced to minimize the frequent data transmission between on-chip PUs and off-chip memory.

To well exploit the data reuse opportunities in the Hestenes-Jacobi method, on-chip memory should be employed to cache processed column pairs for further reuse at the following steps. At a certain step in one sweep, the columns being processed and stored should be reused as many as possible. To satisfy the requirement, data ordering is introduced to determine which column pair should be processed by certain PU at each step to achieve column-based parallel processing and minimize the on-chip and off-chip data communication.

3.2. Existing Round-Robin Ordering and Ring Ordering

In [10], round-robin ordering and ring ordering have been implemented and analyzed for distributed memory system. The two orderings are also popular in some

existing architectures [3], and thus, adopted in this dissertation for comparison. As an example, round-robin ordering and ring ordering for matrix with 8 columns and 2 PUs are shown in Figure 3.2 and Figure 3.3.

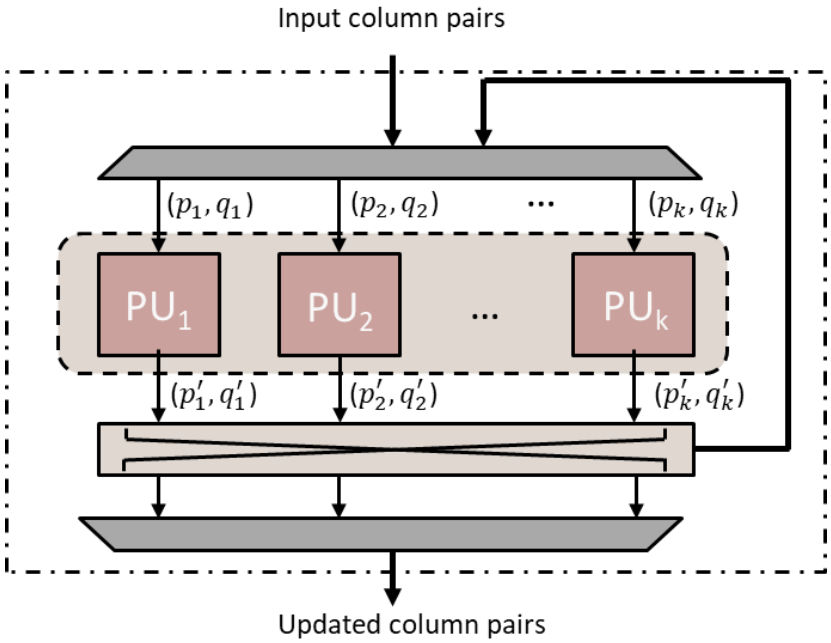

Figure 3.1: General Hestenes-Jacobi architecture.

The generation of round-robin ordering is simple and it can support column-based parallel processing. However, as shown in Figure 3.2, no data sharing is available between different steps, i.e. at each step, all the PUs should request, process and write back new column pairs. In ring ordering, the column updated at the previous step can be reused by the same PU as shown in Figure 3.3. However, to reuse columns between different PUs across different steps, complex interconnection network on-chip is required, especially when the number of PUs increases. Both round-robin ordering and ring ordering cannot fully utilize the data reuse potentiality in the Hestenes-Jacobi architecture.

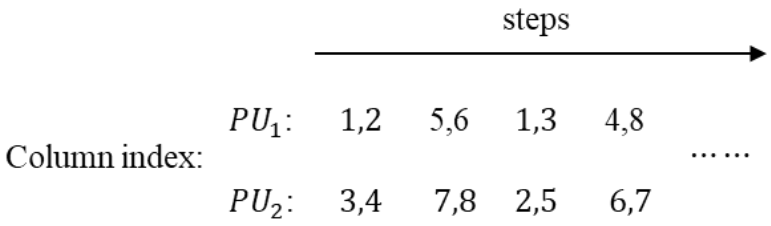

Figure 3.2: Round-robin ordering: “a,b” stands for the indices of the two columns processed by a certain PU at a given step.

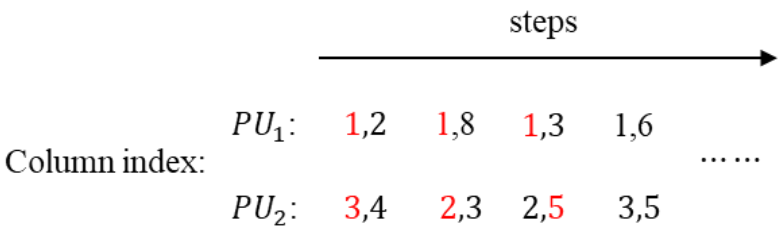

Figure 3.3: Ring ordering: columns in red are cached and will be reused in the same PU at the next step.

