2016

Using reconfigurable computing technology to accelerate matrix decomposition and applications

Xinying Wang
Iowa State University

Using reconfigurable computing technology to accelerate matrix decomposition and applications

by

Xinying Wang

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Program of Study Committee:

Joseph A. Zambreno, Major Professor

Phillip H. Jones

Tien N. Nguyen

Diane T. Rover

Zhao Zhang

Iowa State University

Ames, Iowa

2016

DEDICATION

I would like to dedicate this dissertation to my family for their continuous encouragement and financial assistance in my education. I would also like to thank my friends for their help and support during the writing of this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my gratitude to those who helped me with various aspects of conducting research and the writing of this dissertation. First, thanks to my advisor, Dr. Joseph Zambreno for his vision that lead me to this research. His guidance, patience, support and encouragement provide me the ability to develop research projects, and inspired me for completing my graduate education. His instructions benefit me both in my research and professional development. Thanks also to Dr. Phillip Jones for his help in my research and paper writing. I also appreciate Dr. Diane Rover, Dr. Zhao Zhang, Dr. Tien Nguyen for serving as my committee members; their comments and contributions are of great significance for improving this work.

I would also like to express my appreciation to my family, especially my parents, and all my friends in Ames or around the world in helping me pursue the degree.

ABSTRACT

Matrix decomposition plays an increasingly significant role in many scientific and engineering applications. Among numerous techniques, Singular Value Decomposition (SVD) and Eigenvalue Decomposition (EVD) are widely used as factorization tools to perform Principal Component Analysis (PCA) for dimensionality reduction and pattern recognition in image processing, text mining and wireless communications, while QR Decomposition (QRD) and sparse LU Decomposition (LUD) are employed to solve the dense or sparse linear system of equations in bioinformatics, power system and computer vision. Matrix decompositions are computationally expensive and their sequential implementations often fail to meet the requirements of many time-sensitive applications.

The emergence of reconfigurable computing has provided a flexible and low-cost opportunity to pursue high-performance parallel designs, and the use of FPGAs has shown promise in accelerating this class of computation. In this research, we have proposed and implemented several highly parallel FPGA-based architectures to accelerate matrix decompositions and their applications in data mining and signal processing. Specifically, in this dissertation we describe the following contributions:

- We propose an efficient FPGA-based double-precision floating-point architecture for EVD, which can efficiently analyze large-scale matrices.
- We implement a floating-point Hestenes-Jacobi architecture for SVD, which is capable of analyzing arbitrary sized matrices.
- We introduce a novel deeply pipelined reconfigurable architecture for QRD, which can be dynamically configured to perform either Householder transformation or Givens rotation in a manner that takes advantage of the strengths of each.

- We design a configurable architecture for sparse LUD that supports both symmetric and asymmetric sparse matrices with arbitrary sparsity patterns.
- By further extending the proposed hardware solution for SVD, we parallelize a popular text mining tool—Latent Semantic Indexing with an FPGA-based architecture.
- We present a configurable architecture to accelerate Homotopy ℓ_1-minimization, in which the modification of the proposed FPGA architecture for sparse LUD is used at its core to parallelize both Cholesky Decomposition and rank-1 update.

Our experimental results using an FPGA-based acceleration system indicate the efficiency of our proposed novel architectures, with application and dimension-dependent speedups over an optimized software implementation that range from 1.5× to 43.6× in terms of computation time.

CHAPTER 1. INTRODUCTION

1.1 Problem: matrix decomposition

Matrix decomposition refers to a class of linear algebraic operations that are widely used at the core of many scientific and engineering applications. By representing the data points in the form of matrix, many applications use matrix decomposition as an advanced method to extract interesting information. The use of matrix decompositions typically involves that analyze data pattern and solve linear system of equations. For example, Singular Value Decomposition (SVD) and Eigenvalue Decomposition (EVD) are popular factorization tools to perform pattern recognition and dimensionality reduction in the application of data mining such as Latent Semantic Indexing for text analysis, while QR Decomposition and LU Decomposition are used to solve dense or sparse system of linear equations in signal processing such as ℓ_1-minimization for robust face recognition. However, matrix decompositions are either operated in an iterative manner or iteratively performed in an application. Thus, they are considered as computationally expensive processes and their inherent computational complexity challenges them to satisfy the performance requirements for time-sensitive designs. For example, in the application of image recovery for video surveillance [Candès et al. (2011)], it takes over 100 seconds to recover an image represented as a matrix with the dimensions of 3000 by using SVD technique. As data dimensionality is increasing continuously, the runtime of matrix decomposition is likely to keep growing significantly, which indicates the necessary of exploring solutions to accelerate matrix decomposition.

1.2 Solution: FPGA and reconfigurable computing technology

To improve the performance, parallel solutions for matrix decomposition were proposed by using highly parallel accelerators such as Graphic Processing Unit (GPUs) and multi-core platforms. However, the multi-dimensional threading structure of GPU computing or multi-core platform is not strongly compatible with the highly data-dependent transformations or rotations of matrix decomposition. In practice, the performance improvement of GPU and multi-core platform-based matrix decomposition implementations is limited due to the iterative thread synchronization and irregular memory access.

Modern FPGAs are highly parallel and specialized computational fabrics, and they provide flexible and low-cost opportunities to pursue high-performance implementations of computational and memory intensive applications. Previous FPGA-based implementations have shown the capability of improving the performance for many applications such as linear algebra [Wen et al. (2010)] and graphic computation [Nurvitadhi et al. (2014)]. Compared to other parallel platforms, FPGAs have shown strong ability in providing better solution for performance improvement by parallelizing the matrix decomposition at the granularity of operand level with flexibility, reconfigurability and low energy consumption [Brent and Luk (1982); Wang and Leeser (2009); Kapre and DeHon (2009)].

1.3 Contributions: FPGA-based accelerators for matrix decomposition and applications

Previous FPGA-based implementations have looked at SVD [Brent and Luk (1982)], QRD [Wang and Leeser (2009)] and sparse LUD [Kapre and DeHon (2009)]. However, those approaches all have some limitations in common: either restricted with the scalability of the adapted matrices due to the logic capacity of FPGAs [Brent and Luk (1982); Ahmedsaid et al. (2003); Ma et al. (2006); Ledesma-Carrillo et al. (2011); Wang and Leeser (2009)] or required the input matrices of special property or irregular sparsity structure [Rafique et al. (2012); Tai et al. (2011); Vachranukunkiet (2007); Kapre and DeHon (2009); Wu et al. (2012)].

In this dissertation, we have designed and implemented efficient FPGA architectures for ma-

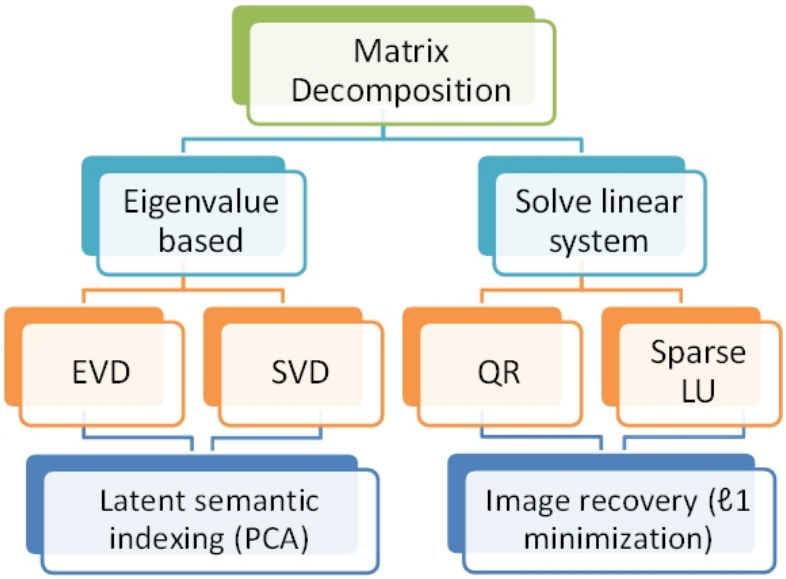


Figure 1.1: The research outline of this dissertation.

trix decompositions and applications to further explore the FPGA's potential in providing high performance solution. To analyze the representatives of Eigenvalue-based matrix decomposition with FPGA, EVD and SVD are selected as the candidates for performance improvement, in which EVD analyze squared symmetric matrices, while SVD operates datasets with arbitrary patterns. In addition, hardware solutions for QRD and sparse LUD are also proposed as both of them are the most popular factorization tools to solve linear system of equation. In this dissertation, the proposed FPGA architectures for SVD and LUD are also extended to accelerate the applications of Latent Semantic Indexing (LSI) and ℓ_1-norm minimization problem. Figure 1.1 demonstrates the research outline of this dissertation, all of which are projects with either separate paper publications in conference proceedings or under submission to Journals or conferences. The contributions of this dissertation are itemized as below, and they will be explained in detail in the later chapters:

- **An efficient architecture for floating-point Eigenvalue Decomposition** [Wang and Zambreno (2014a)]: this chapter presents a novel and efficient FPGA-based architecture for Eigenvalue Decomposition, which attempts to analyze considerably larger matrices than those applied to previous hardware designs, using matrix partition and a pipelined 1D systolic array. Experimental results demonstrate the better efficiency of our system compared to optimized CPU-based software solutions, a recent FPGA design for large matrices, and a GPU-based implementation when the matrix size is under 2000×2000.

- **An FPGA implementation of the Hestenes-Jacobi algorithm for Singular Value Decomposition** [Wang and Zambreno (2014b)]: this chapter proposes an FPGA-based hardware design of the Hestenes-Jacobi algorithm for Singular Value Decomposition with floating-point arithmetic, which attempts to analyze an arbitrary $m \times n$ matrix. Improved efficiency of this architecture compared to an optimized software-based SVD solution has been demonstrated for matrices with small to medium column dimensions, even with comparably large row dimensions.

- **A reconfigurable architecture for QR Decomposition using a hybrid approach** [Wang et al. (2014)]: this chapter introduces a hybrid approach that leverages the strengths of both Householder transformation and Givens rotation by applying the most appropriate algorithm of the two at each stage of the QR Decomposition. A reconfigurable architecture for QR Decomposition, which can be dynamically configured to perform either Householder transformation or Givens rotation, both of which are deeply pipelined. Experimental results demonstrates the achievement of speedups of this architecture compare to Intel Math Kernel Library (MKL), an FPGA-based tiled matrix decomposition, and a single threaded optimized software routine.

- **A configurable architecture for sparse LU Decomposition on matrices with arbitrary patterns** [Wang et al. (2016)]: this chapter proposes an FPGA-based architecture for sparse LU Decomposition, which can efficiently process sparse matrices with arbitrary sparsity patterns. This architecture factorizes columns from the lower triangular part of a matrices and rows from the upper triangular part of the matrix simultaneously, and in parallel with pivot operations. Performance improvement is demonstrated compared to an optimized software implementation for benchmarks containing a wide range of sparsity patterns.

- **Parallelize Latent Semantic Indexing using FPGA**: this chapter introduces an FPGA-based accelerator for Latent Semantic Indexing, which parallelizes the SVD calculation, the computing of cosine similarity between query and documents, and the process of ranking selected documents. This deeply pipelined architecture implements the extension of previous proposed Hastenes-Jacobi SVD architecture and an ordered tree structure. Evaluation of this design indicates the dimension-dependent performance improvements compared to optimized software implementation.

- **A configurable architecture to accelerate Homotopy ℓ_1-minimization**: this chapter implements an FPGA-based highly pipelined architecture for ℓ_1-norm minimization with Homotopy algorithm, which can be dynamically configured to perform either parallel Cholesky Decomposition or rank-1 update. Evaluation of this architecture demonstrates the improved efficiency compared to optimized software solution with both randomly generated datasets and the benchmark of a real-world face image database used.

CHAPTER 2. BACKGROUND

2.1 Matrix decomposition and applications

Matrix decompositions, such as Eigenvalue Decomposition (EVD), Singular Value Decomposition (SVD), QR Decomposition and LU Decomposition, have played vital roles in many scientific and engineering applications, especially in data mining and signal processing. For example, in data mining, Chen et al. (2010) proposed a Semisupervised Non-negative Matrix Factorization (SS-NMF) framework for data coclustering in text mining, information retrieval and bioinformatics. Compared to conventional data mining approach, matrix decomposition shows better capability in tackling the challenge brought by the fast-growing datasets. Joao et al. (2009) proposed a matrix decomposition method to improve the data mining process in telecommunications such as mobile subscriber classification and mobile network optimization. In signal processing, Harteneck and Stewart (1998) presented an adaptive solution for IIR filters, with which matrix decomposition is applied at its core. Studer et al. (2007) investigated matrix decomposition-based architectures and implementations for MIMO systems in wireless communication. However, the inherent computational complexity of matrix decompositions challenges them to satisfy the performance requirements for time-sensitive designs. In the following paragraphs, several most representative matrix decomposition and applications are introduced.

2.1.1 Eigenvalue Decomposition

Eigenvalue Decomposition (EVD) factorizes a symmetric matrix into a product of eigenvalues and eigenvectors, with which key patterns in the data can be identified. In many applications, EVD is used as an estimation technique to perform dimensionality reduction and

pattern recognition. In designing FIR filter bank for efficient subband coding, Redif et al. (2011) modified sequential best rotation (SBR2) algorithm, in which iterative, time-domain polynomial EVD is computed. In [Gabbay and Scott (2012)], the authors presented an EVD-based approach for Discrete Spectrum of Relaxation Frequencies (DSRF) in providing valuable classifying characteristic for buried objects detection and discrimination. However, in many applications, EVD is performed iteratively, which incurs a relatively high computational cost for the entire system. For example, the performance of distributed total least squares estimation in ad hoc sensor networks is dominated by EVD [Bertrand and Moonen (2011)].

2.1.2 Singular Value Decomposition

Singular Value Decomposition (SVD) is normally considered as an extension of EVD that can analyze arbitrary $m \times n$ rectangular matrix. Similar with EVD, SVD is widely used as a popular technique of Principal Component Analysis (PCA) to reduce the dimensions of data in high-dimensional spaces. In PCA, SVD generated singular values and singular vectors are used to approximate original datasets with fewer dimensions without losing significant information. To demonstrate the applications of SVD, Liao et al. (2012) introduced an SVD-based data mining framework to recognize the pattern of partial discharge, the underlying cause of an electrical equipment failure, while the authors of [Mu et al. (2011)] discussed the optimization of SVD usage in accelerating the exact recovery from visual data with corrupted components. However, SVD is a computationally-expensive procedure, and in the video surveillance application of [Candès et al. (2011)], it takes 185.2 seconds to recover the square matrix with the dimensions of 3000 through running partial SVD 15 times, which makes it difficult to satisfy stringent real-time performance requirements.

2.1.3 QR Decomposition

QR Decomposition transforms a matrix into a product of an orthogonal matrix Q and a upper triangular matrix R, on which normally backward substitution is performed to solve a linear equation or compute a matrix inverse. Xue et al. (2013) proposed an QR Decomposition-based method to calculate equalizer matrix for Multiple-Input Multiple-Output (MIMO) Orthogonal

Frequency Division Multiplexing (OFDM) technology used in today's wireless communication system. In CDMA, Liu and McCanny (2003) solved recursive least square problem by using QR Decomposition with proper backward substitution. Although QR Decomposition shows its popularity and efficiency in matrix computation, it still requires $O(n^3)$ operations. Previous research has shown that more than 10 minutes could be taken to perform QR Decomposition-based robust PCA on a $110,592 \times 100$ matrix, which is far beyond the requirements of potential real-time applications such as video surveillance or traffic monitoring [Anderson et al. (2011)].

2.1.4 LU Decomposition

The LU Decomposition on a matrix A produces a lower triangular matrix L and an upper triangular matrix U, whose product is identical to the matrix A. Similar with QR Decomposition, LU Decomposition is of great interest to scientists and engineers in solving dense linear system of equations. Besides, it has also demonstrated its wide usage in sparse linear algebra based applications. For example, Ren et al. (2012) explored the parallelization of sparse LU Decomposition for SPICE (Simulation Program with Integrated Circuit Emphasis) Simulation, which process operates unsymmetric and highly sparse matrices with irregular nonzero pattern. Cunningham et al. (2011) investigated the acceleration strategy for sparse LU Decomposition used for power flow computation and analysis. Due to the requirement of performing LU Decomposition iteratively in many system, and the inherent complexity of LU Decomposition, both sparse and dense LU Decomposition has become a performance dominator in many practical applications, such as image processing [Yang et al. (2013)], circuit simulation [Ren et al. (2012)], and power flow system [Cunningham et al. (2011)].

2.2 FPGA and reconfigurable computing technology

FPGA, which stands for Field Programmable Gate Array, is an integrated circuit contains array of configurable logic blocks (CLB) connected via programmable interconnects, and a configurable logic block (CLB) can be programmed individually to perform a unique function with combinational or sequential logic [Hauck and DeHon (2007)]. On an FPGA, a finite number of logic and memory resources are equipped including programmable look-up tables (LUTs),

flip-flops, multiplexers, dedicated DSPs and block RAMs. FPGA is widely used for ASIC (Application Specific Integrated Circuit) prototyping or random, reconfigurable, special-purpose hardware implementation in various application domains from automotive to aerospace, and from consumer electronics to data processing [Kuon et al. (2008)]. FPGA is featured by its computational characteristic, reconfigurability, flexibility, fine-grained, and energy efficiency, which make it as a candidate of hardware accelerator receive increasingly significant attention from high performance computing community. However, FPGAs are often disadvantaged by limited I/O bandwidth, and the performance can be further significantly improved as the memory bandwidth is scaled to the demands of applications [Zhang et al. (2009)]. In this dissertation, proposed solutions are evaluated on Convey system, an FPGA-based hybrid-core computing platform with I/O bandwidth optimized [Brewer (2010)]. The detail overview of high performance computing processors and convey hybrid-core computing platform can be referred in Appendix A.

2.2.1 Computational characteristic

FPGAs are able to improve the application performance in terms of speed by largely exploring the parallelism. By massively using hardware resources, parallel computations can be achieved through concurrent operations with extensively scheduled if the datasets have few or no data dependencies. Besides, in FPGAs, pipelining is implemented to improve the clock rate at the cost of latencies, and it offers parallelism with operations overlapped in time. With simple control requirement, FPGA applications are usually able to achieve 50 to 100 times the operations per clock cycle of a microprocessor, and their speedups over microprocessor implementations are determined by the amount of exploitable parallelism [Hauck and DeHon (2007)].

2.2.2 Flexibility

Compared to other programmable technologies such as microprocessor and DSPs, FPGAs provide a rich set of implementation alternatives instead of being given a fixed hardware architecture, and they are customizable and reprogrammable to optimize the implementations with

the best use of the devices [Hauck and DeHon (2007)]. With the help of programmable logic, FPGAs are able to flexibly configure logic and memory resources in many different ways in terms of combination and connection. By leveraging the flexibility of FPGAs, implementations can be optimized according to the application-specific performance demands with necessary trade-offs.

2.2.3 Reconfigurability

The hardware architecture of FPGAs can be configured any number of time with the strategies of either configure-once or run-time reconfiguration [Hauck and DeHon (2007)]. Configure-once performs a single system-wide architecture configuration prior to the application execution, and the configuration will be not altered until the application completes. Run-time reconfiguration dynamically reconfigure the hardware to perform different functions that are needed by the application, and it provides a great opportunity to perform numerous algorithms at different phase of execution with the same hardware resources [Ritala et al. (2000)].

2.2.4 Fine-grained

In today's FPGAs, one or more 4-LUTs or 6-LUTs are grouped into a single logic block. The logic blocks made up by small look-up tables possess fine-grained computational capability, which more suitable for operations with bit-level manipulations and arithmetic. With the fine-grained structure, FPGAs are easily tailored to application-specific computational requirements [Hauck and DeHon (2007)].

2.2.5 Energy efficiency

Although compared to ASICs, FPGAs consume more power due to the additional transistors are required by the programmability [Hauck and DeHon (2007)], FPGAs are demonstrated to outperform hardware alternatives (CPUs, GPUs, DSPs and etc.) in energy consumption for compute-intensive applications [Che et al. (2008); Kestur et al. (2010)]. In many applications, FPGA-based implementations show best energy efficiency in terms of performance per Watt compared to CPU, GPU, Multicores and ASIC solutions [Fowers et al. (2012); Hamada et al. (2009); Betkaoui et al. (2010)].

CHAPTER 3. AN EFFICIENT ARCHITECTURE FOR FLOATING-POINT EIGENVALUE DECOMPOSITION

Modified from a paper published in

Proceedings of 2014 *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*

Xinying Wang[1] and Joseph Zambreno[2]

3.1 Abstract

Eigenvalue Decomposition (EVD) is a widely-used factorization tool to perform principal component analysis, and has been employed for dimensionality reduction and pattern recognition in many scientific and engineering applications, such as image processing, text mining and wireless communications. EVD is considered computationally expensive, and as software implementations have not been able to meet the performance requirements of many real-time applications, the use of reconfigurable computing technology has shown promise in accelerating this type of computation. In this paper, we present an efficient FPGA-based double-precision floating-point architecture for EVD, which can efficiently analyze large-scale matrices. Our experimental results using an FPGA-based hybrid acceleration system indicate the efficiency of our novel array architecture, with dimension-dependent speedups over an optimized software implementation that range from 1.5× to 15.45× in terms of computation time. In addition, we also demonstrate the convergence performance of our EVD processing system.