4. PROPOSED DATA ORDERING AND SVD ARCHITECTURE

4.1. Proposed Maximum Data Sharing Ordering

To maximize the data reuse between different steps in the Hestenes-Jacobi method, two types of data reuses should be utilized by caching processed columns in on-chip memory:

a. Column processed at previous step shall be reused by the same PU at next step.

b. Column processed at precious step can be reused by another PU at next step.

Furthermore, a well-designed data ordering for parallel acceleration in the Hestenes-Jacobi architecture should satisfy the following three key properties:

a. Parallelism: Different PUs are processing different column pairs.

b. Reusability: The data ordering should utilize both two types of data reuses.

c. Simplicity: The data reuse should not result in complex interconnection network.

As mentioned in section 3.2, round-robin ordering satisfies parallelism only, while ring ordering is parallel but cannot satisfy reusability and simplicity simultaneously. To overcome the limitations, we propose Maximum Data Sharing ordering (MDS ordering) as shown in Algorithm 2. Our MDS ordering is a parallel ordering that not only explores both two types of data reuses, but also simplifies the required architecture. An example of the MDS ordering for a matrix with 8 columns and an architecture of 4 PUs is shown in Figure 4.1.

4.2. Proposed Linearly-Connected (LC) PU Array

As demonstrated in Figure 4.1, at each step of the MDS ordering, the data sharing between different PUs are completed in a fixed pattern, i.e. the communications happen only between a certain PU and its neighboring two PUs. Thus, implementing our MDS ordering does not need a complex interconnection network on-chip. To take advantage of this property, a linearly-connected (LC) PU array architecture is demonstrated as shown in Figure 4.2.

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**

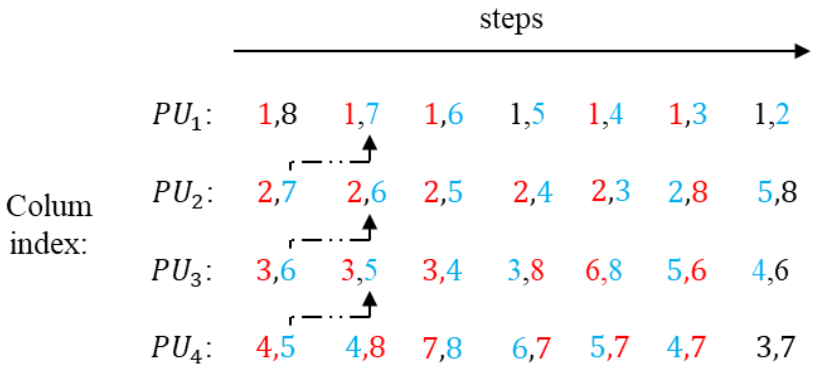

Figure 4.1: Proposed data ordering. Columns with a red index are reused by the same PU while ones with a blue index are reused by another PU in the following step.

In each PU, two local memories cache the two columns to be updated and are configured as either private or shared during a certain step. If a memory caches column that will be reused by the same PU, it will be configured as private. Otherwise, the memory will be configured as shared. At 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 Figure 4.1) will be written back to the same memory while the column from the shared memory (ones with blue indices in Figure 4.1) will be sent to the shared memory of the "next" PU. At the next step, the memories of each PU will be configured again and the same process continues.

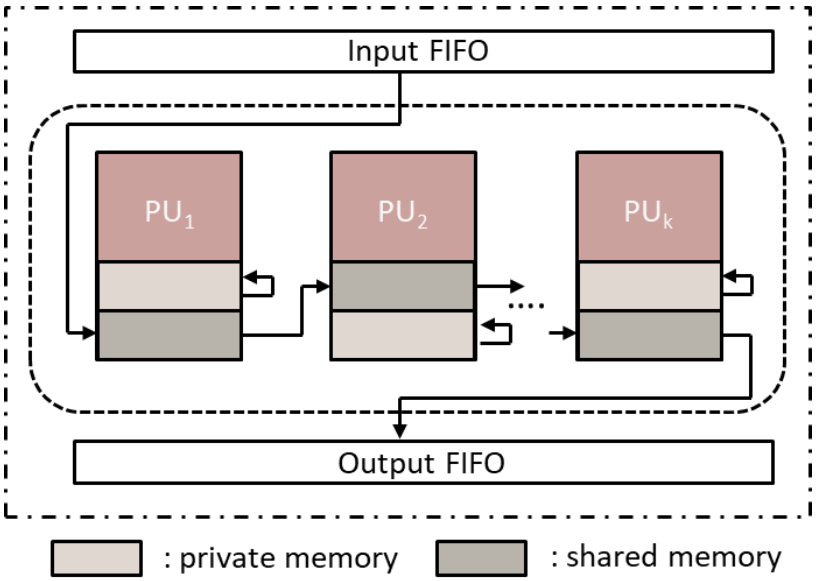

Figure 4.2: Proposed linearly-connected (LC) PU array.