[1]Primary researcher and author
[2]Correspondence author

3.2 Introduction

Eigenvalue Decomposition (EVD) has been widely used as a factorization tool to conduct principal component analysis in many scientific and engineering applications, such as image processing, acoustic processing, mobile communication and remote sensing [Redif et al. (2011); Gabbay and Scott (2012); Bertrand and Moonen (2011)]. To minimize the "dimensionality curse", which refers to the difficulties in managing and analyzing high-dimensional data, EVD can be employed to identify key patterns in the data, after which the original datasets can be approximated with fewer dimensions without losing significant information. In many signal processing applications, EVD is performed iteratively, which incurs a relatively high computational cost for the entire system. For example, distributed total least squares estimation in ad hoc sensor networks is dominated by EVD [Bertrand and Moonen (2011)]. As data dimensionality is continuing to increase in scientific and engineering applications, EVD runtime is likely to keep pace.

Eigenvalue Decomposition is characterized as the process of orthogonal transformations to diagonalize symmetric matrices, in which large amounts of highly data-dependent rotations are performed iteratively. Efficient software implementations such as MATLAB and LAPACK employ the Householder transformation [Golub and Kahan (1965)] to diagonalize matrices, which consists of recursive bidiagonalization process and implicit QR Decompositions; however, the high data dependency and inherent computational complexity of $O(n^3)$ restrict its performance, especially for applications involving large-scale matrices. The recent emergence of Graphic Processing Units (GPUs) in the high performance computing community has allowed for new methods to accelerate many general-purpose computations. However, the multi-dimensional threading structure of GPU computing is not highly compatible with the iterative thread synchronization and irregular memory access required for better EVD convergence making the optimization of these designs on GPUs quite challenging, especially for input matrices with dimensions smaller than 1000 [Lahabar and Narayanan (2009); Kotas and Barhen (2011)].

Modern FPGAs are highly parallel and specialized computational fabrics, and previously researchers have investigated accelerating both EVD and Singular Value Decomposition (SVD)

using FPGAs [Brent et al. (1985); Hestenes (1958)]. However, similar to other FPGA-based designs for matrix decomposition [Tai et al. (2011); Wu et al. (2012)], the logic capacity of FPGAs has typically limited the scalability of the adapted matrices [Brent and Luk (1982); Ahmedsaid et al. (2003); Ma et al. (2006); Ledesma-Carrillo et al. (2011)], even though this previous work targeted applications in real-time signal processing using fixed-point arithmetic, for which hardware resource utilization is significantly less than for floating-point arithmetic.

In this chapter, we present a novel and efficient FPGA-based architecture for Eigenvalue Decomposition, which attempts to analyze considerably larger matrices than those applied to previous hardware designs, using matrix partition and a pipelined 1D systolic array. Our single FPGA-based design supports double precision float-point operands [Microprocessor Standards Committee of the IEEE Computer Society (2008)], offering a wider dynamic range than previous fixed-point implementations. Our experimental results demonstrate the better efficiency of our system compared to optimized CPU-based software solutions, the latest FPGA design for large matrices [Ledesma-Carrillo et al. (2011)], and a GPU-based implementation when the matrix size is under 2000×2000 [Lahabar and Narayanan (2009)].

3.3 Theoretical background

3.3.1 Singular Value /Eigenvalue Decomposition (SVD/EVD)

The Singular Value Decomposition of an $m \times n$ matrix A is in the form of eq. (3.1)

$$A = U\Sigma V' \tag{3.1}$$

where U is an $m \times m$ matrix and V is an $n \times n$ matrix, both of which are orthogonal matrices such that $U' \cdot U = V' \cdot V = I$. Σ is an $m \times n$ diagonal matrix with the nonnegative diagonal elements, which are the singular values.

Factorization is called EVD when A is a squared symmetric matrix with U being identical to V, both of which are orthogonal matrices named as eigenvectors, while Σ is an $n \times n$ diagonal matrix with eigenvalues as diagonal elements. Diagonalizing rotation algorithms are popular in performing Singular Value Decomposition or solving eigenvalue problems.

3.3.2 Jacobi rotations

Jacobi rotations are performed iteratively for matrix diagonalization by using Jacobi rotation matrices J^l and J^r as shown in eq. (3.2) and eq. (3.3). J^l is identical to J^r when A_{pq} and A_{qp} have the same value.

$$A_{i+1} = J_i^l A_i J_i^r \tag{3.2}$$

$$J^{l'} \cdot \begin{pmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{pmatrix} \cdot J^r = \begin{pmatrix} A_{pp}^{"} & 0 \\ 0 & A_{qq}^{"} \end{pmatrix} \tag{3.3}$$

The Jacobi matrices J^l and J^r can be obtained in the forms of $J^l(p,q,\alpha)$ and $J^r(p,q,\beta)$ through the determinations of plain rotation angles with paired diagonal elements and their respective off-diagonal elements, as shown in eq. (3.4) and eq. (3.5), where θ represents plain rotation angles α or β [Brent et al. (1985)].

$$\begin{cases} J_{pp} = cos(\theta); & \\ J_{pq} = sin(\theta); & (p < q) \\ J_{qp} = \text{-}sin(\theta); & (p < q) \\ J_{qq} = cos(\theta); & \\ J_{ii} = 1; & (i \neq p,q) \\ J_{ij} = 0, & Others. \end{cases} \tag{3.4}$$

$$\beta+\alpha = \arctan\left(\frac{A_{qp} + A_{pq}}{A_{qq} - A_{pp}}\right) \quad \beta\text{-}\alpha = \arctan\left(\frac{A_{qp} - A_{pq}}{A_{qq} + A_{pp}}\right) \tag{3.5}$$

The Jacobi rotation approach provides opportunities for pursuing parallelism by simultaneously computing a group of independent 2×2 Jacobi rotation matrices, whose calculations only affect two rows and columns of a matrix.

3.4 Related Work

Solutions for Eigenvalue Decomposition and Singular Value Decomposition can be categorized into two types: the Householder bidiagonalization with implicit QR [Golub and Ka-

han (1965); Golub and Reinsch (1970)] and the Jacobi related approach [Brent et al. (1985); Hestenes (1958)]; among which, the Householder approach has been employed by many standard software implementations (e.g., MATLAB, LAPACK) due to its efficiency in sequential programming. However, its inherent high data dependency poses challenges for further parallelism. The GPU-based implementations for the Householder approach were evaluated, in which possible acceleration was demonstrated only for matrices with significantly large sizes ($\gg 1000 \times 1000$) [Lahabar and Narayanan (2009); Kotas and Barhen (2011)].

Jacobi-related approaches, including the two-sided Jacobi Rotation algorithm [Brent et al. (1985)] and the one-sided Jacobi Rotation algorithm [Hestenes (1958)], provide an opportunity for better parallelism. To parallelize the computations through implementing Jacobi approach on multi-core platforms, block-Jacobi algorithms were proposed to accelerate EVD and SVD [Bečka et al. (2012)]; the experimental results show comparably poor performance of the implementation on a multi-processor platform due to the overhead of inter-processor communication, even when using a modern massive parallel supercomputer with 1024 cores.

FPGAs have demonstrated the capability to parallelize numerous algorithms at the operand-level granularity. Previously, FPGAs were employed to demonstrate the highly parallel implementations of EVD and SVD based on two-sided Jacobi Rotations, by accelerating their independent 2×2 rotations, using a parallel architecture featuring a 2-dimensional systolic array. In this earlier work, the scalability of the applicable matrices had been severely restricted by the limited resources on FPGAs [Brent and Luk (1982); Brent et al. (1985); Ahmedsaid et al. (2003); Ma et al. (2006)]. In [Brent and Luk (1982); Brent et al. (1985)], the authors demonstrated the efficiency of the 2D systolic array designs for EVD and SVD with the time complexity of $O(n \log n)$ for an n-by-n square matrix, in which $\log n$ was proved as the number of iterations for reasonable convergence with certain threshold by applying parallel Jacobi rotation or cyclic Jacobi rotation methods; meanwhile, a number of n^2 processing units (PEs) are needed. The Hestenes-Jacobi Method [Strumpen et al. (2003)], which is also known as one-sided Jacobi rotation, provides a better opportunity for vectorized parallel operations. However, its architectural design with iterative and repetitive processing limited the overall speedup [Kotas and Barhen (2011)], while GPU implementations have suffered from the overhead associated

with thread synchronization and global memory reads [Ledesma-Carrillo et al. (2011)]. While the aforementioned FPGA-based designs were all calculated with fixed-point datasets, it is important to note that floating-point operands are significantly more popular in scientific and engineering applications given the wider range of data representation [Microprocessor Standards Committee of the IEEE Computer Society (2008)].

3.5 The partitioned EVD computation algorithm

Our partitioned EVD computation algorithm (Fig. 3.1) was derived from the two-sided Jacobi approach [Brent et al. (1985)] to zero out all the off-diagonal elements iteratively. The rotations for a symmetric matrix are identical on both sides, whose computations can be reduced by half as the processing on a lower or upper triangular matrix. To improve the scalability of the design, the matrix is first partitioned into a series of vector-blocks and then followed by recursively rotating diagonal elements to annihilate off-diagonals. Each partition consists of numerical diagonal elements and their respective rows and columns in the lower or upper triangular part of the matrix. A partition example is shown in Fig. 3.2, and vector-block partitions related to the diagonal elements of $A_{1,1}$-$A_{4,4}$ and $A_{5,5}$-$A_{n,n}$ are highlighted by the polygons with solid and dashed lines, respectively. In this paper, for each partitioned vector-block, the diagonal elements and off-diagonal elements in this partition are referred to as *host diagonal* elements and *host off-diagonal* elements respectively, while the remainder of diagonal elements and off-diagonal elements in the matrix are referred to as *guest diagonal* elements and *guest off-diagonal* elements respectively. For example, considering the solid lines highlighted partition in Fig. 3.2, $A_{1,1}$-$A_{4,4}$ are host diagonal elements and $A_{5,5}$-$A_{8,8}$ are guest diagonal elements, while the off-diagonal elements in the first four columns are host off-diagonal elements and the rest of off-diagonals are guest off-diagonal elements. At runtime, the partitioned vector-blocks are processed successively; in processing each partition, host diagonal elements are paired with every other diagonal elements of the matrix to perform Jacobi rotations to zero out all the host off-diagonal elements. Meanwhile, the updates of affected off-diagonal elements are calculated.

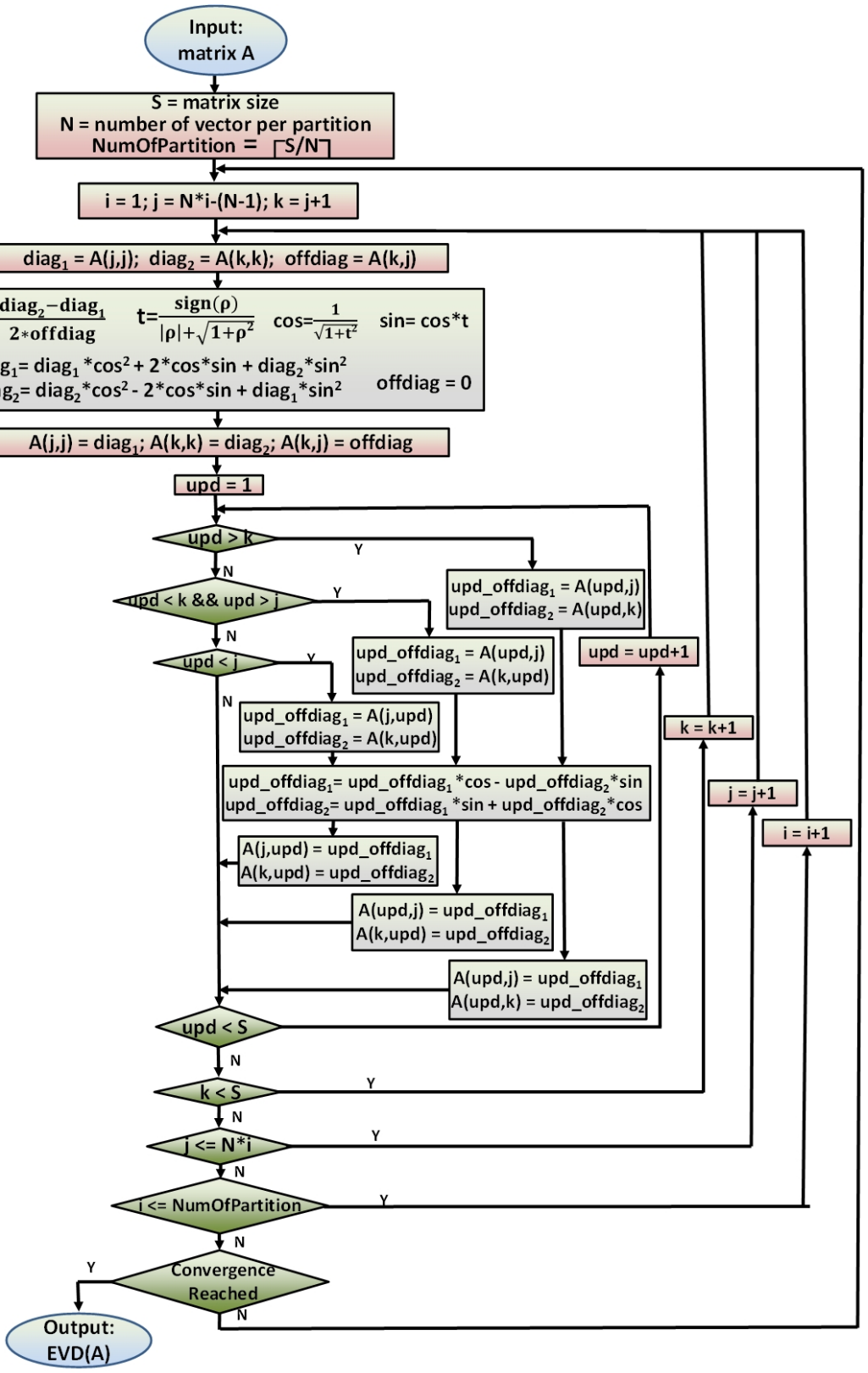


Figure 3.1: Our partitioned EVD computation algorithm.

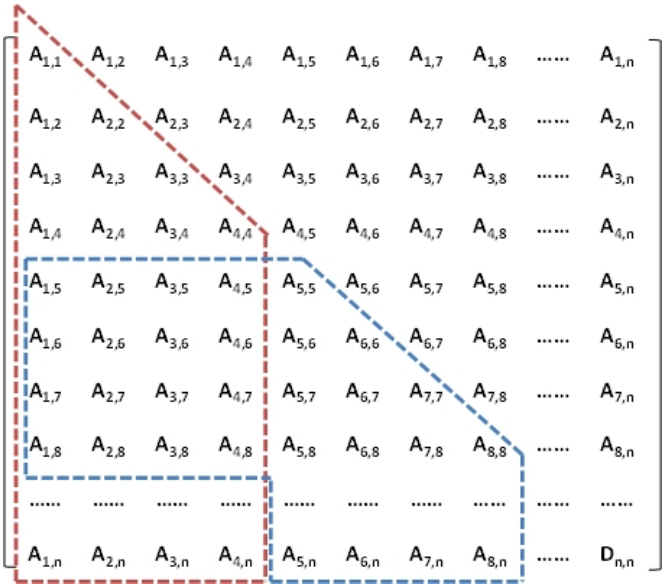


Figure 3.2: An example matrix partition for EVD.

3.6 The EVD architecture

To parallelize our partitioned EVD approach, each partition is mapped to a systolic array of computational processing elements (PEs). Figure 3.3 shows the example of mapping the partition, which is highlighted by polygon with the solid line in Fig. 3.2, to the systolic array. As shown in Fig. 3.3, this systolic array consists of three types of computational PEs: *diagonal Jacobi Rotation* elements, *off-diagonal single update* elements, *off-diagonal double update* elements, in which the diagonal PEs (shown as ovals in Fig. 3.3) are employed to conduct Jacobi Rotation, while the Off-diagonal Single Update elements and the Off-diagonal Double Update elements are used to update off-diagonal vectors affected by one or two rotations respectively (shown as rectangles in Fig. 3.3).

Rotation angle parameters *cos* and *sin* are generated by the diagonal PEs and then broadcast to the respective off-diagonal PEs, which are in the same rows and columns with diagonal PEs, to update the remaining elements. Off-diagonal PEs with two off-diagonal elements, are affected only by the rotations in the same row that a "single update" is needed over the two off-diagonal elements; on the other hand, off-diagonal PEs, with which four off-diagonal elements are included, have to update twice on different combinations of the two off-diagonal

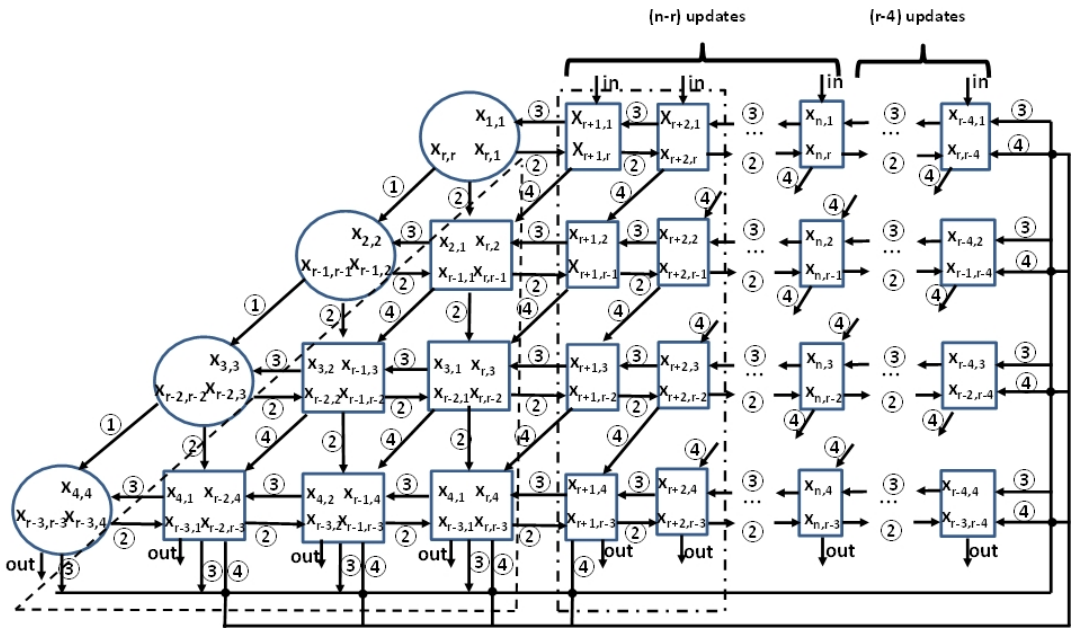


Figure 3.3: Demonstration of partitioned Jacobi rotation approach (for $r \geq 9$).

pairs as named "double update", since they are affected by the rotations both from the same rows and columns. Numerical diagonal PEs perform rotation simultaneously in parallel with the updates of their respective columns and rows that are operated in off-diagonal PEs. Values are transmitted along their dataflow paths once their calculations are completed. The general dataflow is demonstrated in Fig. 3.3 as arrows, in which the arrows labeled with ① and ② indicate the transmission of guest diagonal element and rotation angle parameter (*cos, sin*) respectively while the movements of host off-diagonal elements and guest off-diagonal elements are represented by the arrows labeled with ③ and ④ respectively. The host off-diagonal elements $x_{r+1,1}$, which move leftwards for rotations iteratively, continue to be updated while moving downward after being zeroed and then loop back to the end of the row when they have reached the bottom row of the PEs. The movement of guest off-diagonal elements $x_{r+1,r}$ follows their respective guest diagonal element $x_{r+1,r+1}$; guest off-diagonal elements loop back to the end of row when they are needed by subsequent updates.

To fit our design on a single chip, pipelined computational cores provide the opportunity to reuse the PEs with parallel calculations. One Jacobi Rotation PE is devised to perform all the Jacobi Rotations in a pipeline, while a series of pipelined off-diagonal single update PEs and one

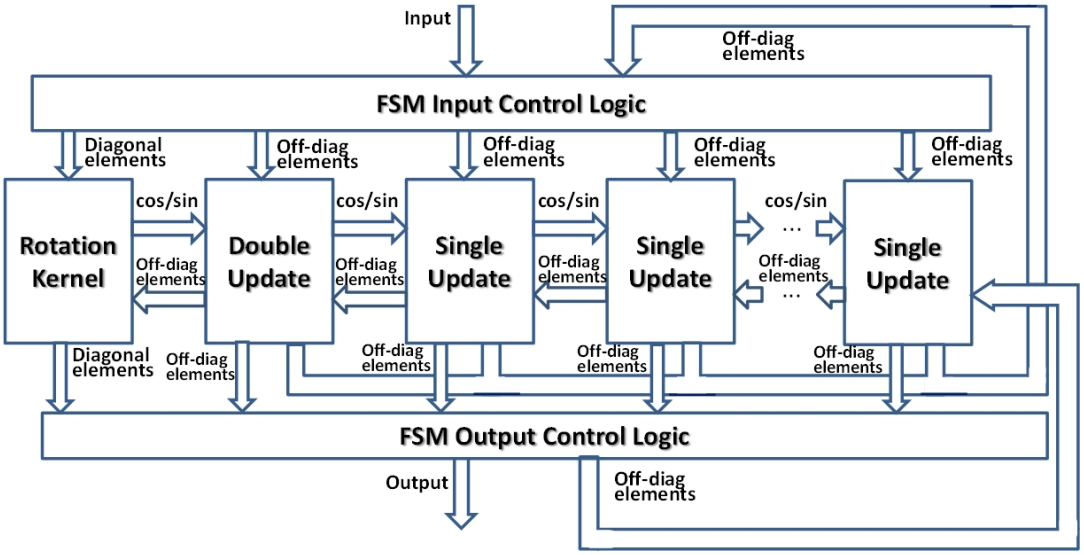


Figure 3.4: Block diagram of the general 1D systolic array architecture for EVD.

pipelined off-diagonal double update PE are used to simultaneously update groups of affected off-diagonal sub-matrices. Consequently, the architecture is converted into a one-dimensional systolic array as shown in Fig. 3.4 with the number of off-diagonal Single Update components determined by the dimension of the matrices and the resource capacity of the hardware.

3.6.1 Diagonal Jacobi rotation component

To zero out an off-diagonal element, Jacobi rotation is performed with its respective two diagonal elements in the same row or column. Jacobi Rotation can be performed through a series of addition, subtraction, multiplication, division and square root operations. Through expanding the rotation formulas, the rotation process is shown in Eqns. (3.6, 3.7), in which a_{pp}, a_{qq}, and a_{pq} represent two diagonal elements and an off-diagonal element respectively. Then, the process is optimized by shortening the latency and parallelizing the calculations.

$$cos = \sqrt{\frac{(a_{qq}-a_{pp})^2+2*a_{pq}^2+|a_{qq}-a_{pp}|*\sqrt{(a_{qq}-a_{pp})^2+4*a_{pq}^2}}{(a_{qq}-a_{pp})^2+4*a_{pq}^2+|a_{qq}-a_{pp}|*\sqrt{(a_{qq}-a_{pp})^2+4*a_{pq}^2}}} \tag{3.6}$$

$$sin = (sign)\sqrt{\frac{2*a_{pq}^2}{(a_{qq}-a_{pp})^2+4*a_{pq}^2+|a_{qq}-a_{pp}|*\sqrt{(a_{qq}-a_{pp})^2+4*a_{pq}^2}}} \tag{3.7}$$

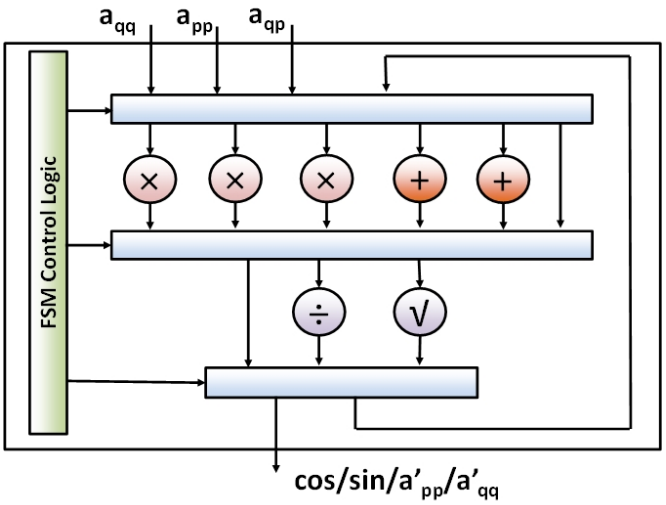


Figure 3.5: The architecture of Jacobi Rotation Component (eight operands can be applied every 64 clock cycles).

An efficient EVD design needs to perform Jacobi rotation and updates of affected rows and columns simultaneously; it is important to balance resource allocation between the Jacobi rotation component and all the updating modules, whose computational latency increases linearly with the growth of matrix dimension. Therefore, to balance the hardware resource consumption and efficiency, pipelined floating-point cores are shared by the calculations. An example Jacobi Rotation component architecture is demonstrated in Fig. 3.5, in which eight double-precision operands are streamed in every 64 clock cycles.

3.6.2 Off-diagonal single update component

The off-diagonal single update component is responsible for updating the off-diagonal elements, which are affected by one Jacobi Rotation each iteration. Although the updating process consists of simple multiplications and addition or subtraction as is shown in eq. 3.8 and eq. 3.9, it is infeasible to fit an arbitrary number of updating components on a single chip. In our design, floating-point computational cores are employed to process the updates of sub-matrices of host off-diagonal elements and guest off-diagonal elements in a pipeline, in which limited number of PEs can perform large-scale updates in parallel. Every time the sub-matrices are completed with their update, a vector of host off-diagonal elements will be sent leftwards to

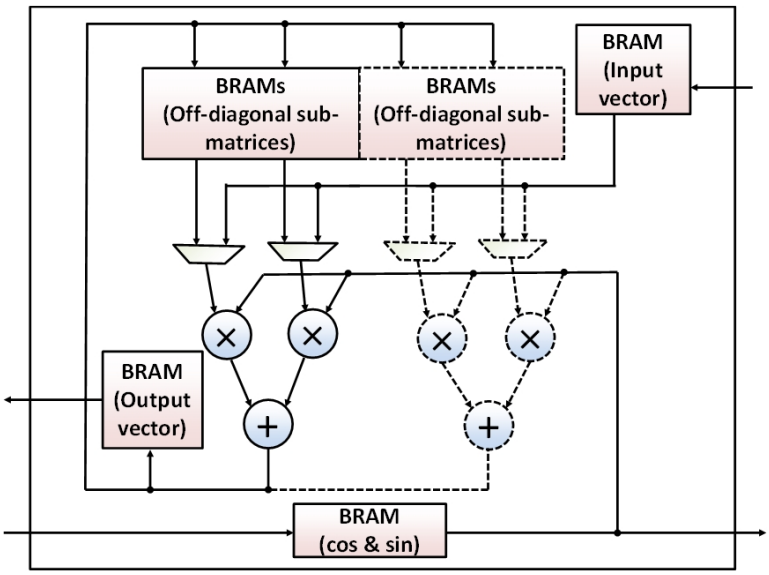


Figure 3.6: Update component architecture.

the next off-diagonal single update component, and a vector of guest off-diagonal elements will be transmitted to external memory or looped back as input of another off-diagonal single update component according to the request.The architecture of our off-diagonal single update component is shown by the solid lines in Fig. 3.6.

$$Offdiag'_{host} = Offdiag_{host} \times cos - Offdiag_{guest} \times sin \tag{3.8}$$

$$Offdiag'_{guest} = Offdiag_{host} \times sin + Offdiag_{guest} \times cos \tag{3.9}$$

3.6.3 Off-diagonal double update component

The off-diagonal double update component is responsible to update sub-matrices of off-diagonals, which have to be processed with two updates successively each time, since both of their respective columns and rows are involving with the rotations. To integrate all of the updates affected by two rotations into one component, local memories are used to hold four triangular sub-matrices of off-diagonal elements, and the number of computational cores is at least twice as many used in the off-diagonal single update component in order to synchronize

with the other components. An example off-diagonal double update component architecture is shown in Fig. 3.6 with both solid and dashed lines.

3.7 Experiments and evaluations

3.7.1 Implementation and experimental setup

To evaluate our design, we programmed our architecture on a single high-end Xilinx Virtex-5 FPGA (XC5VLX330) of the Convey HC-2 system [Convey Computer HC-2 (2012)]. In our implementation, we generated IEEE-754 double-precision floating-point calculators using the Xilinx Logic IP core generator [Xilinx Inc. (2012)]. In the diagonal Jacobi Rotation component, eight rotations can be initiated for every 64 clock cycles with double-precision calculators as one divider, one square root, two adders and three multipliers, among which adders and multipliers were configured to use dedicated multiplier circuitry (DSPs). In each off-diagonal single update component, IP core generated Block RAMs are used to hold the sub-matrices of off-diagonal elements and rotation angle parameters, while one double-precision floating-point adder and two double-precision floating-point multipliers were implemented by dedicated multiplier circuitry (DSPs) and logics respectively. The IP cores of adder, multiplier, divider and square root are configured with latency of 14, 9, 57 and 57 cycles respectively, among which some of the multiplier or adder may have different latency due to the use of DSPs.

3.7.2 Performance analysis

In our design, maximally, 32 off-diagonal updating components can be allocated on our target FPGA, in which 31 off-diagonal single update components and one off-diagonal double update component are included. By evaluating our design at the frequency of 100 Mhz with 6 iterations, which was believed sufficient for convergence on matrices with certain thresholds, the performance of our design has demonstrated dimensional-dependent speedups from 1.5× to 15.45× for moderate- to large-sized matrix compared to optimized Matlab 7.10.0 software SVD solution that was processed on a 2.2 GHz dual core Intel Xeon processor with 16 GB installed memory as shown in Table 3.1.

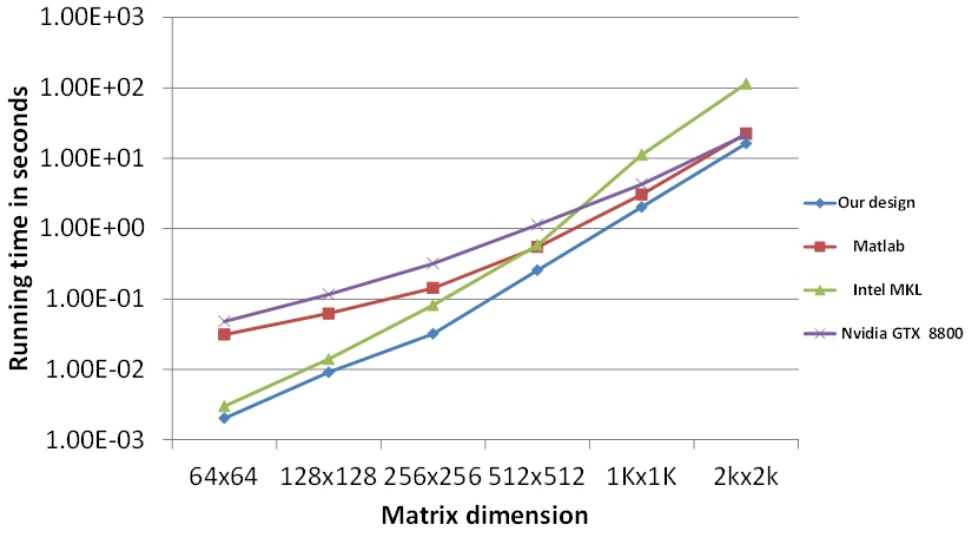


Figure 3.7: EVD/SVD computation time (in seconds) for symmetric matrix by our design, Intel MKL and GPU.

Fig. 3.7 demonstrates the quantitative comparison among dimensional dependent execution times of EVD processing by using different approaches. The blue line demonstrates the EVD performance by using our architecture while the performance of the Matlab 7.10.0 EVD routine running on the Intel platform is shown by the red line. The execution time of EVD/SVD solutions with Intel MLK 10.0.4 and NVIDIA 8800 GPU [Lahabar and Narayanan (2009)], both of which are using a 2.66 GHz Intel Core 2 Duo CPU, are depicted as green and purple lines respectively. By analyzing those data points in Fig. 3.7, although GPU-based solution has demonstrated better efficiency when matrix size grows over thousands, our design is more efficient for processing matrices up to 2000×2000.

In Fig. 3.8, the number of floating point operations that are performed in a single update component after every group of Jacobi rotations is demonstrated. The floating point operations can be conducted in pipelining with new operation started every clock cycle, and the black line shows the latency of Jacobi rotation, which is 357 clock cycles. Observations can be made that the update operations have become an increasingly significant factor to determine the entire performance as the dimension grows, especially when the update operations take more clock cycles than the latency of Jacobi rotation, the update operation becomes the performance

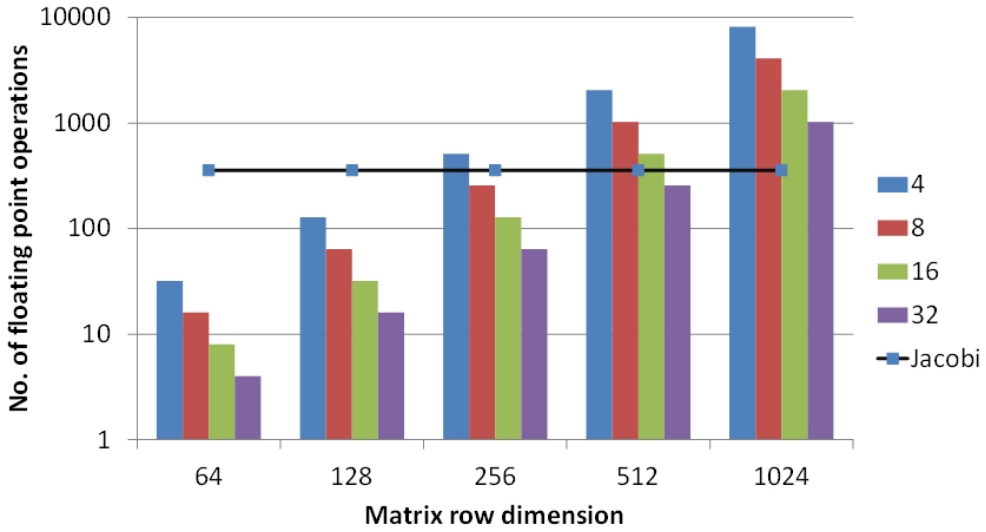


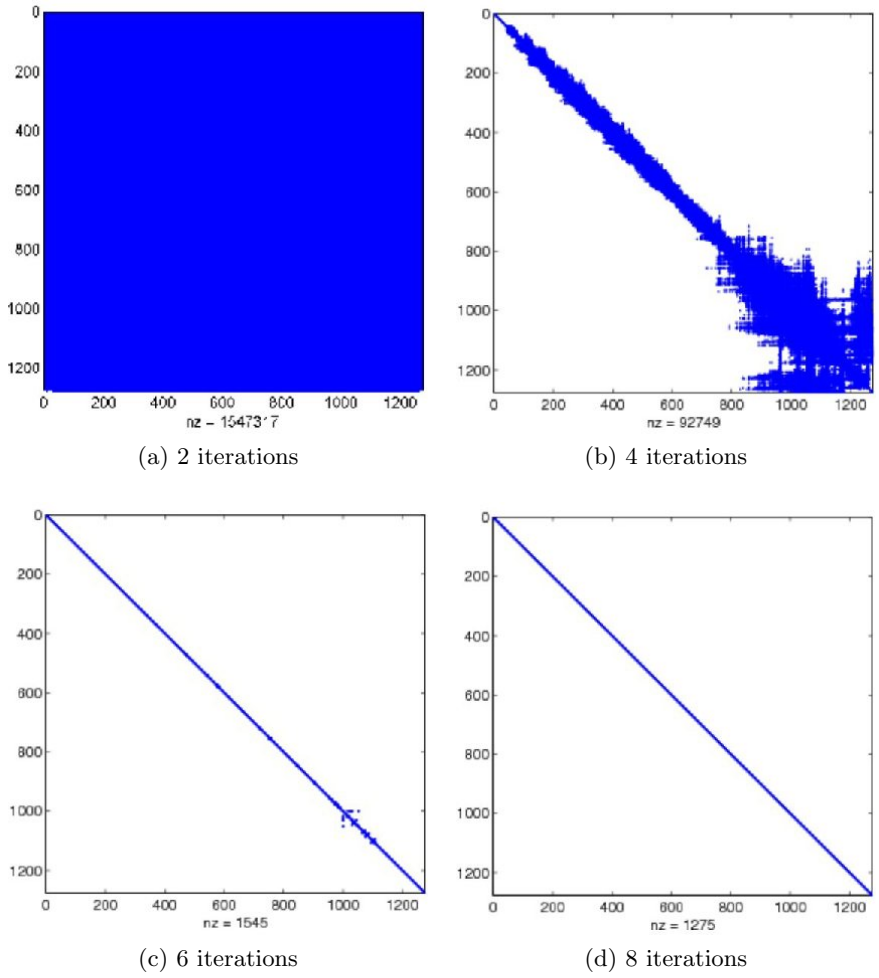
Figure 3.8: The No. of floating point update operations in a single update component after each rotation for matrices with various dimensions and partition stategies.

dominant.

Practically, the system performance of our design is dominated by the time consumption of rotations for small-scale applications; however, when the matrix size grows over a comparably large value such as 512 with certain partition strategy, updates consume more time than rotating, which incurs a performance degradation. Additionally, to the best of our knowledge, Ledesma-Carrillo et al. (2011) was the latest and only scalable architecture for FPGA-based EVD/SVD design; however, its performance suffered from the iterative design and low-capacity platform they employed, and these previous published results are slower than our results by two orders of magnitude with a matrix size limitation of 32×128.

Table 3.1: Proposed architecture performance in speed.

Dimension	Our Architecture	Matlab	Speedup
64×64	0.00202s	0.0312s	15.45×
128×128	0.0091s	0.0624s	6.4×
256×256	0.0320s	0.1428s	4.46×
512×512	0.2558s	0.5446s	2.2×
1024×1024	2.0290s	3.0607s	1.5×



(a) 2 iterations (b) 4 iterations

(c) 6 iterations (d) 8 iterations

Figure 3.9: Matrix sparsity after processing numeric iterations

3.7.3 Convergence analysis

To analyze the data convergence of our approach, we applied real values from the dataset of a published digital image research work [Qiu and Vaswani (2011)] using a 1275×1275 matrix, to the software model of our approach. By setting the threshold as 0.1, which was 10000 times less than the mean absolute value of the experimental data, the visualized sparsity of matrices after multiple iterations is shown in Fig. 3.9. We also generated random data with various dimensions and the mean variance from zero of the off-diagonals after being processed by different number of iterations are shown in Table 3.2. Both of them indicate reasonable convergence can be reached using our approach.

Table 3.2: Mean variance from zero of the off-diagonals after numeric iterations.

Dimension	Origional	2 iterations	4 iterations	6 iterations
64×64	8.02	8.45×10^{-5}	2.47×10^{-5}	1.16×10^{-6}
128×128	15.98	6.34×10^{-4}	1.56×10^{-4}	4.86×10^{-6}
256×256	31.99	1.78×10^{-4}	1.12×10^{-4}	4.26×10^{-7}
512×512	64.04	4.63×10^{-4}	6.39×10^{-5}	9.18×10^{-6}
1024×1024	128.04	7.39×10^{-3}	2.7×10^{-3}	8.1×10^{-5}
2048×2048	256.03	4.93×10^{-3}	3.4×10^{-3}	1.1×10^{-4}

3.8 Conclusion

An efficient reconfigurable FPGA-based hardware architecture is proposed to perform Eigenvalue Decomposition; which employed a novel modified Partitioned-Jacobi algorithm and a pipelined one dimensional systolic array. The analysis of our architecture demonstrates the scalability and dimensional dependent efficiency of our design. By applying both real values from a published digital image research work and randomly generated data to our design, convergence is demonstrated for our approach. Our proposed framework will be extended to perform principal component analysis for future work.

CHAPTER 4. AN FPGA IMPLEMENTATION OF THE HESTENES-JACOBI ALGORITHM FOR SINGULAR VALUE DECOMPOSITION

Modified from a paper published in

Proceedings of 2014 *IEEE International Parallel&Distributed Processing Symposium Workshops (IPDPSW)*

Xinying Wang[1] and Joseph Zambreno[2]

4.1 Abstract

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

[1]Primary researcher and author
[2]Correspondence author

large row dimensions. The dimensional speedups can be achieved range from 3.8× to 43.6× for matrices with column dimensions from 128 to 256 and row sizes from 128 to 2048. Additionally, we also evaluate the accuracy of our SVD process through convergence analysis.

4.2 Introduction

In many real-world applications, data dimensionality is rapidly growing. Principal Component Analysis (PCA) is widely employed to reduce the dimensions of data in high-dimensional spaces. Among the classical solutions for PCA, Singular Value Decomposition (SVD) is the most popular technique to approximate high-dimensional data through orthogonal transformations. SVD-based PCA has been used in many signal processing applications such as image processing, computer vision, pattern recognition and remote sensing [Xu et al. (2012); Mu et al. (2011); Liao et al. (2012)]. However, SVD is a computationally-expensive procedure, which makes its use unlikely to meet the requirements of many time-sensitive designs, especially when it is processed iteratively in those applications. For instance, in the application of video surveillance [Candès et al. (2011)], it takes 185.2 seconds to recover the square matrix with the dimensions of 3000 through running partial SVD 15 times, which makes it difficult to satisfy stringent real-time performance requirements. As data dimensionality is increasing continuously, the runtime of SVD is likely to have further substantial growth.

The SVD operation diagonalizes an arbitrary $m \times n$ matrix through a series of orthogonal transformations [Trefethen and Bau (1997)]. Optimized software implementations (e.g., MATLAB, LAPACK) employ the Householder transformation [Golub and Kahan (1965)] to perform SVD computation; however, their performance is restricted by their inherent computational complexity and high data dependency. Highly parallel accelerators such as Graphic Processing Unit (GPUs) and multi-core platforms have been employed to explore parallel implementations, although these previous works only achieved speedups when the input matrices have dimensions greater than 1000 [Lahabar and Narayanan (2009); Haidar et al. (2013)].

In the reconfigurable architecture community, systolic-arrays have been implemented on modern FPGAs to compute the classic two-sided Jacobi rotations [Brent et al. (1985)]. Although improved performance has been demonstrated, the scalability of those implementations

are often limited, and the designs are restricted to operate only on the input with squared matrices.

Hestenes [Hestenes (1958)] discovered the equivalence between zeroing out an off-diagonal a_{ij} and orthogonalizing the ith and jth vectors through plane rotation. Instead of annihilating every non-zero off-diagonal element by rotating 2×2 matrices, the Hestenes-Jacobi method is capable of decomposing an arbitrary $m \times n$ non-square matrix through vector computations. GPUs and FPGAs have been employed to evaluate parallel Hestenes-Jacobi designs; however, the performance has suffered from the iterative thread synchronizations (in the case of GPUs [Kotas and Barhen (2011)]) and repeated calculations (in the case of FPGA implementations [Ledesma-Carrillo et al. (2011)]).