In Figure 4.3, an example data sharing in the MDS ordering for a matrix with 8 columns in an array of two PUs, e.g. the first two steps of PU_1 and PU_2 in Figure 4.1, is demonstrated.

In general, we suppose our target matrix has n columns and the PU array has k PUs. In a sweep (with $\frac{n(n-1)}{2}$ steps) of utilizing the MDS ordering, there are $n/2k$ steps where the columns processed at previous step cannot be reused at next step. In this case, all PUs have to flush the cached data and refill new data. The data communication between the PU array and the off-chip memory will be done sequentially as shown in Figure 4.4. At other steps, only the "first" and the "last" PUs (as shown in Figure 4.2) need to communicate with off-chip memory, thus the required bandwidth is reduced by a factor of $2k$. Since the first situation is not so frequent during a sweep, we are able to implement more PUs without increasing the bandwidth requirement.

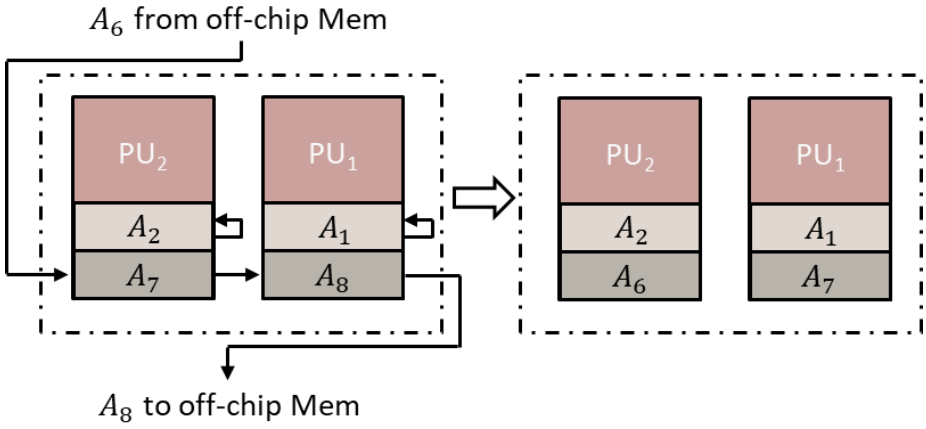

Figure 4.3: Data sharing example in the proposed LC PU array.

4.3. Row-based Parallelism and Vector PU

Noticeably, the design of a Hestenes-Jacobi SVD accelerator needs a careful balancing in the utilization of two types of resources on-chip:

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

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

When the column size is huge, the local memory of each PU has to be sufficiently large to cache the whole column. Due to the high requirement of local memory per PU, the number of PUs that can be supported, i.e. the level of column-based parallelism will be fairly small, despite the fact that logic/compute resources may be still under-utilized.

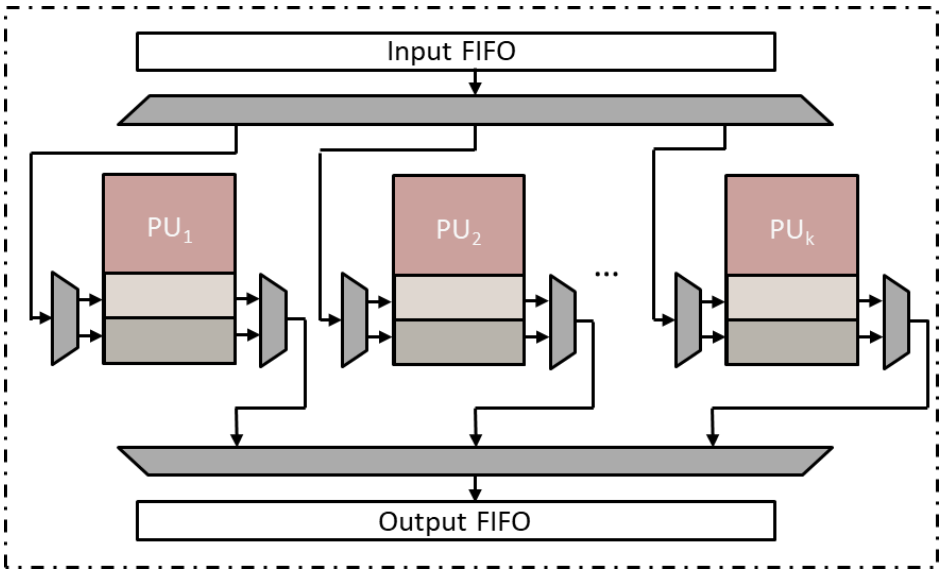

Figure 4.4: Data flow for PU array when data sharing is not available.

To fully utilize the available logic resources on-chip and achieve the maximum throughput, row-based parallelism should be introduced as an additional design freedom. With the row-based parallelism, vector-based computation, i.e. the calculation of norms, covariances and the updating of columns can be split into multiple column segments simultaneously.