In this paper, we present an FPGA-based hardware design of the Hestenes-Jacobi algorithm for SVD with floating-point arithmetic, which attempts to analyze an arbitrary $m \times n$ matrix. Compared to a previous FPGA-based Hestenes-Jacobi implementation [Ledesma-Carrillo et al. (2011)], our architecture optimizes the calculations through improving data reuse, and employs IEEE-754 double precision floating-point operators to provide a wider dynamic range. Also, off-chip memory is employed to break the restriction of the analyzable matrix dimensions. Our experimental results have demonstrated the efficiency of our design for matrices with small to medium column dimensions, even when they have comparably large row dimensions. The dimension-dependent speedups that can be achieved range from 3.8× to 43.6× for matrices with column sizes from 128 to 256 and row dimensions from 128 to 2048. Compared to other GPU-based and FPGA-based implementations of Hestenes-Jacobi SVD, our architecture is currently the fastest in terms of overall performance. We have also evaluated the accuracy of our approach through analysis of the convergence properties.

4.3 Theoretical background

4.3.1 Singular Value Decomposition (SVD)

The Singular Value Decomposition transforms an $m \times n$ matrix into a product of an $m \times m$ orthogonal matrix, an $m \times n$ diagonal matrix with singular values and the transpose of an $n \times n$ orthogonal matrix [Trefethen and Bau (1997)] in the form of eq. (4.1).

$$A_{m\times n} = U_{m\times m}\Sigma_{m\times n}V^T_{n\times n} \tag{4.1}$$

4.3.2 Classic two-sided Jacobi rotations

Jacobi rotations are widely used in diagonalizing matrices. To perform Jacobi rotation, Jacobi rotation matrices J^l and J^r are applied to the matrix from both sides as shown in eq. (4.2). By applying the Jacobi rotation matrices to a 2×2 matrix, the off-diagonal elements are annihilated as in eq. (4.3).

$$A_{i+1} = J^l_i A_i J^r_i \tag{4.2}$$

$$J^{l'} \cdot \begin{pmatrix} A_{pp} & A_{pq} \\ A_{qp} & A_{qq} \end{pmatrix} \cdot J^r = \begin{pmatrix} A^{"}_{pp} & 0 \\ 0 & A^{"}_{qq} \end{pmatrix} \tag{4.3}$$

The Jacobi rotation matrices are generated through eq. (4.4) and eq. (4.5), where θ represents plain rotation angles α or β [Brent et al. (1985)].

$$\begin{cases} J_{pp} = cos(\theta); & \\ J_{pq} = sin(\theta); & (p < q) \\ J_{qp} = \text{-}sin(\theta); & (p < q) \\ J_{qq} = cos(\theta); & \\ J_{ii} = 1; & (i \neq p,q) \\ J_{ij} = 0, \ Others. & \end{cases} \tag{4.4}$$

$$\beta+\alpha = \arctan\left(\frac{A_{qp} + A_{pq}}{A_{qq} - A_{pp}}\right) \quad \beta\text{-}\alpha = \arctan\left(\frac{A_{qp} - A_{pq}}{A_{qq} + A_{pp}}\right) \tag{4.5}$$

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

4.3.3 Hestenes-Jacobi method

In [Hestenes (1958)], Hestenes observed that the annihilation of a matrix element is equivalent to orthogonalizing two column vectors. Instead of directly annihilating non-zero off-diagonal elements, the Hestenes-Jacobi algorithm (also known as the one-sided Jacobi method) performs the matrix decomposition through iterative orthogonal transformations between every pair of vectors. In the Hestenes-Jacobi method, the matrix is orthogonalized by columns through post-multiplying an orthogonal matrix, which is generated through a product of plane rotations as in eq. (4.6).

$$A \cdot V = B, \qquad where\ b_i^T \cdot b_j = 0 \tag{4.6}$$

Next, matrix B is further normalized through the equation $B = B \cdot \Sigma^{-1} \cdot \Sigma$, in which Σ is a diagonal matrix with the squared column norms as diagonal elements. Then, by setting $U = B \cdot \Sigma^{-1}$, eq. (4.6) can be rewritten as eq. (4.7), which is the result form of SVD.

$$A \cdot V = U \cdot \Sigma \quad \longleftrightarrow \quad A = U \cdot \Sigma \cdot V^T \tag{4.7}$$

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

4.4 Related work

In recent years, the significant surge of data dimensionality has made the application of SVD seem ubiquitous [Martin and Porter (2012)]. To compute SVD, the Householder

transformation-based method and the Jacobi rotation-based method have demonstrated satisfied stability and accuracy [Chan (1982); Drmac (1997)]. The Householder transformation [Golub and Kahan (1965); Golub and Reinsch (1970)] is capable of efficiently bi-diagonalizing matrices through vector computations, which is then followed by iterative implicit QR factorization [Demmel and Kahan (1990)] or divide-and-conquer iterations [Gu and Eisenstat (1995)] for generating singular values. In Householder transformation-based method, the SVD process is dominated by the calculations of Householder vectors and their respective updates, whose performance improvement is challenged by the inherent data dependency. To parallelize the Householder transformation, implementations have been demonstrated on GPUs [Lahabar and Narayanan (2009); Kotas and Barhen (2011)] and multi-core platforms [Haidar et al. (2013)], in which possible accelerations of GPU-based designs are achieved only for matrices with significantly large dimensions due to the iterative thread synchronization, while, the performance of implementation on multi-core platform is dominated by the task splitting and time consumption of communications.

The emergence of reconfigurable fabrics such as FPGAs introduces low-cost solutions to parallelize the algorithm at the operand-level granularity. To perform SVD, 1-dimensional or 2-dimensional systolic arrays have been employed to parallelize the classic two-sided Jacobi rotation algorithm [Brent and Luk (1982); Brent et al. (1985); Ahmedsaid et al. (2003); Ma et al. (2006)]. With the featured independent 2×2 rotations, a highly parallel 2-dimensional systolic array is employed to map the classic two-sided Jacobi rotation algorithm into FPGA devices with the computational complexity of $O(n \log n)$ for an n-by-n square matrix. However, to fit the architecture on a single chip, the scalability is limited, as n^2 processing elements (PEs) is needed by the systolic array implementation.

Compared to the classical Jacobi rotation approach, the Hestenes-Jacobi algorithm provides a more flexible solution to analyze the rectangular matrices. To explore the high performance SVD design, FPGAs and GPUs have been employed to demonstrate the parallel implementations of the Hestenes-Jacobi SVD algorithm [Hestenes (1958)]; however, the performance has suffered from iterative thread synchronizations for the implementation on GPUs [Kotas and Barhen (2011)], and the iterative design with duplicated computations in the case of FPGA implementation [Ledesma-Carrillo et al. (2011)].

4.5 Modified Hestenes-Jacobi algorithm

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

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

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

Input: matrix A
Output: singular value vector Sig
$R \leftarrow A$
`/* Generating the squared 2-norms of column vectors and their associated covariances */`
for $i \leftarrow 1$ ***to*** $NumofColumn$ **do**
  for $j \leftarrow i$ ***to*** $NumofColumn$ **do**
    $D_{i,j} \leftarrow R_i^T * R_j$
  end
end
repeat
  for $i \leftarrow 1$ ***to*** $NumofColumn - 1$ **do**
    for $j \leftarrow i$ ***to*** $NumofColumn$ **do**
      `/* Generating Jacobi rotation angle parameters with squared 2-norms of column vectors and their respective covariance */`
      $norm_1 \leftarrow D_{j,j}$; $norm_2 \leftarrow D_{i,i}$; $cov \leftarrow D_{i,j}$
      $\rho \leftarrow (norm_2 - norm_1)/(2 * cov)$
      $t \leftarrow sign(\rho)/(|\rho| + \sqrt{1+\rho^2})$
      $\cos \leftarrow 1/\sqrt{1+t^2}$; $\sin \leftarrow cos * t$
      `/* Updating the squared 2-norms affected by rotation */`
      $D_{j,j} \leftarrow D_{j,j} + t * cov$; $D_{i,i} \leftarrow D_{i,i} - t * cov$; $cov \leftarrow 0$
      `/* Updating the covariances affected by rotation */`
      for $k \leftarrow 1$ ***to*** $i - 1$ **do**
        $D_{k,i} = D_{k,i} * \cos - D_{k,j} * \sin$; $D_{k,j} = D_{k,i} * \sin + D_{k,j} * \cos$
      end
      for $k \leftarrow i + 1$ ***to*** $j - 1$ **do**
        $D_{i,k} = D_{i,k} * \cos - D_{k,j} * \sin$; $D_{k,j} = D_{i,k} * \sin + D_{k,j} * \cos$
      end
      for $k \leftarrow j + 1$ ***to*** $NumofRow$ **do**
        $D_{i,k} = D_{i,k} * \cos - D_{j,k} * \sin$; $D_{j,k} = D_{i,k} * \sin + D_{j,k} * \cos$
      end
    end
  end
until *convergence reached*
for $i \leftarrow 1$ ***to*** $\min(NumofColumn, NumofRows)$ **do**
  $Sig_i \leftarrow \sqrt{D_{i,i}}$
end

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

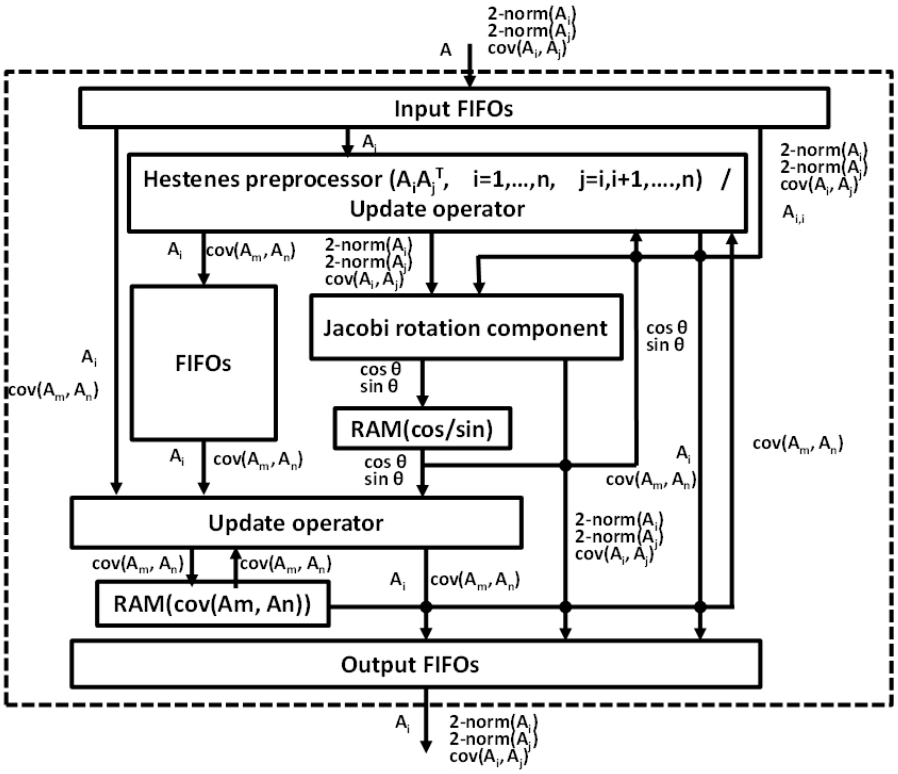


Figure 4.1: Block diagram of the Hestenes-Jacobi SVD architecture.

4.6 Our Hestenes-Jacobi SVD architecture

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

To implement the Hestenes-Jacobi SVD, we created three components: the *Hestenes preprocessor*, the *Jacobi rotation component* and the *Update operator* (shown in Fig. 4.1), all of which are pipelined. The Hestenes preprocessor is responsible for computing the squared column 2-norms and the associated covariances. The Jacobi rotation component is used to zero out the covariance through applying plane rotation with its associated vectors. The Update operator is employed to update the affected columns and covariances.

The Hestenes-Jacobi SVD is an iterative diagonalization process, which performs the orthogonal transformations between every pair of columns by numerous iterations to achieve

satisfied convergence. To reduce the amount of computations, instead of repeatedly regenerating all the squared 2-norms and covariances in each iteration, our Hestenes-Jacobi SVD architecture calculates all the squared 2-norms and covariances only in the first iteration, and then those squared 2-norms and covariances are directly updated and reused in the subsequent iterations. To reduce the hardware resource usage, the Hestenes preprocessor is reconfigured to function as an additional Update operator after the first iteration. The square-root operator in the Jacobi rotation component is employed to finalize the SVD process, from which the singular values are produced. Besides, as shown in Fig. 4.1, FIFOs are employed to synchronize the computations and transmit data between the Hestenes preprocessor and the Update operator. Local BRAMs are used to hold the generated rotation angle parameters cos and sin, and the covariances whose computations have not been completed with the current vector pairing.

4.6.1 Hestenes preprocessor

The Hestenes preprocessor is responsible for calculating the squared column 2-norms and the covariances between column vectors, in which $A_i^T * A_j$ is computed. Considering the overall system performance, we have to balance the amount of parallel computation with the I/O requests. In the Hestenes preprocessor (shown in Fig. 4.2), multiple layers of pipelined multiplier-arrays are devised, in which operands are reused by all the multipliers successively in a multiplier-array to calculate the partial results of various squared column 2-norms and their related covariances. The resulting product of a multiplier is summed up with the results of its corresponding multiplications across layers, whose operands are the matrix elements from the same columns. For example, in Fig. 4.2, the matrix element $A_{i,j+3}$ multiplies with $A_{i,j}$ at the first layer, whose product is then added to the product of multiplying $A_{i+1,j+3}$ by $A_{i+1,j}$ at the second layer, the product of multiplying $A_{i+2,j+3}$ by $A_{i+2,j}$ at the third layer, and the product of multiplying $A_{i+3,j+3}$ by $A_{i+3,j}$ at the forth layer, whose sum is the partial result of the covariance between the jth and $(j+3)th$ columns. Meanwhile, $A_{i,j+3}$ moves leftwards to be applied to the adjacent multiplier for multiplication of $A_{i,j+3}$ and $A_{i,j+1}$, whose product is used for computing the covariance between $(j+1)th$ and $(j+3)th$ columns.

The example input for a single multiplier-array with four multipliers is described in Fig. 4.3, in which the new operand requests for the multiplier array are underlined. The dashed arrows highlights the data movement for reuse and the dashed circles indicate the entered operands, which are reused in later computations. In this case, in a single layer, four double-precision floating-point numbers and at most one double-precision floating-point number are needed as the input for the starting cycle and every subsequent cycle respectively to perform the computations on a sub-vector. Then, the computations on different layers are initialized successively. Thus, 16 cycles are used for the input to obtain the covariance matrix of an 8×8 matrix if 8 layers of multiplier-arrays are equipped. Additional adders are employed to process the accumulations of partial results of covariances and squared 2-norms for vectors with the lengths over 8.

4.6.2 Jacobi rotation component

Jacobi rotation component performs the orthogonal transformation between two column vectors through a series of operations on their squared column 2-norms and the covariance between them. To calculate the Jacobi rotations, the CORDIC (for COordinate Rotation DIgital Computer) algorithm [Meher et al. (2009)] is a popular choice in the research literature, due to its advantages on efficiently performing complicated trigonometric functions through simple shift-and-add operations. Although CORDIC has been demonstrated as a hardware-efficient algorithm for fixed-point operations, its efficient floating-point implementation is challenged by its inherent bit-width shift-and-add structure. As floating-point arithmetic has become increasingly popular in signal processing applications for its support of a much wider range of values compared to decimal fixed-point format, our architecture is designed to perform floating-point calculations by using pipelined IEEE-754 double-precision floating-point operators.

As described in Algo. 1, Jacobi rotation of two column vectors is computed with their squared 2-norms and covariance through a series of addition, subtraction, multiplication, division and square-root. The Jacobi rotation equations can be represented as eq. (4.8), eq. (4.9), eq. (4.10), where n_1 and n_2 represents the squared 2-norms of column vectors, while the covariance between them is represented by $c_{1,2}$. The calculated parameter t is then applied

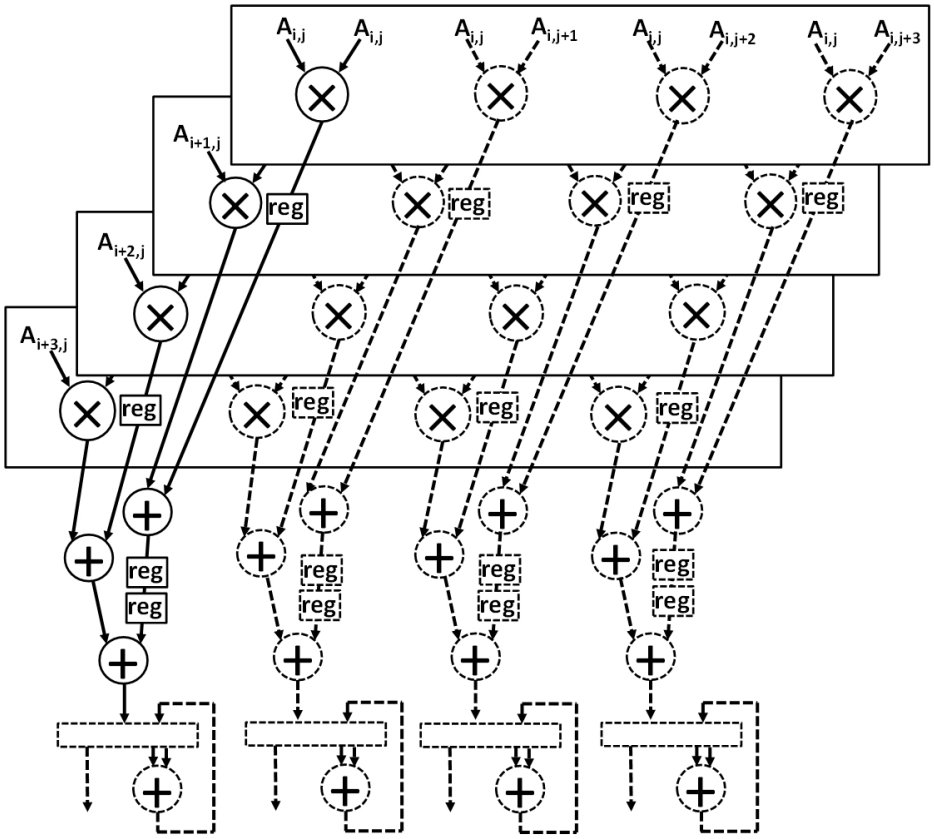


Figure 4.2: Example architecture of the Hestenes preprocessor.

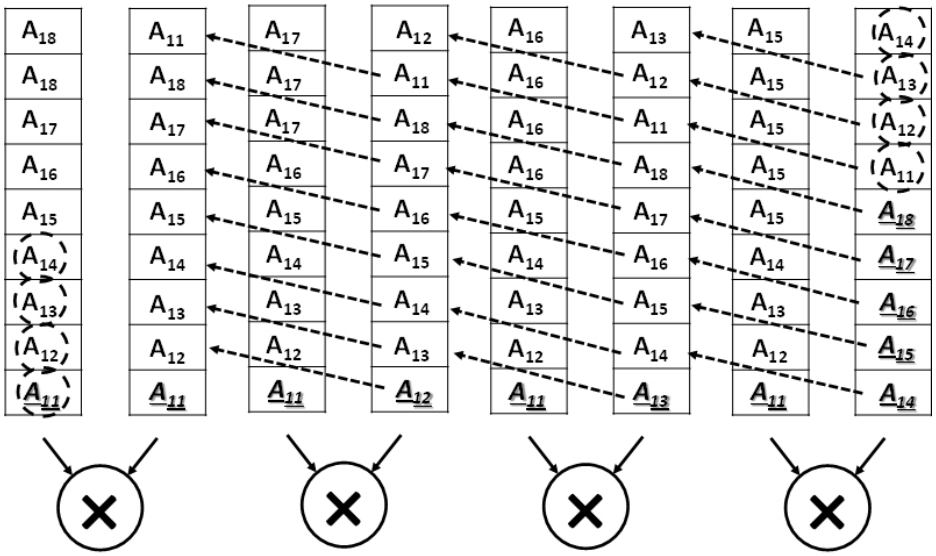


Figure 4.3: Example input to a single layer of multiplier-array.