In this case, a vector PU can be developed with multiple single PUs as mentioned before (shown in Figure 4.5). Inside the vector PU, each single PU is responsible for caching and computing one segment of a column pair.

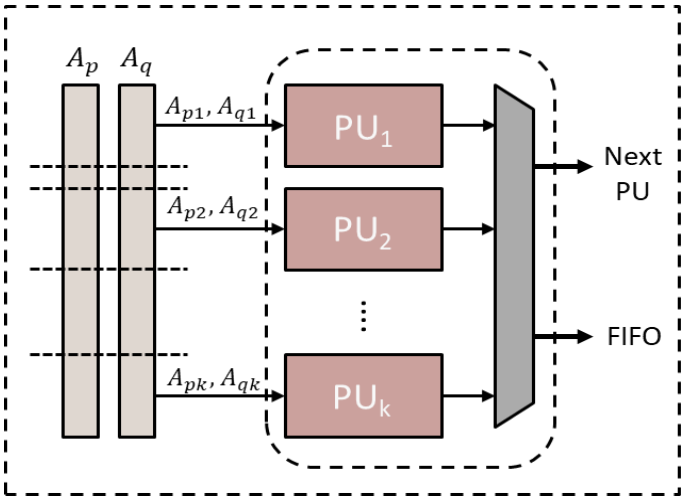

Figure 4.5: An example of a vector PU.

4.4. Proposed SVD Architecture with Row and Column-based Parallelism

Figure 4.6 demonstrates an overview of our proposed Hestenes-Jacobi architecture which can be mainly divided into three parts: a vector PU array, a data ordering generator, and a rotation parameter calculator. The data ordering generator utilizes algorithm 2 to generate required column indices of the MDS ordering for all the PUs. The vector PU array consists of multiple vector PUs. At a certain step, each vector PU is responsible for caching a column pair, calculating the corresponding norms, covariances and updating the two columns. The rotation parameter calculator is employed to generate rotation parameters which are used to perform Jacobian rotation given the covariances and Euclidean norms. Two FIFOs a are implemented to synchronize data between the on-chip processing PU array and off-chip memory.

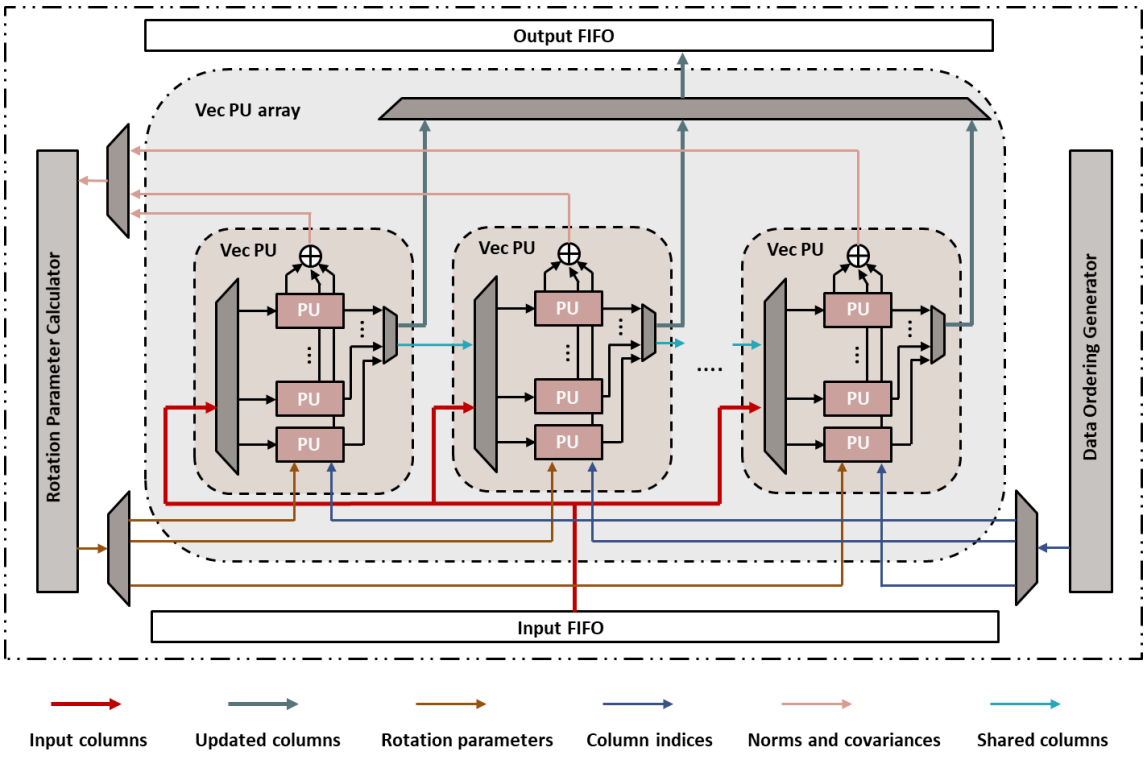

Figure 4.6: The proposed SVD architecture with the MDS ordering and both column and row-based parallelisms.

5. IMPLEMENTATION DETAILS

5.1. Data Ordering Generator

The data ordering is generated by data ordering generator with algorithms 2. The reconfigurable sequential logic in this component can generate data orderings for matrices with arbitrary sizes. After initialization, the component generates all column pairs for k PUs in k clock cycles. Our data ordering generator can also be configured to generate round-robin ordering or ring ordering. Thus, our system is able to utilize, analyze and compare three different orderings.

5.2. Processing Unit

The calculation of the Euclidean norms and updating of columns in algorithm 1 are finished in each PU with three modules: the covariance calculator, the update kernel, and the local memory. A single PU is shown in Figure 5.1.

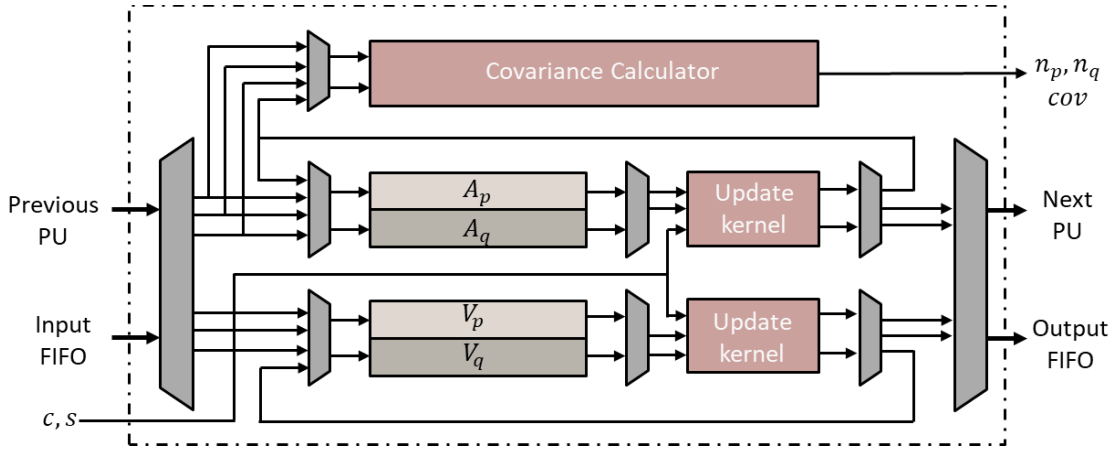

Figure 5.1: The architecture of a single processing unit (PU).

5.2.1. Covariance Calculator

The Euclidean norms and covariances of two columns ($\boldsymbol{A}_p, \boldsymbol{A}_q$) can be calculated with eq. (5.1).

$$n_p = \sum_{i=1}^{n} A_{pi}^2 \, , n_q = \sum_{i=1}^{n} A_{qi}^2 \, , cov = \sum_{i=1}^{n} A_{pi} A_{qi} \tag{5.1}$$

Each covariance calculator consists three floating point multipliers and three floating point accumulators (in Figure 5.2 (left)). When all entries in the two columns are accumulated, the final norms and covariances will be sent to rotation parameter calculator.

5.2.2. Update Kernel

The update kernel performs Jacobi rotation with eq. (2.6). A single update kernel is shown as Figure 5.2(right), where four floating point multipliers and two floating point adders are utilized.

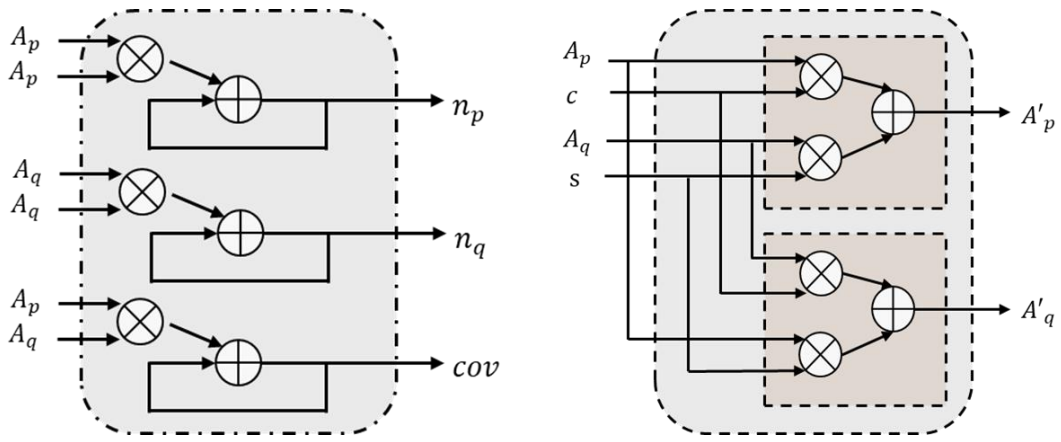

Figure 5.2: Covariance calculator (left) and update kernel (right).