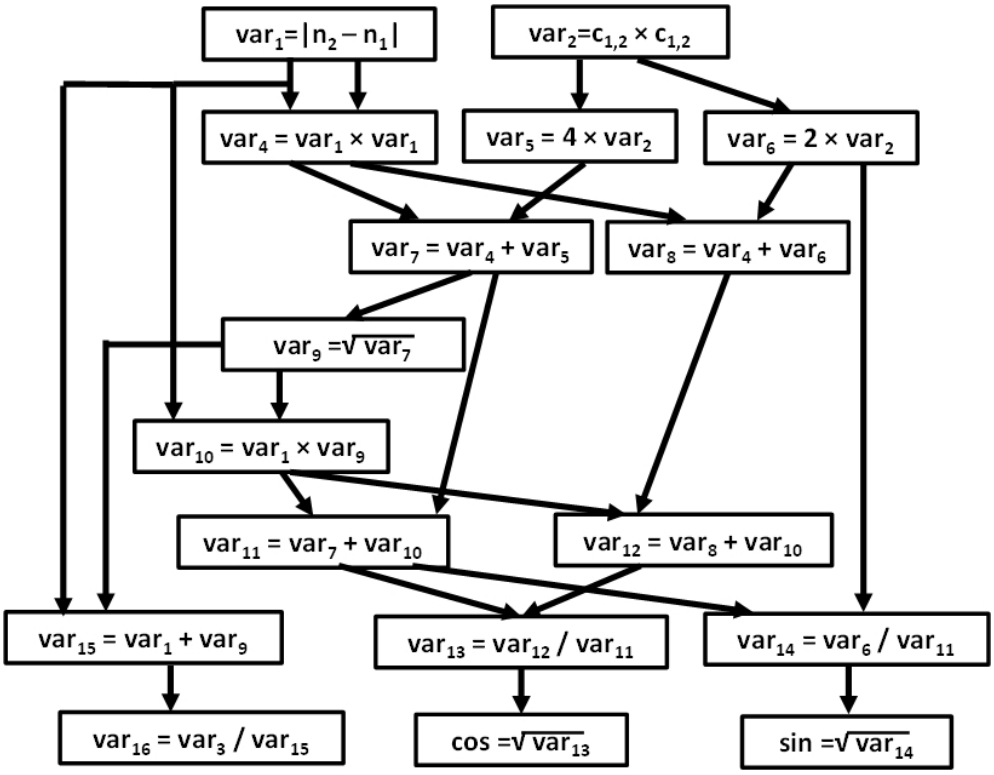


Figure 4.4: Dataflow of the Jocobi rotation procedure.

to update the squared 2-norms of rotated vectors and zero out their covariance. In Fig. 4.4, the computations of Jacobi rotation is demonstrated, in which independent calculations can be processed simultaneously. To minimize hardware resource usage, the expensive floating-point computational cores are reused by those calculations. Once all the orthogonal transformations are completed, the square-root operator in the Jacobi rotation component is used to generate the singular values by applying it to the diagonal elements of the processed matrix.

$$t = \frac{|2 * c_{1,2}|}{|n_2 - n_1| + \sqrt{(n_2 - n_1)^2 + 4 * c_{1,2}^2}} \tag{4.8}$$

$$cos = \sqrt{\frac{(n_2 - n_1)^2 + 2 * c_{1,2}^2 + |n_2 - n_1| * \sqrt{(n_2 - n_1)^2 + 4 * c_{1,2}^2}}{(n_2 - n_1)^2 + 4 * c_{1,2}^2 + |n_2 - n_1| * \sqrt{(n_2 - n_1)^2 + 4 * c_{1,2}^2}}} \tag{4.9}$$

$$sin = (sign)\sqrt{\frac{2 * c_{1,2}^2}{(n_2 - n_1)^2 + 4 * c_{1,2}^2 + |n_2 - n_1| * \sqrt{(n_2 - n_1)^2 + 4 * c_{1,2}^2}}} \tag{4.10}$$

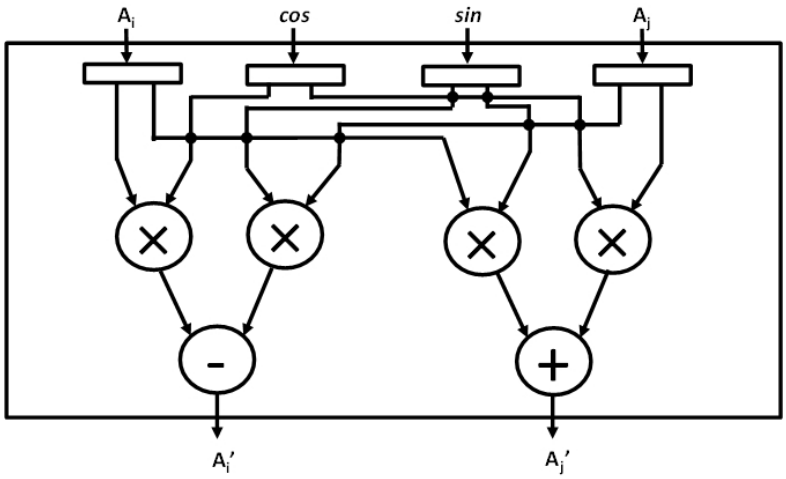


Figure 4.5: The architecture of a single update kernel.

4.6.3 Update operator

$$A_i^{'} = A_i \times cos - A_j \times sin \tag{4.11}$$

$$A_j^{'} = A_i \times sin + A_j \times cos \tag{4.12}$$

The Update operator is responsible for updating column elements and covariances which are affected by the processed rotations. Generated rotation angle parameters cos and sin are employed to update the covariances before they are used by later rotations. The update process for a pair of elements contains simple multiplications, additions and subtractions as is shown in eq. 4.11 and eq. 4.12. An architecture of a single update kernel is demonstrated in Fig. 4.5, in which pipelined multipliers, an adder and a subtractor are employed. Multiple update kernels are included in the Update operator. The number of update kernels that can be allocated to a single chip is determined by the resource capacity on the chip. This determines the efficiency of the system, especially for large-scale matrices, where performance is dominated by the amount of updates after each rotation. The convergence of SVD requires the orthogonal transformation of the matrix to be performed in numerous iterations. Both individual column elements and covariances have to be updated in the first iteration, and in the subsequent iterations, only

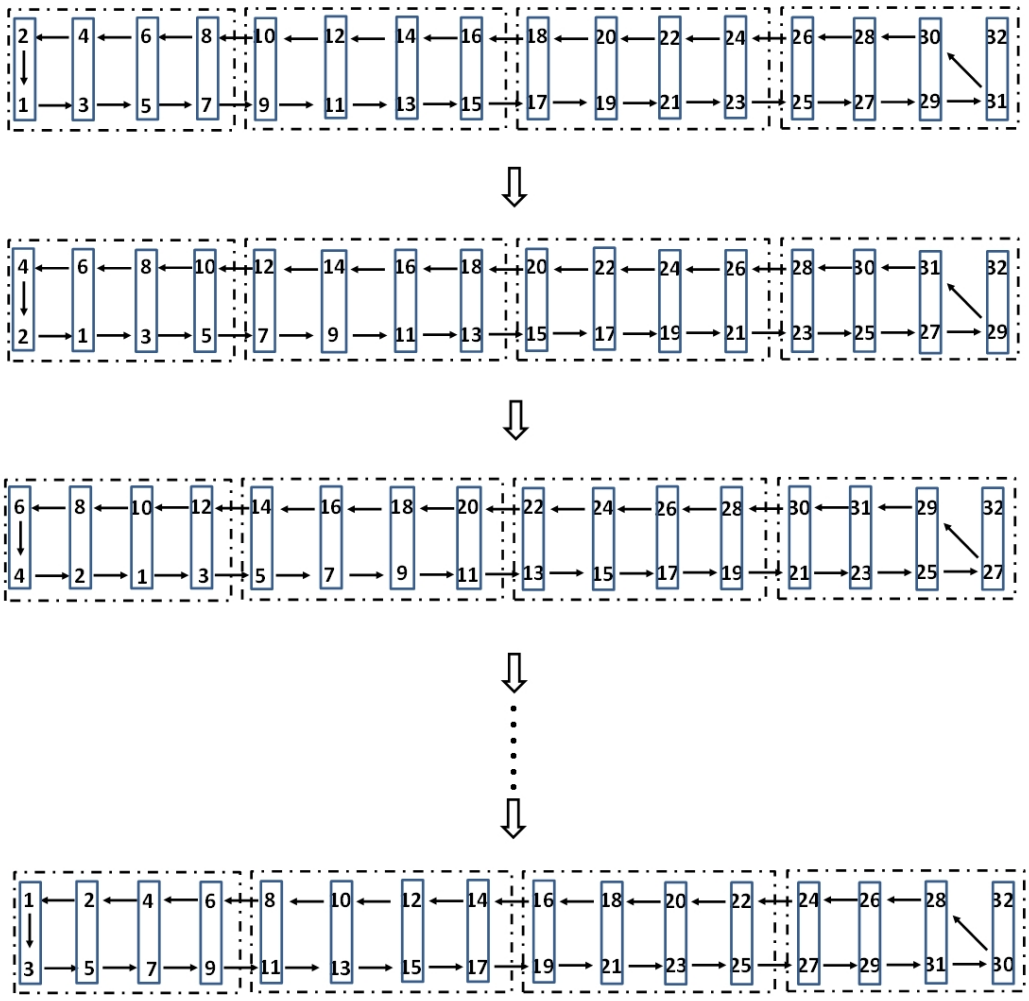


Figure 4.6: Demonstration of employed cyclic order for vector pairing.

covariances are operated. To optimize the use of hardware resources, the Hestenes preprocessor is able to be reconfigured to function as multiple update kernels.

4.6.4 The cyclic order for vector pairing

The order of vector pairing determines the speed and feasibility of the convergence. In our design, we employ the cyclic ordering, which was demonstrated with the capability of achieving convergence efficiently [Brent et al. (1985)]. In Fig. 4.6, cyclic ordering is demonstrated with 32 vectors, in which the numbers represent the column indexes, and the arrows indicate the direction for the movement of indexes to form the new vector pairs. In the cyclic ordering, each

column has to be paired with every other column. The paired vectors are highlighted by solid boxes in Fig. 4.6. Besides, due to the limited hardware resources on a single chip, a limited number of vector pairs can be operated simultaneously. In Fig. 4.6, a dashed box highlights a group of vector pairs, whose computations can be performed in parallel. All the vector-pair groups enter our Hestenes-Jacobi architecture successively.

4.7 Experiments and evaluations

4.7.1 Implementation and experimental setup

To evaluate the performance of our Hestenes-Jacobi design, a single Xilinx Virtex-5 FPGA (XC5VLX330) on our Convey HC-2 system [Convey Computer HC-2 (2012)] is used to implement our architecture. In our implementation, double-precision floating-point computational cores are generated by using Xilinx Coregen generator [Xilinx Inc. (2012)]. In the Hestenes preprocessor, four layers of multiplier-array are implemented, in which 16 multipliers and 16 adders are used. To improve the computational intensity on the limited hardware resources, the Hestenes processor calculates all the squared 2-norms and covariances in the first iteration of orthogonalization, and it is then reconfigured as four update kernels with 16 multipliers and 8 adders in the remaining iterations. To perform Jacobi rotation, 1 multiplier, 2 adders, 1 divider and 1 square-root calculators are used, which can start 8 independent Jacobi rotations in every 64 clock cycles. Additionally, an array of eight update kernels are implemented in the Update operator, which contains 32 multipliers and 16 adders or subtractors. The IP core generated computational cores are configured with default latencies as 9, 14, 57, 57 clock cycles for multiplier, adder or subtractor, divider and square-root calculator respectively. Two groups of eight 64-bit width FIFOs are programmed to synchronize the input and output, while a group of eight 127-bit width FIFOs are used for the data transmissions between the Hestenes processor and the Update operator. Simple dual port RAMs are employed to temporarily cache the rotation angle parameters and some covariances. The whole covariance matrix can be stored in the local memory for matrices of column dimension no greater than 256. The system is tested by executing at 150Mhz for 6 iterations, which is believed sufficient for achieving convergence

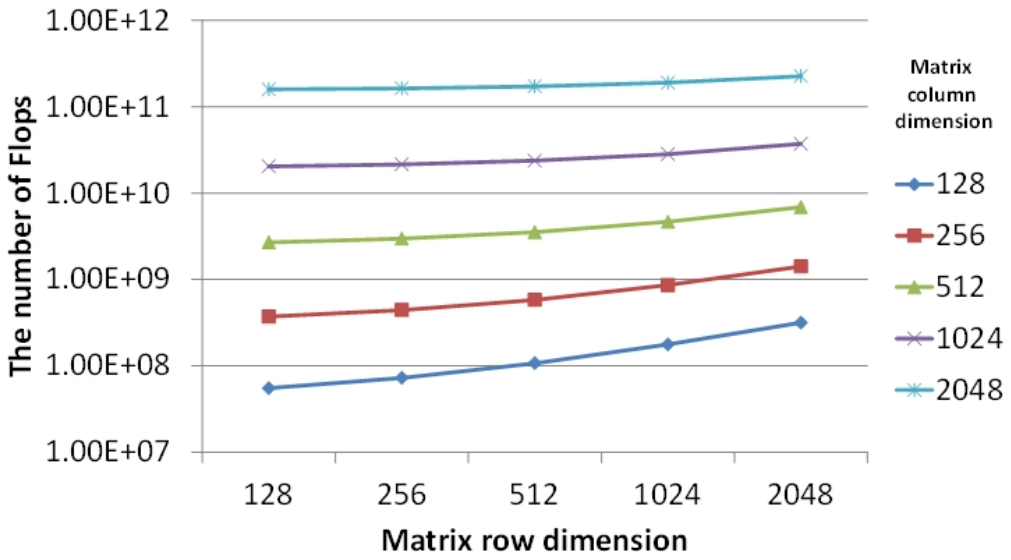


Figure 4.7: No. of floating point operations for our Hestenes-Jacobi SVD on matrices with various dimensions.

with certain thresholds. Also, a software model is implemented using Matlab to conduct the convergence evaluation.

4.7.2 Performance analysis

We experimented with both square and rectangular matrices with various dimensions, the performance of which has been summarized in Table 4.1. The experimental results demonstrate that the execution time grows significantly as the number of matrix columns increases, which determines the amount of covariances, whose computation dominates the overall performance. Comparably, the number of rows, which only affects the execution time of the Hestenes preprocessing, has smaller impact on the performance. In Fig. 4.7, the number of floating point operations required for Hestenes-Jacobi SVD on matrix with various dimensions is demonstrated, in which column dimensions have more significant impact on determining the amount of floating operations due to only the covariance matrix is operated after the first iteration of computing, and the size of covariance matrix is determined by the column dimension of the input matrix.

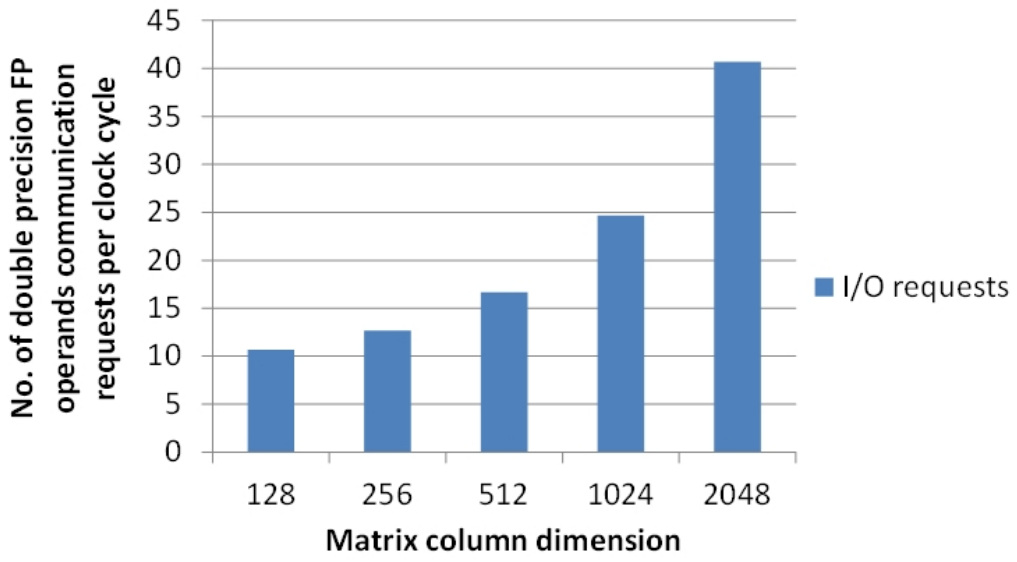


Figure 4.8: The average No. of double precision floating point operands communication requests per clock cycle for input matrix with various column dimensions.

When the matrix column size grows over 256, the performance is increasingly affected by the I/O bandwidth due to the increased covariance communications have to be made between our Hestenes-Jacobi architecture and off-chip memory. Fig. 4.8 demonstrates the average number of double-precision floating-point I/O requests per clock cycle for input matrix with various column dimensions. Due to the Hestenes-Jacobi SVD algorithm is column vector oriented, and the amount of I/O requests is determined by the matrix column dimension. With limited on-chip storage resources, the number of I/O requests grows as the column dimension increases. The I/O bandwidth starts affecting the system performance as it grows over 16, which number is the optimized I/O bandwidth that is provided by our experimental platform.

Comparisons of execution times have been made between our implementation and experimental results from the published literature [Lahabar and Narayanan (2009)] as well as a Matlab SVD routine. In Fig. 4.9 and Fig. 4.10, the performance of our design, the Matlab 7.10.0 SVD routine running on a 2.2 GHz dual core Intel Xeon processor, SVD solutions with Intel MLK 10.0.4 and NVIDIA 8800 GPU with 128 stream processors [Lahabar and Narayanan (2009)] have been demonstrated. By analyzing those data points in Fig. 4.9, our architecture has better efficiency than other software solutions when matrix with dimensions under 512, and

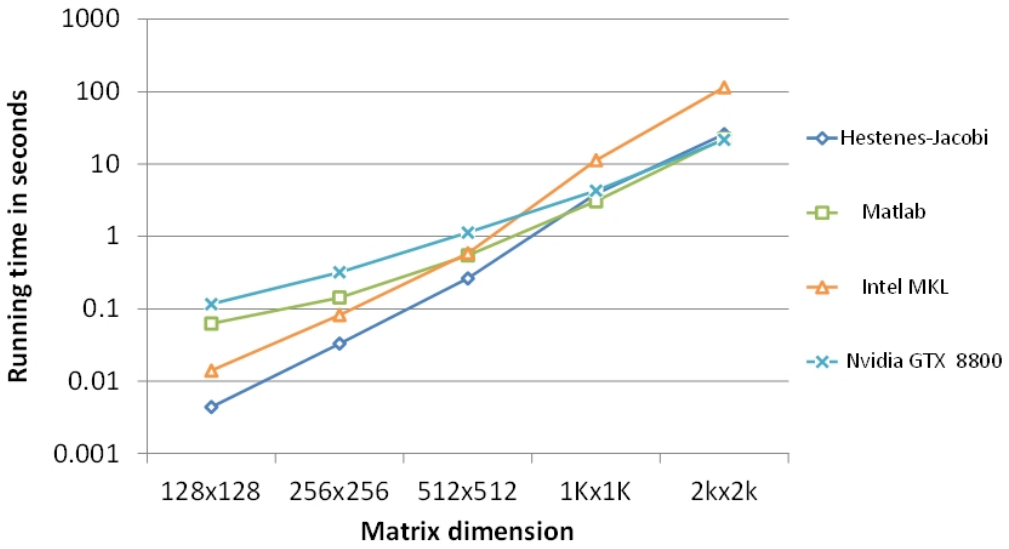


Figure 4.9: SVD computation time (in seconds) for square matrices by our Hestenes-Jacobi architecture, Matlab, Intel MKL and GPU.

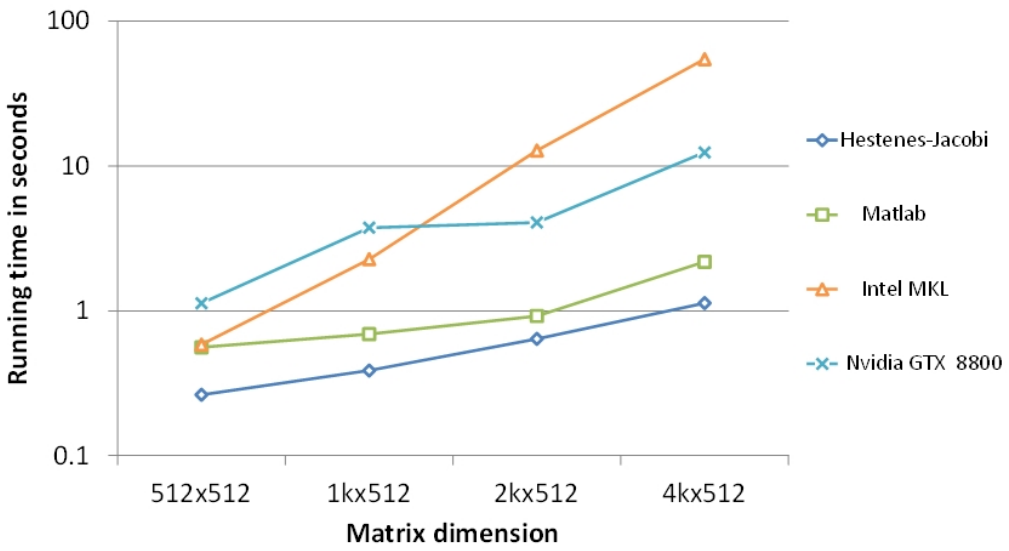


Figure 4.10: SVD computation time (in seconds) for rectangular matrices by our Hestenes-Jacobi architecture, Matlab, Intel MKL and GPU.