In each PU, two update kernels are employed, one for updating of column pairs in $\boldsymbol{A}$, the other is used to update column pairs in $\boldsymbol{V}$.

In the final processing of calculating $\boldsymbol{U}$ with eq. (2.8), $1/\sigma_p, 1/\sigma_q$ will be provided by rotation parameter calculator and two multipliers will be reconfigured to complete the updating.

5.2.3. Local Memory

To utilize on-chip data reuse, four true dual port BRAMs are employed in each PU and are used to cache the whole column pairs of $\boldsymbol{A}$ and $\boldsymbol{V}$.

At a certain step, the column pairs are firstly cached while their corresponding norms and covariances are calculated. After that, according to the reuse type of the cached columns, the memories will be configured as either private or shared. When the rotation parameters have been calculated and Jacobi rotations have been performed, the updated columns will be sent back to locations according to its memory type: updated columns from private memory will be sent back to the same memory, while the updated columns from shared memory will be sent to another PU or back to off-chip memory.

5.3. Rotation Parameter Calculator

The calculation of the rotation parameters requires multiple steps of multiplication, addition/subtraction, division and square root operation. In previous small-scale application, fixed-point representation is adopted and Coordinate Rotation Digital Computer (CORDIC) algorithm is applied because of its efficiency and simplicity by performing iterative shifting-adding operation [3][7][8]. In our architecture, to achieve large dynamic range and reasonable precision, single precision floating point representation is adopted, and the CORDIC algorithm, limited by inherent shifting-adding

procedure, is no longer suitable for this situation. Thus, the floating-point IP, provided by Xilinx is utilized.

The calculation is performed with eq. (2.7). In our rotation parameter calculator (RP calculator), a comparator, 5 adders/subtractors, three multipliers, three square root operators and a divider are employed. Two shift registers are utilized to hold the inner value P and Q (as shown in Figure 5.3). The component is fully pipelined with a delay of 145 cycles.

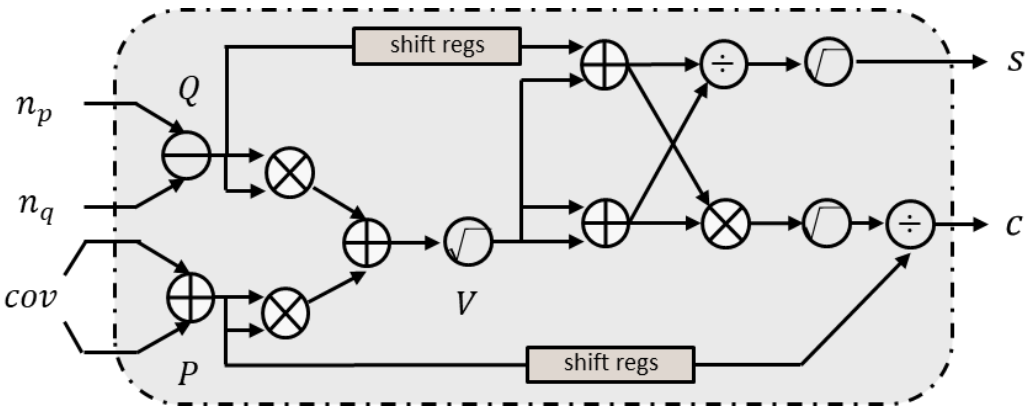

Figure 5.3: The architecture of the rotation parameter calculator.

When the algorithm is converged, the input norms, n_p, n_q are actually the squared pth and qth singular values. The square root operator will be used to calculate the singular values (σ_p, σ_q) and a divider will calculate $1/\sigma_p, 1/\sigma_q$ which will be sent to update kernel.

6. EXPERIMENT SETUP AND RESULTS

6.1. Experiment Setup

Our proposed Hestenes-Jacobi architecture is implemented on Xilinx ZC706 evaluation board. The data in the architecture is represented in IEEE754 single precision floating point format. All the computation is done by utilizing the Xilinx Floating point generator [12]. The on-chip logic array is functioning at a clock rate of 150MHz. Input matrices and results are stored in the DRAM at the Cortex-A9 ARM core side. The data communication between off-chip logic and DRAM utilizes Xilinx direct memory access (DMA) IP [13].