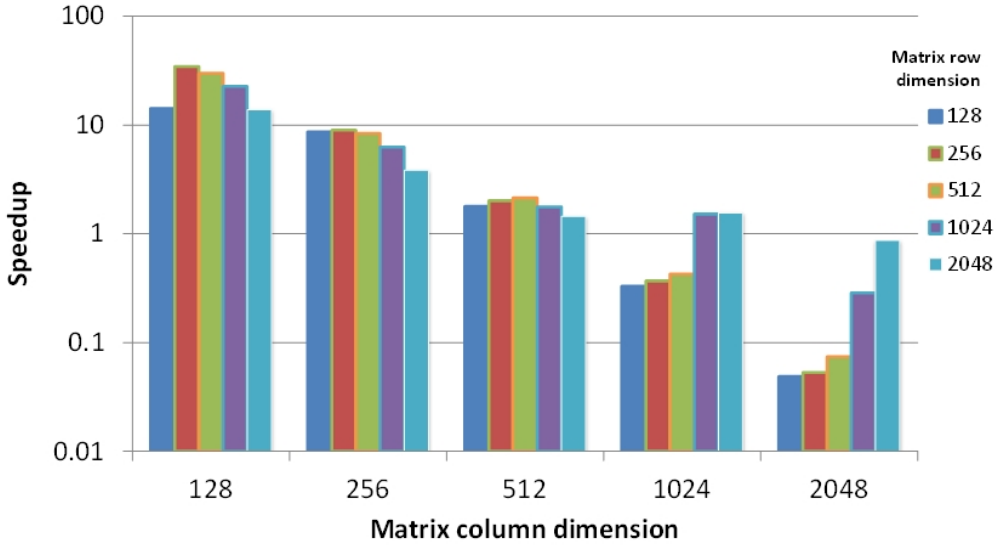


Figure 4.11: Speedups of our Hestenes-Jacobi SVD compare to Matlab SVD.

our execution slows down when the dimensions over 512 due to the limits of our chosen platform's I/O throughput. In Fig. 4.10, the comparison is made between matrices with identical column dimensions but various row sizes, which indicates the growth of row number causes a relatively slow increase of the execution time due to the quantity of covariances is determined by the column size.

In Fig. 4.11, the dimensional speedups of our FPGA-based Hestenes-Jacobi SVD compared to the Matlab SVD solution running on an Intel platform are presented, in which our Hestenes-Jacobi architecture shows better efficiency in analyzing matrices with small to medium column dimensions compared to the standard software solution, even when they are with comparably large row dimensions. The dimensional speedups that can be achieved range from 3.8× to 43.6× for matrices with column sizes from 128 to 256 and row dimensions from 128 to 2048. Table 4.2 shows the resource utilization by our Hestenes-Jacobi architecture.

Compared to the experimental results of the latest GPU-based and FPGA-based Hestenes-Jacobi implementations, our architecture shows the best performance [Ledesma-Carrillo et al. (2011); Kotas and Barhen (2011)]. In [Ledesma-Carrillo et al. (2011)], the GPU-based implementation, which ran 106.90ms and 1022.92ms to decompose a 128×128 and a 256×256 matrix respectively, failed to achieve any speedup compared to a conventional software solution. The

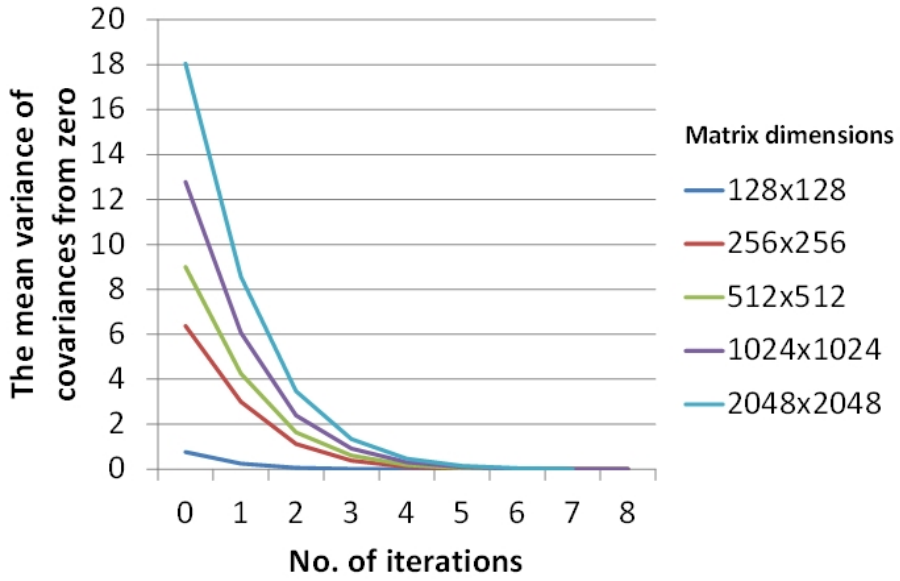


Figure 4.12: Convergence process of different dimensional matrices.

FPGA-based design [Kotas and Barhen (2011)] was devised to perform fixed-point operations, which can only analyze the matrices with the size up to 32×128 due to the limitation of on-chip memory. Although the better performance has been demonstrated compared to their software execution with Matlab SVD for matrices with dimensions range from 2×2 to 32×127, our Matlab SVD routine runs 100 times faster than their Matlab SVD, and shares comparable speeds with their FPGA-based design. In further comparison to [Kotas and Barhen (2011)], in which 24.3143ms is needed to decompose the largest analyzed matrix with the dimensions of 32×127, the execution time of operating a 128×128 matrix by our architecture shows more than 5 times speedup.

Table 4.1: Execution time in seconds.

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

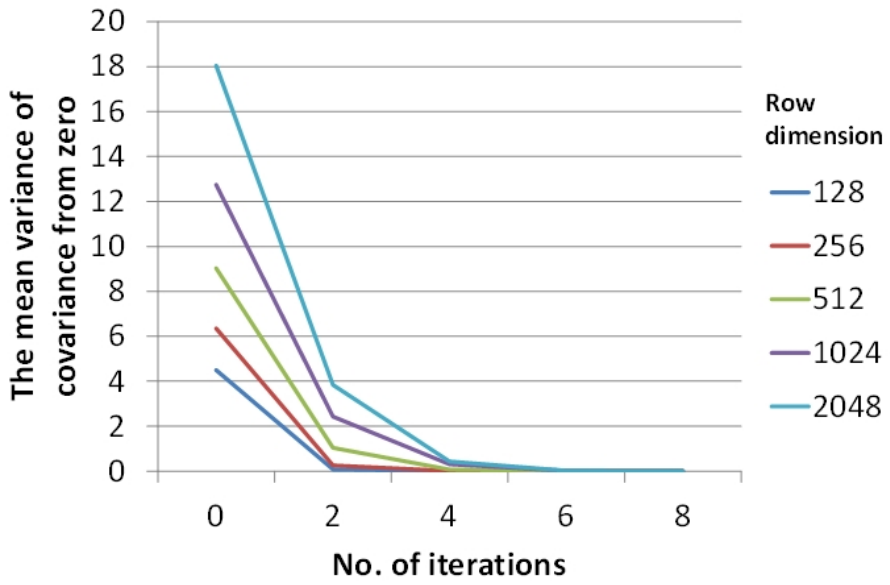


Figure 4.13: Convergence process of matrices with column size of 1024 and various row sizes.

Table 4.2: Resource consumption of our Hestenes-Jacobi architecture.

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

4.7.3 Convergence analysis

SVD is a process of diagonalizing matrix through iterative rotations; to evaluate the correctness and accuracy of the Hestenes-Jacobi produced singular values, the convergence speed needs to be analyzed. In our evaluation, randomly generated datasets have been applied to the implemented software model of our Hestenes-Jacobi design. The mean absolute deviations from zero of the covariances after being processed by a number of iterations are shown in Fig. 4.12, in which covariances between column vectors are rapidly converged to zero as the number of processing iterations increase. Reasonable convergence can be achieved within 6 iterations of operations for matrices of dimensions no greater than 2048. Also, similar observations can be obtained from the convergence performance evaluation of $m \times n$ matrices (see Fig. 4.13), in which the applied datasets are with identical column size of 1024 but various row dimensions.

4.8 Conclusion

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

CHAPTER 5. A RECONFIGURABLE ARCHITECTURE FOR QR DECOMPOSITION USING A HYBRID APPROACH

Modified from a paper published in

Proceedings of 2014 *International IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*

Xinying Wang[1], Phillip Jones and Joseph Zambreno[2]

5.1 Abstract

QR Decomposition has been widely used in many signal processing applications to solve linear inverse problems. However, QR Decomposition is considered a computationally expensive process, and its sequential implementations fail to meet the requirements of many time-sensitive applications. The Householder transformation and the Givens rotation are the most popular techniques to conduct QR Decomposition. Each of these approaches have their own strengths and weakness. The Householder transformation lends itself to efficient sequential implementation, however its inherent data dependencies complicate parallelization. On the other hand, the structure of Givens rotation provides many opportunities for concurrency, but is typically limited by the availability of computing resources. We propose a deeply pipelined reconfigurable architecture that can be dynamically configured to perform either approach in a manner that takes advantage of the strengths of each. At runtime, the input matrix is first partitioned into numerous sub-matrices. Our architecture then performs parallel Householder transformations on the sub-matrices in the same column block, which is followed by parallel Givens rotations to annihilate the remaining unneeded individual off-diagonals. Analysis of our design indicates the

[1]Primary researcher and author
[2]Correspondence author

potential to achieve a performance of 10.5 GFLOPS with speedups of up to 1.46×, 1.15× and 13.75× compared to the MKL implementation, a recent FPGA design and a Matlab solution, respectively.

5.2 Introduction

QR Decomposition has been widely used in many signal processing applications such as MIMO systems [Xue et al. (2013)], beamforming [Liu and McCanny (2003)] and image recovery [Qiu and Vaswani (2011)] to calculate the inverse of matrices or solve linear systems. However, its inherent computational complexity makes it unlikely to satisfy the requirements of many time-sensitive designs, especially when the system operates on a large-scale dataset. QR Decomposition is generally considered as an $O(n^3)$ operation, and previous research has shown that more than 10 minutes could be taken to perform QR Decomposition-based robust Principal Component Analysis on a $110,592 \times 100$ matrix, which is far beyond the requirements of potential real-time applications such as video surveillance or traffic monitoring [Anderson et al. (2011)].

The Gram-Schmidt process, Householder transformation and Givens rotation are known as the most popular algorithms for QR Decomposition [Golub and Van Loan (1996)], among which, the Householder transformation and the Givens rotation are considered numerical stable algorithms, while the Gram-Schmidt process provides an opportunity to perform successive orthogonalizations. Parallel designs have been previously investigated to accelerate QR Decomposition on traditional multi-core systems [Dongarra et al. (2012); Soliman (2011)], GPUs [Kerr et al. (2009)] and reconfigurable computing platforms [Wang and Leeser (2009); Tai et al. (2011); Rafique et al. (2012); Aslan et al. (2012)].

The Householder transformation is efficient in its vectorized operations. However, parallelization of the Householder transformation is challenged by the data dependencies among vectors [Leoncini et al. (1996)]. To help mitigate the issue of data dependency, a tiled QR Decomposition (also known as the blocked Householder transformation) was proposed [Bouwmeester et al. (2011)], and has been demonstrated to better exploit the parallelism available on multicore CPUs [Dongarra et al. (2012)], GPU [Kerr et al. (2009)] and FPGAs [Rafique et al. (2012); Tai et al. (2011)].

The Givens rotation provides better opportunities for highly parallel designs. However, the scalability of Givens rotation-based QR Decomposition is typically limited by the $O(n^2)$ processing elements (PEs) needed to fully parallelize those rotations for an $n \times n$ matrix [El-Amawy and Dharmarajan (1989)]. A two-dimensional systolic array was devised for fast parallel Givens rotations on a single FPGA [Wang and Leeser (2009)]. However, the scalability was severely restricted due to the large amounts of resources required.

In this chapter, we present a hybrid approach that leverages the strengths of both Householder transformation and Givens rotation by applying the most appropriate of the two at each stage of the QR Decomposition process. We propose a reconfigurable architecture for QR Decomposition, which can be dynamically configured to perform either Householder transformation or Givens rotation, both of which are deeply pipelined. To process large data sets, the input matrix is partitioned into multiple columns of sub-matrices. The sub-matrix columns are processed successively, while the sub-matrices in the same column are applied with parallel independent Householder transformations. Then, the dense sub-matrix column is transformed into numerous upper triangular sub-matrices, on which highly parallel Givens rotations are performed to annihilate the remaining non-zero elements. Our experimental results show our design can achieve 10.5 GFLOPS with speedups of up to 1.46×, 1.15× and 13.75× compared to the Intel Math Kernel Library (MKL) implementation on a single CPU core [Buttari et al. (2009)], an FPGA-based tiled matrix decomposition [Tai et al. (2011)], and a single threaded Matlab routine, respectively.

5.3 Theoretical background

5.3.1 QR Decomposition

QR decompostion of an $m \times n$ matrix A has a form given by eq. (6.1)

$$A{=}QR \tag{5.1}$$

where Q is an $m \times m$ matrix, which is an orthogonal matrix such that $Q^T \cdot Q = I$, and R is an $m \times n$ upper triangular matrix [Golub and Van Loan (1996)].

5.3.2 Householder transformation

The Householder transformation [Golub and Van Loan (1996)] is a linear process that reflects a vector through a plane containing the origin. The transformed vector is orthogonal to and has the same norm as the original vector. To perform the linear reflection of a vector x, the Householder matrix as shown in eq. (5.2) is employed, in which v is a unit vector orthogonal to the plane.

$$H{=}\ I\ \text{-}\ 2vv^T \tag{5.2}$$

To perform the transformation so that all the elements in the transformed vector below the first entry are zero, the unit vector v can be constructed as shown in eq. (5.3) and eq. (5.4), where e is a unit vector $(1, 0, 0, \cdots, 0)^T$.

$$u{=}\ x{+}\|x\|e \tag{5.3}$$

$$v{=}\ \frac{u}{\|u\|} \tag{5.4}$$

By applying the Householder matrix, the householder transformation is performed, eq. (5.5), where c is $\pm\|x\|$.

$$H \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \cdots \\ x_n \end{bmatrix} = \begin{bmatrix} c \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{5.5}$$

5.3.3 Givens rotation

The Givens rotation introduces zeros to matrices through plane rotations. After determining the plane rotation angle (θ) for paired elements, as shown in the eqs. (5.7,5.8,5.9), zero elements can be introduced by conducting rotations in the form of eq. (5.6) [Golub and Van Loan (1996)]. Since only two elements are operated on in a single rotation, the Givens rotation provides better opportunities to process individual components in parallel.

$$\begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} x \\ 0 \end{bmatrix} \tag{5.6}$$

$$x = \sqrt{a^2 + b^2} \tag{5.7}$$

$$\cos\theta = \frac{a}{x} \tag{5.8}$$

$$\sin\theta = \frac{-b}{x} \tag{5.9}$$

5.4 Related work

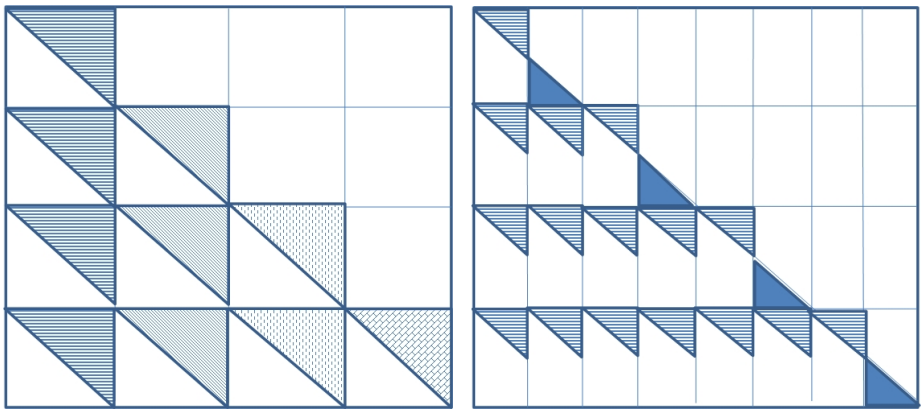
The numerical stability and sequential implementation efficiency of the Householder transformation has lead it to be employed in many standard software packages (e.g. Matlab, LAPACK) [Walker (1988)]. Parallel implementations of the Householder transformation have

been investigated on multi-core systems, GPUs and reconfigurable computing platforms [Soliman (2011); Kerr et al. (2009); Rafique et al. (2012)]. However, the performance improvement is challenged by its inherent data dependencies. Although the tiled matrix [Bouwmeester et al. (2011)] was introduced to efficiently partition data sets, near ideal speedups on multi-core platform-based or GPU-based designs are achieved only for matrices with large dimensions due to heavy inter-core communication [Soliman (2011); Kerr et al. (2009)]. In [Rafique et al. (2012)], an efficient FPGA-based QR decomposer for tall-skinny matrices was presented. An additional concern with this decomposer is that during the merge stage parallelism decreases as the quantity of intermediate results reduce.

The Givens rotation has been proven to be the most accurate and stable approach for QR Decomposition [Golub and Van Loan (1996); Gentleman (1975)]. Compared to the Householder transformation, the Givens rotation provides more opportunities for parallelism, especially when annihilating individual isolated elements. In [El-Amawy and Dharmarajan (1989); Wang and Leeser (2009); Echman and Owall (2005)], a 2-dimensional systolic array was employed to demonstrate the parallel implementation of Givens rotation in hardware. However, their scalability was constrained due to the limited resources on a single chip. As demonstrated in [Echman and Owall (2005)], 86% of a Virtex-II FPGA's resources were consumed to factorize a 4×4 matrix. Our proposed architecture looks to leverage the benefits of both the sequential efficiency of the Householder transformation and parallelizability of Givens rotation by using a hybrid approach.

5.5 Hybrid QR algorithm

As previously mentioned, the Householder transformation is able to efficiently zero out all the components of a vector below the first entry. However, its inherent data dependency makes parallelization challenging. The Givens rotation provides better opportunities to pursue parallelism and more flexibility for processing individual isolated elements. However, decomposing a large dense matrix by Givens rotation requires a large number of rotation operations. To improve the performance of QR Decomposition by combining the advantages of both algorithms, our approach divides the input matrix into a number of sub-matrices, on which local House-



(a) $p \times p$ sub-matrices (b) $p \times q$ sub-matrices

Figure 5.1: An example sub-matrix partition of an $m \times m$ matrix for our Hybrid QR Decomposition algorithm.

holder transformations are performed in parallel on sub-matrices in the same column block. Then, parallel Givens rotations are employed to annihilate the remaining isolated non-zero lower triangular elements. Compared to [Rafique et al. (2012)], which targeted tall-skinny matrices and employed the Householder transformation both to factorize sub-matrices and merge the transformed sub-matrices, Givens rotations are better for conducting parallel processing at the merge stage, especially when floating-point arithmetic is used, whose computations have relatively long latencies. In addition, the Givens rotation can potentially achieve additional acceleration when factorizing partially sparse matrices.

In our hybrid QR Decomposition algorithm, the input matrix is divided into $m \times n$ sub-matrices. The sub-matrices from the same columns can be processed in parallel both for factorization and updates, while sub-matrices having the same row indices are processed successively from left to right. The factorization process and update operation can be performed simultaneously if no data dependencies exist between them. The sub-matrices can either be square as shown in Fig. 5.1a, or rectangular, as shown in Fig. 5.1b. As demonstrated in Fig. 5.1, the lower triangular part of the matrix has been transformed into a number of upper triangular sub-matrices (as depicted by the shaded area) by parallel Householder transformations, which then will be annihilated by parallel Givens rotations.

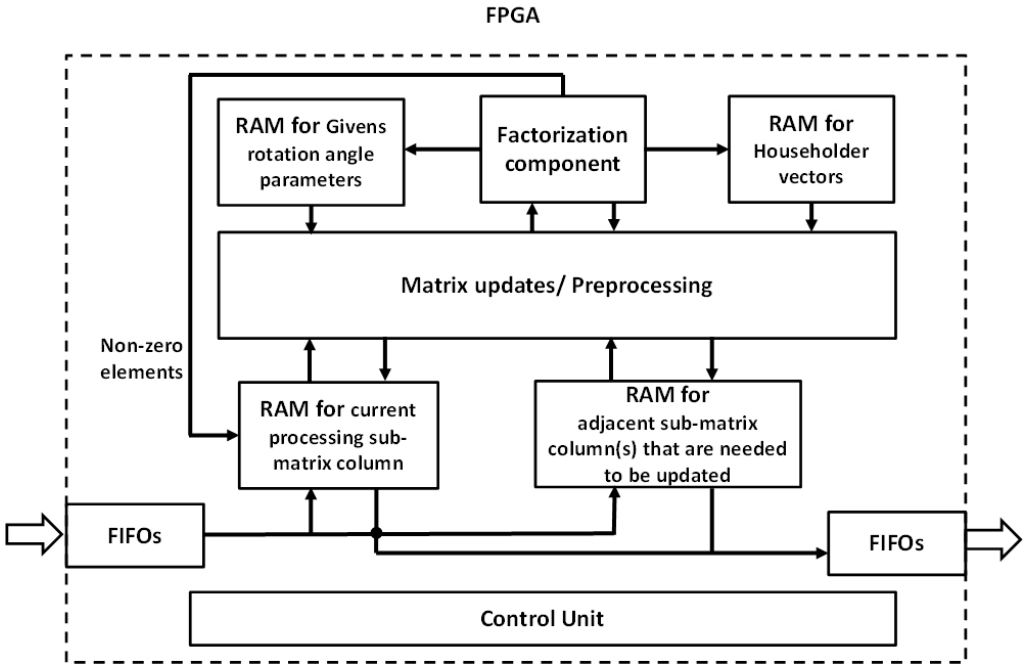


Figure 5.2: Block diagram of our Hybrid QR Decomposition architecture.

5.6 Our architecture for QR Decomposition

A reconfigurable computing platform provides a flexible medium for dynamically configuring our architecture to perform either the Householder transformation or the Givens rotation algorithm. The detailed calculations of the Householder transformation algorithm and the Givens rotation approach are described in Table 5.1, both of which can be primarily summarized as three steps: a) preprocessing, b) factorization, and c) matrix updates. In the preprocessing stage, the Householder transformation employs vector-vector multiplication to compute the squared vector norms, while the Givens rotation performs multiplication-addition operations, both computations are required during the matrix update phase. To help optimize hardware resource usage, a deeply pipelined component is devised that performs both preprocessing and matrix update operations. Fig. 5.2 provides a high-level block diagram of our hybrid QR Decomposition architecture. The matrix updates/preprocessing component and the factorization component are the pipelined computational engines. RAMs are employed to temporarily hold the generated Householder vectors and Givens rotation parameters. Additionally to help reduce communication overheads, the current processing sub-matrix column and the adjacent

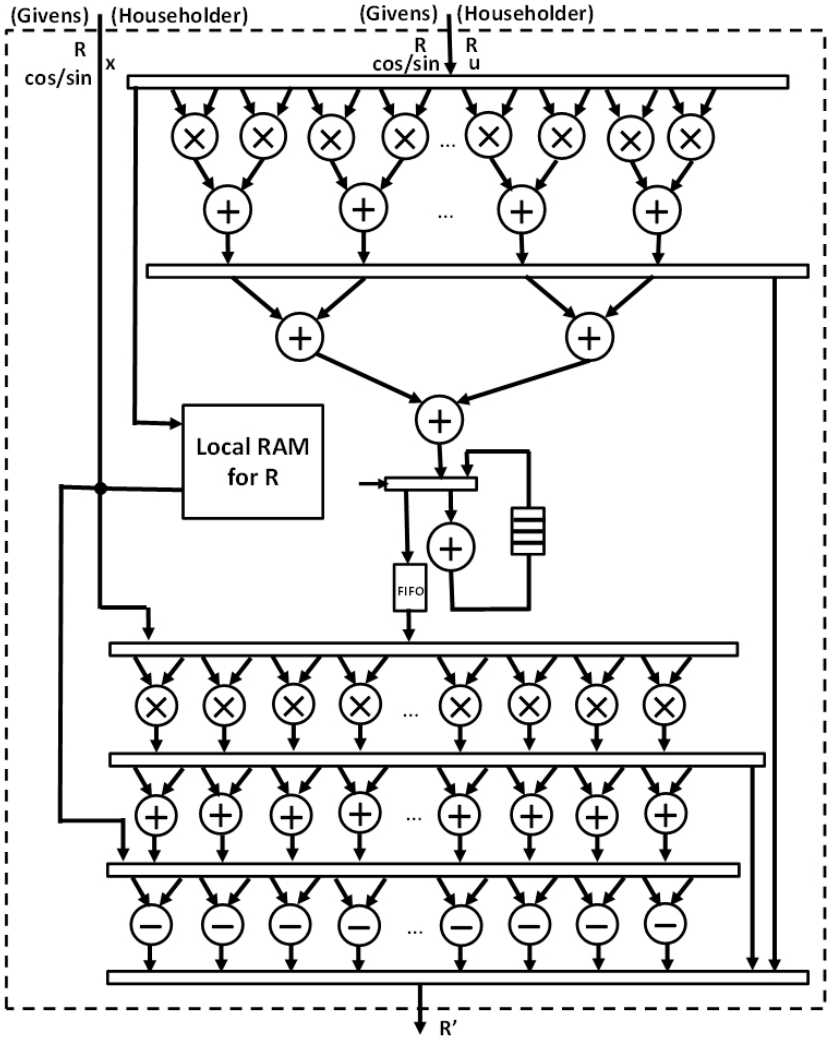


Figure 5.3: Matrix updates/preprocessing component architecture.

sub-matrix columns are kept in local memory. The adjacent sub-matrix column is moved to the local storage of current processing sub-matrix column when all its respective updates have completed. The number of sub-matrix columns that can be held on chip is determined by the amount of on-chip resources as well as the matrix dimensions.

5.6.1 Preprocessing component

The preprocessing component is responsible for producing the squared vector norms for the Householder transformation and square sums for the Givens rotations. To calculate the squared vector norms, a reverse binary tree structure was implemented as shown in Fig. 5.3.

Table 5.1: Hybrid QR Decomposition approach computations.

Phases	Householder	Givens Rotation
Preprocessing	$x \leftarrow \begin{bmatrix} R_{i,i} \\ R_{i+1,i} \\ R_{i+2,i} \\ \cdots \\ R_{n,i} \end{bmatrix}$ $v_1 \leftarrow R_{i,i}^2 + R_{i,i+1}^2 \cdots + R_{n,i}^2$	$t_1 \leftarrow R_{i,i}$ $t_2 \leftarrow R_{j,i}$ $t \leftarrow t_1^2 + t_2^2$
Factorization	$e_1 \leftarrow \sqrt{v_1}$ $v_2 \leftarrow (R_{i,i} - e_1)^2 + R_{i,i+1}^2 \cdots + R_{n,i}^2$ $e_2 \leftarrow \sqrt{v_2}$ $x_1 \leftarrow R_{i,i} - e_1$ $u \leftarrow \frac{x}{e_2}$	$r \leftarrow \sqrt{t}$ $cos\theta = \frac{t_1}{r}$ $sin\theta = -\frac{t_2}{r}$ $R_{i,i} \leftarrow r$ $R_{j,i} \leftarrow 0$
Matrix Updates	$R \leftarrow R - 2 * u * u^T * R$	$k \leftarrow 1$ **to** $NumofColumn$ $\begin{bmatrix} Q_{k,i} \\ Q_{k,j} \end{bmatrix} \leftarrow \begin{bmatrix} cos\theta & sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} Q_{k,i} \\ Q_{k,j} \end{bmatrix}$

The top level is equipped with n multipliers, under which there are $\lceil \log(n) \rceil$ levels of adders. An additional adder is employed as an accumulator when the vector length is greater than the number of multipliers. Only the multipliers and top level adders are used to perform preprocessing for Givens rotation. To save hardware resources, the preprocessing component is implemented as part of the matrix update component.

5.6.2 Factorization component

$$e_1 = \sqrt{R_{i,i}^2 + R_{i+1,i}^2 \cdots + R_{n,i}^2} \tag{5.10}$$

$$\begin{aligned} e_2 &= \sqrt{(R_{i,i} - e_1)^2 + R_{i+1,i}^2 \cdots + R_{n,i}^2} \\ &= \sqrt{R_{i,i}^2 - 2 * R_{i,i} * e_1 + e_1^2 + R_{i+1,i}^2 \cdots + R_{n,i}^2} \\ &= \sqrt{2 * (R_{i,i}^2 + R_{i+1,i}^2 \cdots + R_{n,i}^2 - R_{i,i} * e_1)} \end{aligned} \tag{5.11}$$

The factorization component uses a unified architecture to perform both the Householder transformation and the Givens rotation. Fig. 5.4 illustrates this reconfigurable architecture as a data flow graph. All of the computational cores are deeply pipelined, and factorization can be switched between the Householder transformation and the Givens rotation seamlessly.

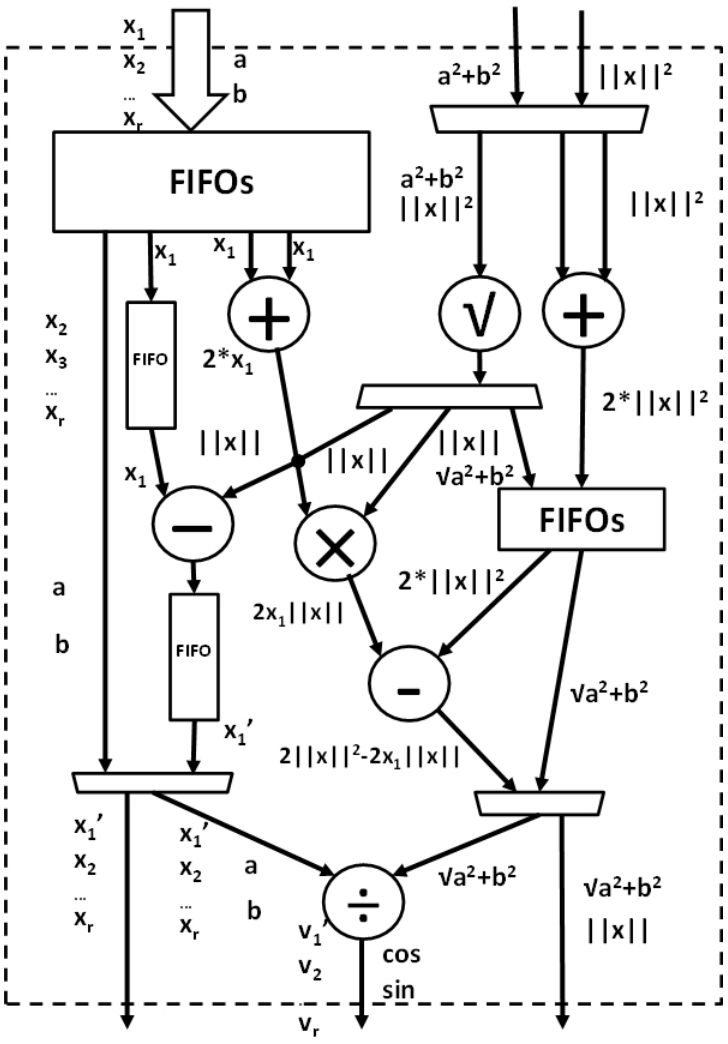


Figure 5.4: Dataflow view of the factorization component.

FIFOs are employed to synchronize the data streams, while FSM-based control units are used to manage the runtime configuration of the architecture.

For the Householder transformation, whose input is the original untransformed vector and the sum of its squared vector elements, the process is started by computing the L2-norm e_1 with the square root operation as shown in eq. (5.10), where i is the index of the column vector within a sub-matrix and n is its row dimension. In parallel, additions are performed by a pair of adders to calculate $2 * R_{i,i}$ and $2 * (R_{i,i}^2 + R_{i+1,i}^2 \cdots + R_{n,i}^2)$ respectively, in which $R_{i,i}$ is the first entry of the input vector. Then, a multiplier is used to multiply $2 * R_{i,i}$ by e_1, which is followed by the subtraction, $2 * (R_{i,i}^2 + R_{i+1,i}^2 \cdots + R_{n,i}^2) - 2 * R_{i,i} * e_1$, whose result

is the square of e_2 (eq. 5.11). To obtain the Householder vector, a division calculates $\frac{x}{e_2^2}$, in which e_2^2 is the result of the subtractor and x is identical to the input vector except the first entry has been transformed to $x_1 - e_1$. Since the use of a Householder vector is in the form of vector-vector multiplication ($\frac{x}{e_2} * \frac{x'}{e_2}$) in the subsequent matrix updates, its computation can be replaced by $x * \frac{x'}{e_2^2}$. This makes the square root operation of e_2^2 unnecessary. The final output of the Householder transformation is the Householder vector $\frac{x}{e_2^2}$ and e_1, the first and only non-zero entry of the transformed vector.

To perform the Givens rotation, paired matrix elements and the sum of their squares are entered. The square root operation is employed to compute $\sqrt{a^2 + b^2}$, after which $\frac{a}{\sqrt{a^2+b^2}}$ and $\frac{b}{\sqrt{a^2+b^2}}$ are calculated by the divider to produce the Givens rotation parameters $\cos\theta$ and $\sin\theta$ respectively.

5.6.3 Matrix update component

As described in Table 5.1, the update process of the Householder transformation is conducted through matrix-vector operations, while the Givens rotation uses simple element-wise multiplications and additions/subtractions to update the affected matrix entries. The brute-force way to compute $R \leftarrow R - 2 * u * u^T * R$ is started by the vector-vector multiplication $u * u^T$, whose result is a matrix. Then, the result matrix is multiplied by matrix R, which makes the time complexity of the update process $O(n^3)$. To help optimize the computation, $u^T * R$ is calculated first, whose product is a row vector. Next, vector-vector multiplication is performed. This reduces the time complexity of the update process to $O(n^2)$, thus largely reducing computational workload, especially for large-scale data sets.

Fig. 5.3 illustrates the update component architecture. The reverse binary tree structure is responsible for vector-matrix multiplication, while the multipliers in the lower half of the architecture are employed to perform vector-vector multiplication, which is followed by adders to double the results from the multipliers. The Householder transformation update process ends with subtractions. The Givens rotation updates only use the multipliers and the first level of adders below those multipliers. Just as for the factorization component, the architecture is deeply pipelined and can be configured to perform updates for either the Householder transformation or the Givens rotation.

5.6.4 I/O considerations

In our design, the input matrix is assumed to be stored in an off-chip memory. Only the column of sub-matrices under processing and their adjacent sub-matrix columns are temporally held on-chip. The number of sub-matrices that can be kept on-chip is determined by the sub-matrix dimension and the capacity of on-chip memory. Our analysis indicates that on average only 4 double-precision floating-point operands need to be communicated between the architecture and the off-chip memory each clock cycle to factorize a 1024×1024 matrix, which consists of 16×16 sub-matrices based on the available on-chip memory.

5.7 Implementation and evaluation

5.7.1 Implementation and experimental setup

Our design is implemented in VHDL on a single Xilinx Virtex-5 XC5VLX330 FPGA of the Convey HC-2 platform [Convey Computer HC-2 (2012)], for which eight memory controllers with a total of 16 DDR2 channels are available. Our architecture uses double-precision floating-point IP cores [Xilinx Inc. (2012)] that are configured to use 9, 14, 57 and 57 pipeline stages for multipliers, adders/subtractors, dividers and square roots respectively. The usage of double-precision computational cores of our hardware implementation is presented in Table 5.2. All multipliers, dividers and square roots are implemented as LUT logic only, while adders/subtractors are configured to use LUT logic or dedicated multiplier circuitry (DSPs). To

avoid bottlenecks incurred by the bandwidth of on-chip memories, high-bandwith BRAMs are needed. To improve the flexibility of data usage, instead of using single ported high-bandwith BRAMs, simple dual-ported BRAMs were employed, for which each BRAM was configured with a width of 64-bits for input and output ports. Besides, a number of FIFOs with configurations of both width and depth as 64 are deployed in modules to buffer the data communicated among different computational components of this architecture. The detail on-chip memory usage is demonstrated in Table 5.3. The Xilinx design tools reported that our placed and routed design consumes 85.7% of the slice LUTs, 84.4% of DSP48Es and 79.5% of BRAMs. Re-synthesis is required if the input matrix or sub-matrix dimensions are changed.

Table 5.2: The usage of floating-point double-precision computational cores in numbers.

Modules	Factorization component	Preprocessing& Matrix Update	Total	Latency
Multipliers	1	32	33	9
Adders/Subtractors	4	48	52	12
Dividers	1	0	1	57
Squareroots	1	0	1	57

Table 5.3: On-chip memory usage in our hardware implementation.

Modules	Factorization Component	Preprocessing & Matrix update	Others
$FIFOs$	14	17	16
$36kBRAMs$	14	31	165
$18kBRAMs$	0	1	1

5.7.2 Performance analysis

In Fig. 5.5, the clock cycles that are used to perform QR Decomposition on various dimensional data sets with different matrix partition strategies are demonstrated, which indicates the matrix partition would improve the parallelism for performing QR factorization; however, the partition strategy plays significant roles on the performance of the hybrid approach. As

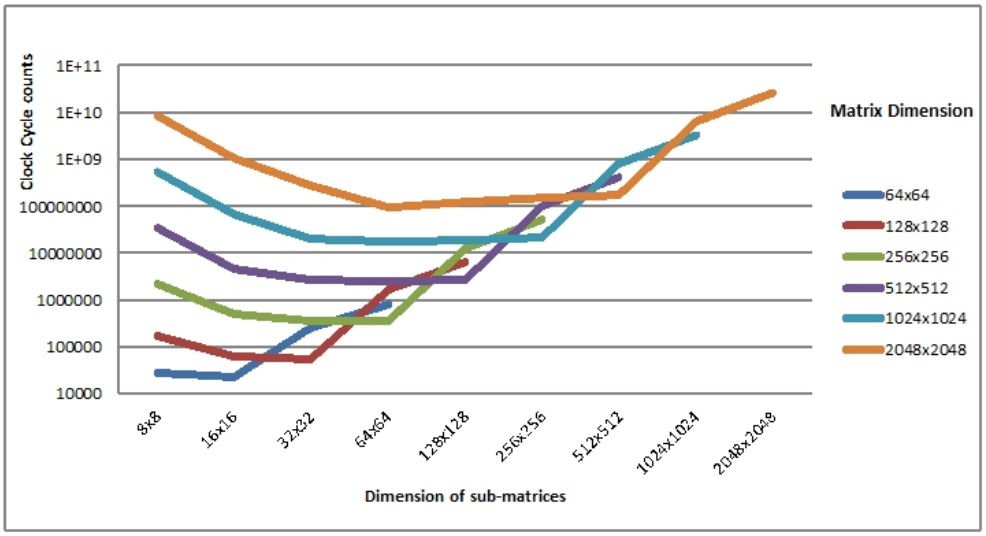


Figure 5.5: The clock cycle counts for our QR Decomposition computing on matrices with various dimensions and partitions.

it is shown in Fig. 5.5, different sized matrix have to be applied with their respective appropriate matrix partition strategy to maximize the efficiency, which is impact by many factors such as the data-dependencies, computational latency of parameter generation and matrix updates. To partition the matrix into tall-skinny sub-matrices could achieve better efficiency; however, the key criteria is to balance the computation of Householder Transformation and Givens Rotations.

We use GFLOPS (Giga FLoating-point Operations Per Second) as our metric to compare dimensional and partition-dependent performance. In Fig. 5.7a, the performance of our QR Decomposition architecture running at a frequency of 150 MHz is demonstrated, in which different dimensional matrices are applied and various partition strategies are evaluated. Dimensional peak performance is achieved with partitioned sub-matrices of different sizes. Parallelism is able to reduce the idle time of floating-point cores caused by the latencies of accumulation due to vector lengths greater than the number of multipliers used for calculating the norms (16 in our case). Thus, the number of partitions in one sub-matrix column dictates efficiency. In our implementation, at least 16 sub-matrices are needed in a sub-matrix column for efficient computation. This number is the same as the latency of the floating-point accumulator. Processing a sub-matrix column may introduce idle time for computational cores as the number of

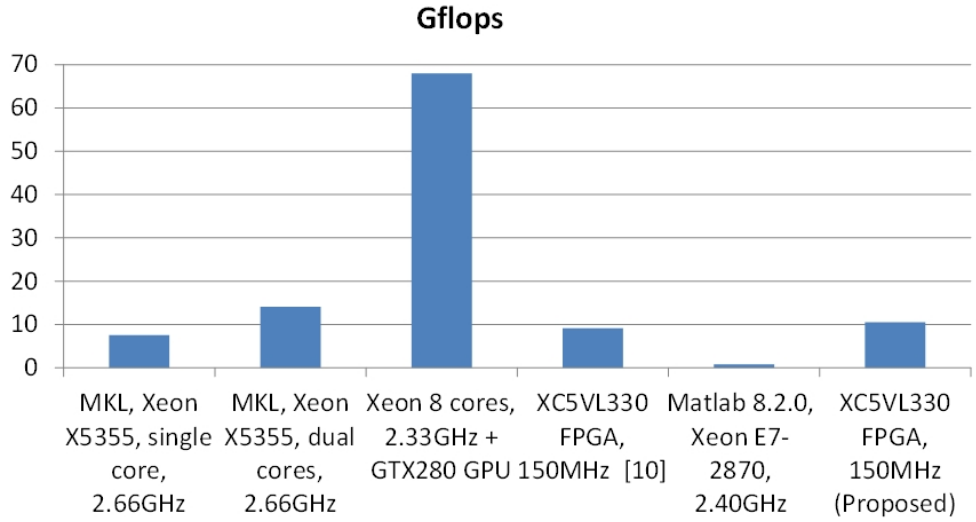
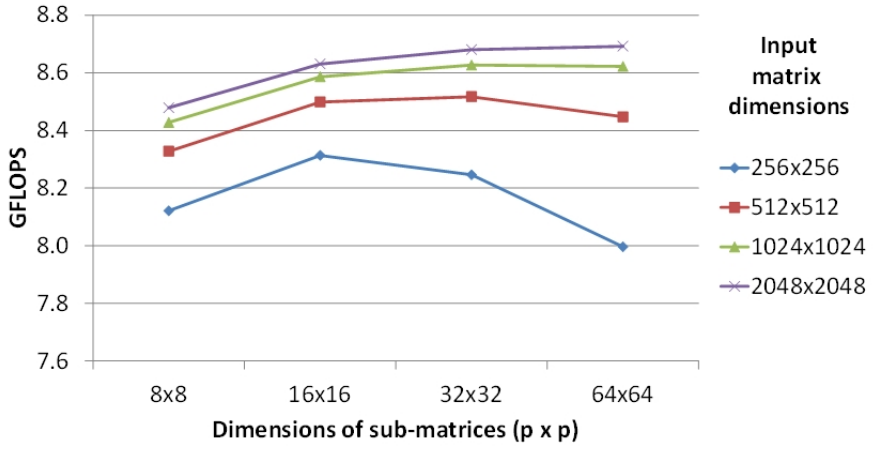


Figure 5.6: Performance comparison with single core, multi-core, GPU and recent FPGA work.