As a demonstration of our SVD engine, three different designs with different degrees of parallelism are implemented. The first design has an array of 20 single PUs, where each PU has a local memory size of 0.5 Mb and is able to cache columns that have up to 4,096 entries. The second design has an array of 10 single PUs with the same single PU local memory size of 0.5 Mb. The third design has a vector PU array of 10 vector PUs, where each vector PU consists two single PUs. The local memory size of a vector PU is set to be 0.5 Mb.

The analysis of our proposed architecture is demonstrated in three aspects. First, resource utilization and power dissipation are reported. Second, randomized matrices of various dimensions are used to measure the execution time and speedup of the accelerators. Third, the MDS ordering is analyzed by comparing with round-robin ordering and ring ordering running on a same SVD engine.

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 6.1, approximately proportional to the number of single PUs in each design.

Table 6.1: Resource utilization and power dissipation of three designs.

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 Boost Analysis

Performance boost is measured by comparing the execution time of decomposing various random generated matrices by our designs running at 150 MHz and two SVD routines in Eigen library, the Jacobi algorithm and Bidiagonal-Divide-and-Conquer (BDC) algorithm, running on a Xeon E5-2680 CPU clocked at 2.8GHz. The execution times are shown in Table 6.2. Normalized execution times of Eigen solvers with respect to the 20 single PU design are shown in Figure 6.1. Our architecture can reach a speedup of up to 300X compared with the software solutions in Eigen package.

As a demonstration of row-based parallelism, the comparison of execution time of the three FPGA designs are shown in Figure 6.2. Noticeable that the 10 single PU design and the 10 vector PU design has the same amount of total BRAM utilized. However, by introducing the row-based parallelism, the 10 vector PU design is able to achieve 2X speedup compared with the 10 single PU design. Generally speaking, when the on-chip

BRAM resource serves as a limiting factor, i.e. the degree of column-based parallelism is limited, introducing the row-based parallelism can help achieve further speedup.

Table 6.2: Execution times (seconds) of 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

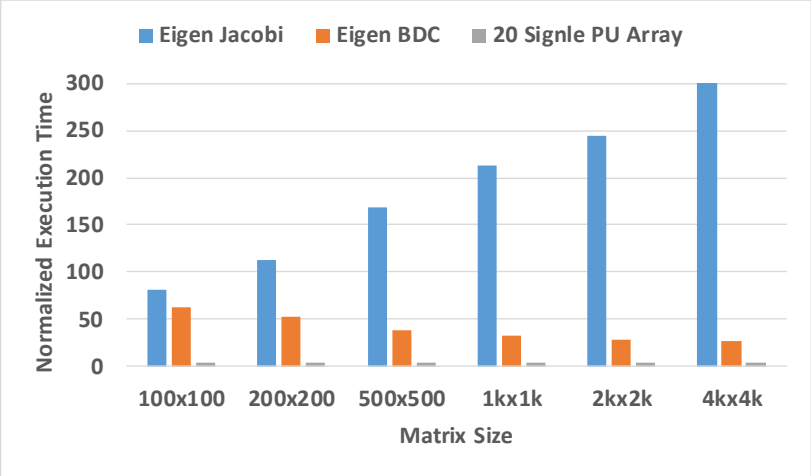

Figure 6.1: Normalized execution times of Eigen Jacobi & Eigen BDC @2.8GHz and 20 Single PU Array @150MHz.

To analyze the proposed MDS ordering, various randomized matrices are decomposed by our 20 single PU array design with round-robin ordering, ring ordering and the proposed MDS ordering. The execution times of three different data orderings are shown in Table 6.3. And the normalized execution times with respect to MDS ordering are shown in Figure 6.3.

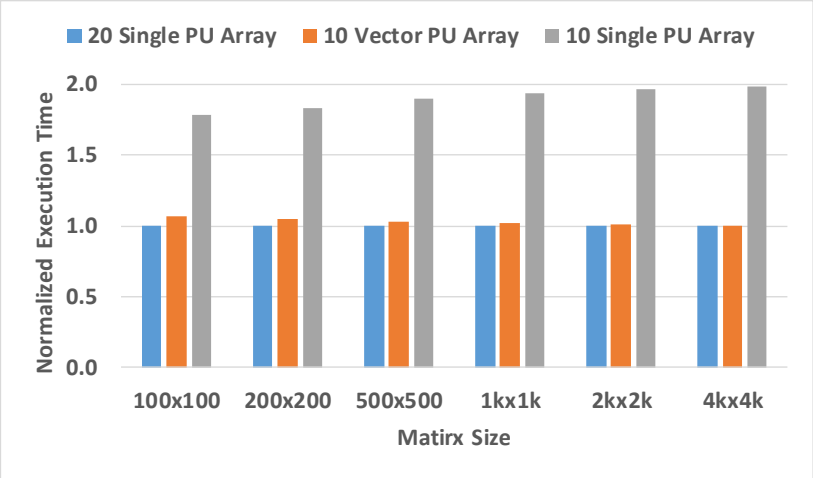

Figure 6.2: Normalized execution times of the 20 Single PU Array, 10 Vector PU Array and 10 Single PU Array all running at 150MHz on the FPGA.

Table 6.3: Execution time (seconds) of three different data ordering.

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

When the bandwidth is the limiting factor, the MDS ordering is able to achieve significant performance boost with a well utilization of on-chip data reuse. In the same situation for round-robin and ring ordering, the PUs have to compete for using the bus for communicate with off-chip memory and cannot achieve the desired maximum throughput. The MDS ordering allow us to implement more PUs and achieve higher level of parallelism with the same bandwidth available. Experiment shows the MDS ordering can achieve roughly 20X speedup over round robin ordering and approximately 10X speedup over ring ordering.

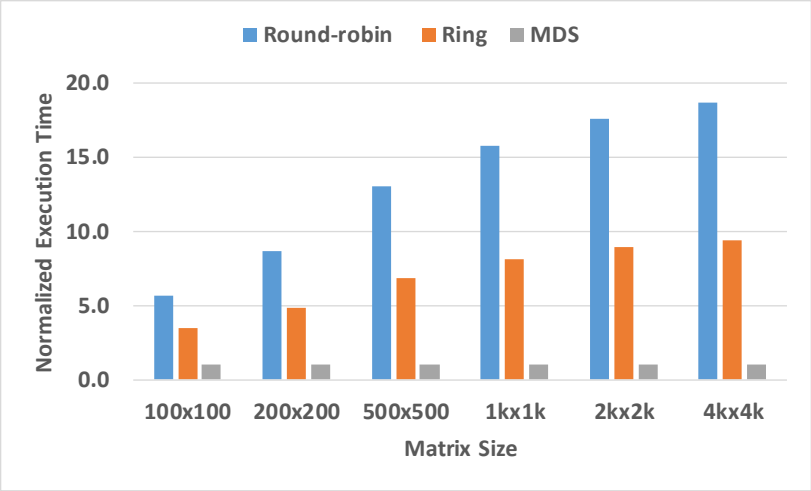

Figure 6.3: Normalized execution time of three orderings.

7. CONCLUSIONS AND FUTURE WORKS

7.1. Conclusion

In the dissertation, we first go through the emerging need for high performance heterogeneous computation. The importance and challenges in the current accelerator designs are discussed. We then use SVD, a popular computation intensive algorithm as a case study to analyze in this particular algorithm, how optimized data flow is able to reduce data transmission and bandwidth requirement.

After that, we propose a new MDS data ordering and implement the corresponding 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 provide higher degrees of parallelism for the Hestenes-Jacobi algorithm and can achieve better performance boost with limited bandwidth available. Compared with some existing SVD accelerators, our design is improved at the following aspects: first, the design provides a full solution for reasonable big matrices of arbitrary dimensions; second, the MDS ordering significantly reduces the bandwidth requirement and allows for higher parallelism; third, our proposed vector PU architecture with the row-based parallelism introduces an additional design freedom to well balance the logic and memory requirements.

7.2. Future Works

Although the dissertation provides a relatively thorough research on the Hestenes-Jacobi method and its related architectures, there exist limitations which can be further improved in future works.

The architecture proposed in the dissertation is limited by the local memory size of each PU. The accelerator in general, cannot process matrix whose columns (i.e. the size of which is larger than 4096 in the design) cannot be fully cached in a vector PU's local memory. In this case, row-based parallelism has to be introduced which results in additional requirement for bandwidth. Currently, the implementation is a fixed array and cannot be reconfigured to balance the trade-off between maximum processing size, required bandwidth and the required speedup. Developing the accelerator at a higher perspective, i.e. auto reconfiguring and adaptation to various requirements, is an interesting topic in the future.

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, M. E. Kaye, D. M. Luke, R. Doraiswami, “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] S. Cuomo, L. Marcellino, G. Navarra, “A parallel implementation of the Hestenes-Jacobi-One-Sides method using GPU-CUDA”, 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)

[7] R. Monhanty, G. Anirudh, T. Pradhan, B. Kabi, “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.

[8] 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," Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig), 2011.

[9] 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.

[10] 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, pp. 358-366.

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

[12] 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

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

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

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

[16] Orly Alter, Patrick O. Brown, and David Botstein, “Singular value decomposition for genome-wide expression data processing and modeling,” PNAS. 97 (18): 10101–10106.

[17] 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, April15, 2016.

[18] Jacobi, C.G.J. “"Über ein leichtes Verfahren, die in der Theorie der Säkularstörungen vorkommenden Gleichungen numerisch aufzulösen".”, *Crelle's Journal* (in German). 30 (30): 51–94.