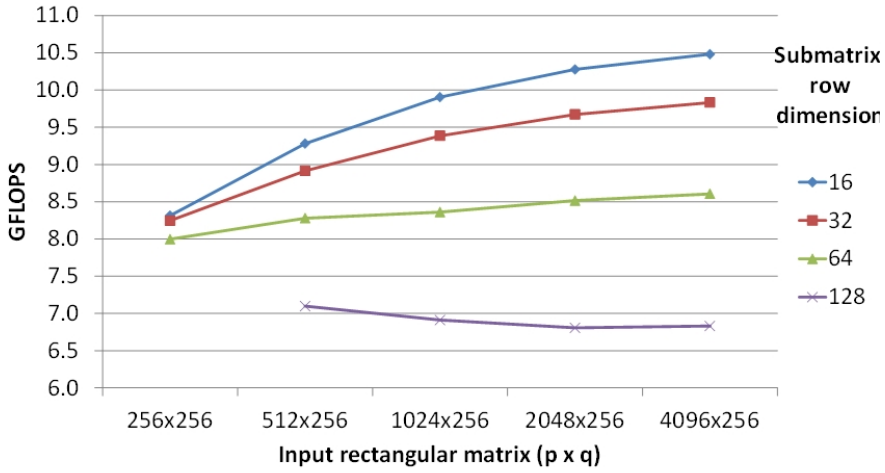
elements needing processing decreases. Therefore, to obtain the best dimensional performance for square matrices, the number of sub-matrix columns has to be minimized, while sufficient parallelism is maintained.

In Fig. 5.7b, rectangular matrices are evaluated, in which the number of sub-matrices in a sub-matrix column is equal to that in a sub-matrix row. As row dimension increases, the rectangular sub-matrix becomes more skinny, thus performance improves as more parallelism is achieved, while the remaining elements after the Householder transformation is reduced.

We have also analyzed the average I/O requests for our QRD hardware implementation with input matrix partitioned into 16×16 sub-matrices (see Fig. 5.8). Due to the limited on-chip resources, not all the intermediate data can be stored on chip when the matrix dimension grows over 512×512. When intermediate data are hold on-chip, the average I/O requests will be reduced as the column dimensions grow, due to the increase of the latency for parallel Householder transformations. The use of off-chip memory for intermediate data would increase the number of I/O requests, and the system performance will be degraded when the required amount of I/O communications surpass the provided bandwidth of experimental platform (16 in our case).



(a) square matrices with square sub-matrices



(b) rectangular matrices with rectangular sub-matrices

Figure 5.7: The performance of our architecture for performing QR Decomposition on square and rectangular matrices.

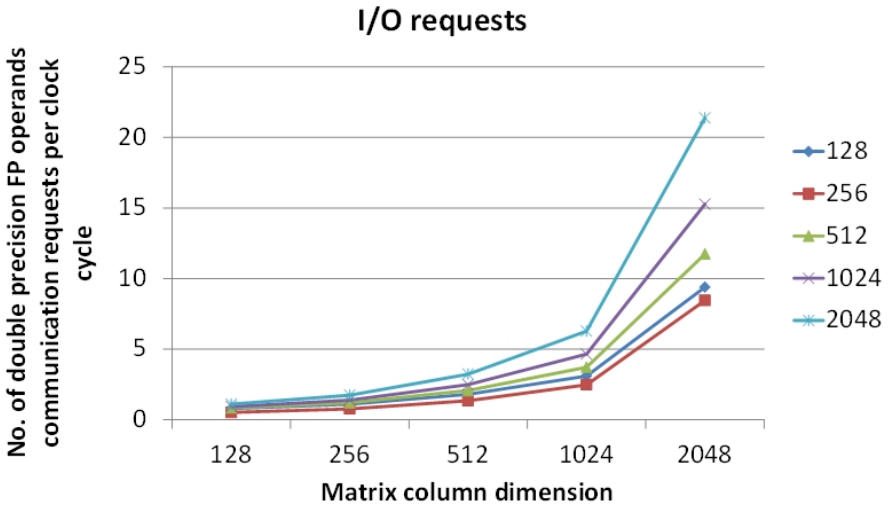


Figure 5.8: The average No. of double precision floating point operands communication requests per clock cycle for input matrix with various dimensions.

We compare the performance of our design with the results of CPU based MKL implementations [Buttari et al. (2009)], a GPU implementation [Tomov et al. (2010)], a Matlab routine and a recent FPGA work [Tai et al. (2011)] in Fig. 5.6. Our design shows speedups of up to 1.46×, 1.15× and 13.75× compared to the MKL implementation on a single core [Buttari et al. (2009)], FPGA-based tiled matrix decomposition [Tai et al. (2011)] and Matlab routine respectively. We are able to achieve 10.5 GFLOPS, which is 80.5% of the theoretical maximum computation throughput of our design (i.e. 87 floating-point cores × 150 MHz = 13.05 GFLOPS). In [Rafique et al. (2012)], they have demonstrated the performance of FPGA, GPU and CPU (MKL) implementations, although their implementation shows better performance for their FPGA-based tall-skinny matrix QR decomposer, a Virtex-6 SX475T is employed which provides a much higher clock frequency and more computing resources than our platform.

5.8 Conclusion

A reconfigurable architecture is proposed and implemented on an FPGA-based platform to perform QR Decomposition, which exploits both the advantages of the Householder transformation in its efficient vectorized computation and the Givens rotation in its flexible and highly parallel operations. The architecture can be configured to perform either the Householder

transformation or the Givens rotation at runtime, in which the Householder transformations are employed to transform the sub-matrices from dense to triangular, while the Givens rotation is used to zero out the remaining unneeded non-zero elements. Our experimental results show our design can achieve a performance of 10.5 GFLOPS with speedups of up to 1.46×, 1.15× and 13.75× compared to a MKL implementation, a recent FPGA design and a Matlab solution respectively. For future work, we plan to explore potential applications of our architecture (e.g. beamforming, image recovery).

CHAPTER 6. A CONFIGURABLE ARCHITECTURE FOR SPARSE LU DECOMPOSITION ON MATRICES WITH ARBITRARY PATTERNS

Modified from a paper published in

Proceedings of 2015 *International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART)*

Xinying Wang[1], Phillip Jones and Joseph Zambreno [2]

6.1 Abstract

Sparse LU Decomposition has been widely used to solve sparse linear systems of equations found in many scientific and engineering applications, such as circuit simulation, power system modeling and computer vision. However, it is considered a computationally expensive factorization tool. While parallel implementations have been explored to accelerate sparse LU Decomposition, irregular sparsity patterns often limit their performance gains. Prior FPGA-based accelerators have been customized to domain-specific sparsity patterns of pre-ordered symmetric matrices. In this paper, we present an efficient architecture for sparse LU Decomposition that supports both symmetric and asymmetric sparse matrices with arbitrary sparsity patterns. The control structure of our architecture parallelizes computation and pivoting operations. Also, on-chip resource utilization is configured based on properties of the matrices being processed. Our experimental results show a 1.6 to 14× speedup over an optimized software implementation for benchmarks containing a wide range of sparsity patterns.

[1]Primary researcher and author
[2]Correspondence author

6.2 Introduction

Many important scientific and engineering applications (e.g. circuit simulation [Chen et al. (2015, 2013)], power system modeling [Yu and Wang (1990)], and image processing [Donoho and Tsaig (2008)]) have at their core a large system of sparse linear equations that must be solved. Sparse LU Decomposition has been widely used to solve such systems of equations, but it is considered a computationally expensive factorization tool.

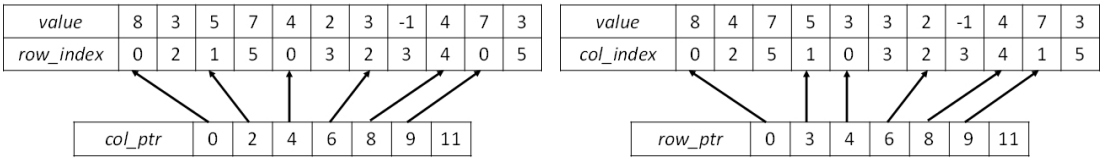
Left-looking, Right-looking and Crout are the main direct methods for sparse LU Decomposition, but are not efficient when implemented in software. Supernodal [Cleveland Ashcraft et al. (1987)] and Multifrontal [Duff and Reid (1983)] are approaches that lend themselves well to software implementations of sparse LU Decomposition. Supernodal considers the input matrix as sets of continuous columns with the same nonzero structure, while Multifrontal organizes large sparse datasets into small dense matrices. Parallel implementations of Supernodal and Multifrontal have been demonstrated on multi-core platforms and GPUs with shared or distributed memory [Chen et al. (2013, 2015); Demmel et al. (1999); Gårtner (2004)]. However, Supernodes may not exist in some applications, and it is a challenge to parallelize the Multifrontal method in a distributed memory system.

Although many FPGA-based architectures have been proposed for accelerating LU Decomposition of dense matrices, only a few have been proposed for accelerating sparse matrix LU Decomposition [Kapre and DeHon (2009); Siddhartha and Kapre (2014a,b); Vachranukunkiet (2007); Wu et al. (2012)]. Of these architectures, they either target domain-specific sparsity patterns [Kapre and DeHon (2009); Siddhartha and Kapre (2014a,b); Vachranukunkiet (2007)], or require a pre-ordered symmetric matrix [Wu et al. (2012)].

In this paper, we propose an FPGA-based architecture for sparse LU Decomposition, which can efficiently process sparse matrices having arbitrary sparsity patterns. Our architecture mitigates issues associated with arbitrary sparsity patterns and extracts parallelism in two primary ways. First, it factorizes columns from the lower triangular part of a matrices in parallel with factorizing rows from the upper triangular part of the matrix. Second, our control structure performs pivoting operations in parallel with the factorizations of rows and columns.

$$\begin{pmatrix} 8 & 0 & 4 & 0 & 0 & 7 \\ 0 & 5 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 \\ 0 & 7 & 0 & 0 & 0 & 3 \end{pmatrix}$$

(a) Example sparse matrix



(b) Compressed Sparse Column format

(c) Compressed Sparse Row format

Figure 6.1: Compact storage formats for sparse matrices

Our experimental results show a 1.6 to 14× speed up over an optimized software implementation for benchmarks containing a wide range of sparsity patterns.

6.3 Theoretical background

6.3.1 Sparse LU Decomposition with pivoting

LU Decomposition factorizes a matrix A into two matrices, L and U, as shown in eq. (6.1)

$$A=LU \tag{6.1}$$

Here, A is an $m \times n$ matrix, L is an $m \times n$ lower triangular matrix, and U is an $n \times n$ upper triangular matrix. This linear process is called sparse LU Decomposition when matrix A is sparse.

Sparse matrices commonly appear in a broad variety of scientific and engineering applications. These matrices are characterized by having relatively few non-zero elements. This property can be leveraged to store them in efficient formats. The two most popular of these formats are Compressed Sparse Column (CSC) format and Compressed Sparse Row (CSR). Fig. 6.1 illustrates the CSC and CSR format of a sparse matrix. Both CSC and CSR formats consist of three components: 1) an array of non-zero values, 2) an integer array of row or col-

umn indexes of those non-zero elements, and 3) an array of pointers where each element points the first non-zero element of a column or a row.

Stability and Accuracy. To ensure stability during LU Decomposition, pivoting operations are performed to remove zero elements from the diagonal of matrix A. This can be accomplished conceptually by applying a pivoting matrix, P, to matrix A as $PA{=}LU$. P is an $m \times m$ matrix in which each column has one element of value 1 and all other elements are 0 [Golub and Van Loan (1996)].

6.3.2 Algorithms for sparse LU Decomposition

Direct methods for sparse LU Decomposition include Left-looking, Right-looking, and Crout [Press et al. (2007)], which generally contain division operations and update processes (see Fig. 6.2).

6.3.2.1 Left-looking

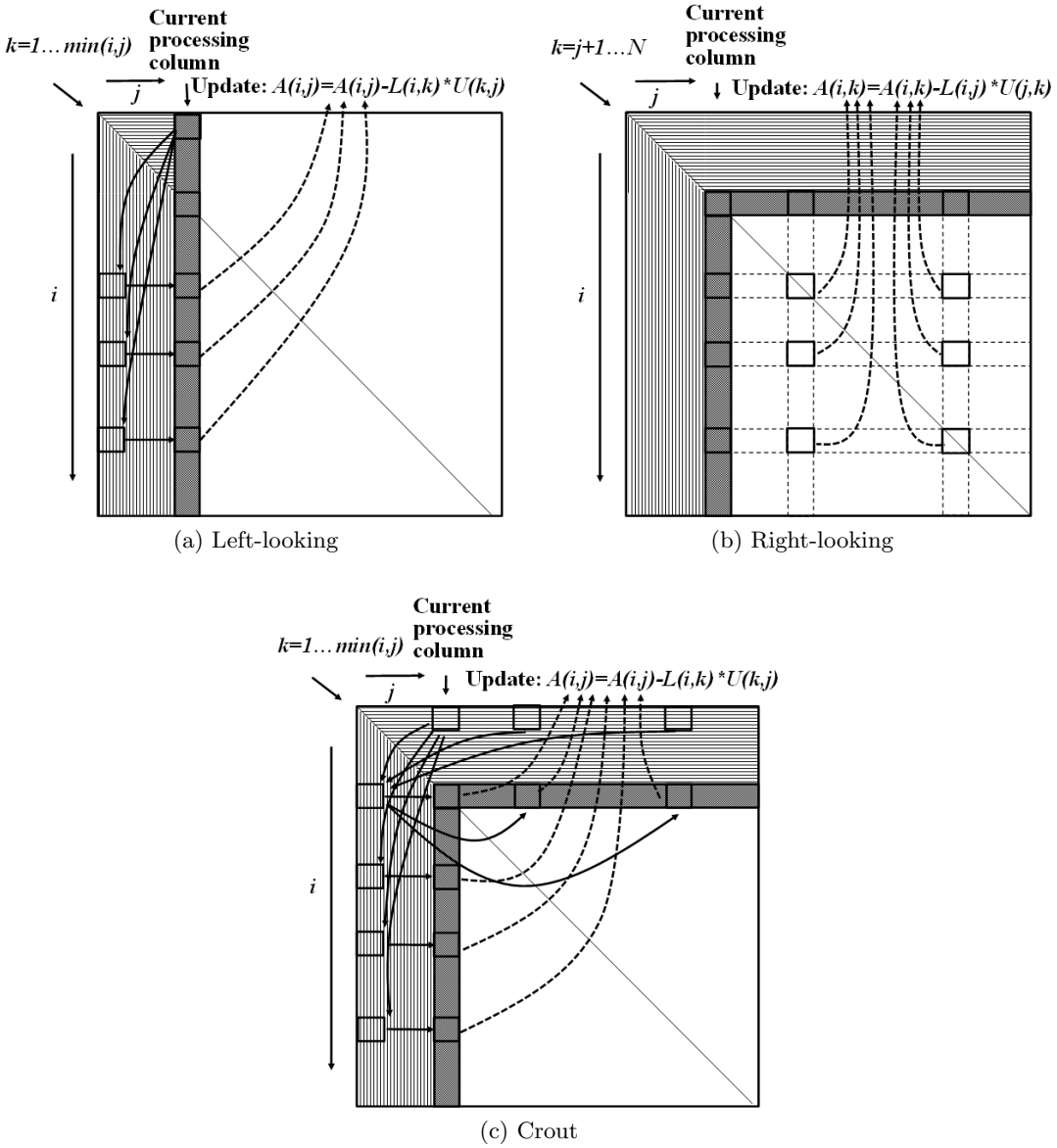
The Left-looking algorithm factorizes a matrix in a column-by-column manner. Before normalizing a given column, non-zero elements of the previously factored column are used to update the current column elements $A(i,j)$ using the equation $A(i,j) = A(i,j) - L(i,k) * U(k,j)$, where $k = 1 \cdots \min(i,j)$ (see Fig. 6.2a).

6.3.2.2 Right-looking

The Right looking algorithm first factorizes a column from the lower triangular part of a matrix, then uses the resulting non-zero elements of that column to update the affected components in the rest of the matrix by using the equation $A(i,k) = A(i,k) - L(i,j) * U(j,k)$, where $k = j+1 \cdots N$, j is the index of current factored column, and N is the column dimension of matrix A (see Fig. 6.2b).

6.3.2.3 Crout

Similarly to the Left-looking algorithm, the Crout method performs updates with previously factored elements before normalizing a given vector. The difference is that the Crout method



(a) Left-looking

(b) Right-looking

(c) Crout

Figure 6.2: Popular algorithms for sparse LU Decomposition

operates both on columns and rows, while the Left-looking algorithm only operates on columns (see Fig. 6.2c).

6.4 Related work

FPGA architectures have been shown to be effective in accelerating a wide range of matrix operations. However, accelerating the LU Decomposition of large sparse matrices with arbitrary sparcity patterns is a challenge for hardware acceleration. In [Vachranukunkiet (2007)], the authors propose an efficient sparse LU Decomposition architecture targeting the power flow analysis application domain. Their FPGA-based architecture implements the right looking algorithm and includes hardware mechanisms for pivoting operations. The performance of their architecture is primarily I/O bandwidth limited. Kapre et al., in [Kapre and DeHon (2009); Siddhartha and Kapre (2014a)], introduced an FPGA implementation of sparse LU Decomposition for the circuit simulation application domain. A matrix factorization compute graph is generated to capture the static sparsity pattern of the application domain, and it is exploited to distribute the explicit data flow representation of computation across processing elements. For their approach, they also illustrate that their performance gains are highly sensitive to the manner in which computations are distributed across the available processing elements. Wu et al., in [Wu et al. (2012)], devised a more general hardware design to support sparse LU Decomposition for a wider range of application domains. Their architecture parallelizes the left-looking algorithm to efficiently support processing symmetric positive definite or diagonally dominant matrices. One factor limiting the performance of their architecture arises from dynamically determining data dependency during their column-by-column factorization, which leads to their processing elements stalling for the purpose of synchronizing to resolve data dependency across processing elements.

6.5 The parallel sparse LU Decomposition algorithm

In general, the process of LU factorization primarily consists of pivot, division, and update operations. These operations can be performed in parallel when no data dependencies exist

```
1: [m n] ← size(A);
2: for i ← 1 to min m, n do
3:    {1.Perform pivoting operations in parallel};
4:    A(i, i) ← max(A(:, i));
5:    for j ← 1 to n do
6:       A(i, j) ↔ A(max_index, j);
7:    end for
8:    {2.Update column entries before division in parallel};
9:    for j ← i to m do
10:      if F_L(j, k) ∈ block(pid_row) then
11:         A(j, i) ← A(j, i) − F_L(j, i);
12:      end if
13:   end for
14:   {3.Parallelized division operations};
15:   for j ← i + 1 to m do
16:      if A(j, i) ≠ 0 then
17:         L(j, i) ← A(j, i)/A(i, i);
18:      end if
19:   end for
20:   {4.Update row entries after division in parallel};
21:   for k ← i + 1 to n do
22:      if F_U(j, k) ∈ block(pid_col) then
23:         U(i, k) ← A(i, k) − F_U(i, k);
24:      end if
25:   end for
26:   {5.Calculate Update factors in parallel}
27:   for z ← 1 to i do
28:      for j ← z + 1 to m(or n) do
29:         for k ← i + 1 to i + 1 + block_size do
30:            if L(j, i) ≠ 0 and U(i, k) ≠ 0 and F_L(j, k)(or F_U(j, k)) ∈ block(pid_row( or
               pid_col)) then
31:               F_L(j, k) = F_L(j, k) + L(j, i) ∗ U(i, k)
32:               (or F_U(j, k) = F_U(j, k) + L(j, i) ∗ U(i, k))
33:            end if
34:         end for
35:      end for
36:   end for
37: end for
```

Algorithm 2: Our parallel sparse LU Decomposition algorithm

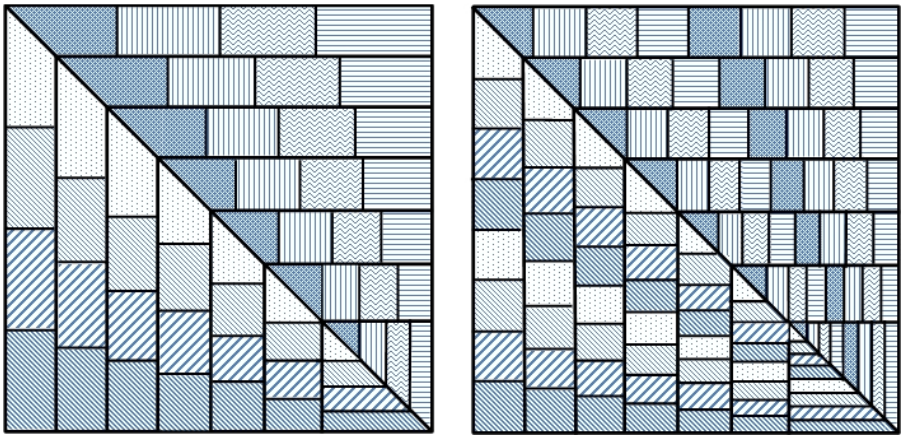


Figure 6.3: Two examples of block partitioning.

among them. Our architecture aims to extract this type of parallelism specifically from the Crout method of sparse LU Decomposition at three different levels: 1) we leverage the fact that the structure of the Crout method naturally allows parallelization of processing columns and rows from the lower and upper triangular part of a matrix respectively (Fig. 6.2c), 2) we perform block partitions of the matrix to identify elements for which update operations can be performed in parallel and thus share an update processing element (see Fig. 6.3, the computations of matrix elements in the blocks with identical shaded pattern can be assigned to the same update processing element), and 3) the structure of the architecture control logic performs pivot operations in parallel with update and division operations.

6.6 The sparse LU Decomposition architecture

Our architecture consists of four primary types of processing elements: 1) Input, 2) Update, 3) Division, and 4) Pivot. Fig. 6.4 provides a high-level view of how these processing elements are related within our architecture.

As a brief summary of the flow of data through our architecture, the Input processing elements are responsible for the initial processing of columns from the lower triangular part of the matrix and rows from the upper triangular part of the matrix. The output of the Input processing elements is then forwarded to the Update processing elements, which implement a major portion of the Crout method's computation. After a set of entries within a column of

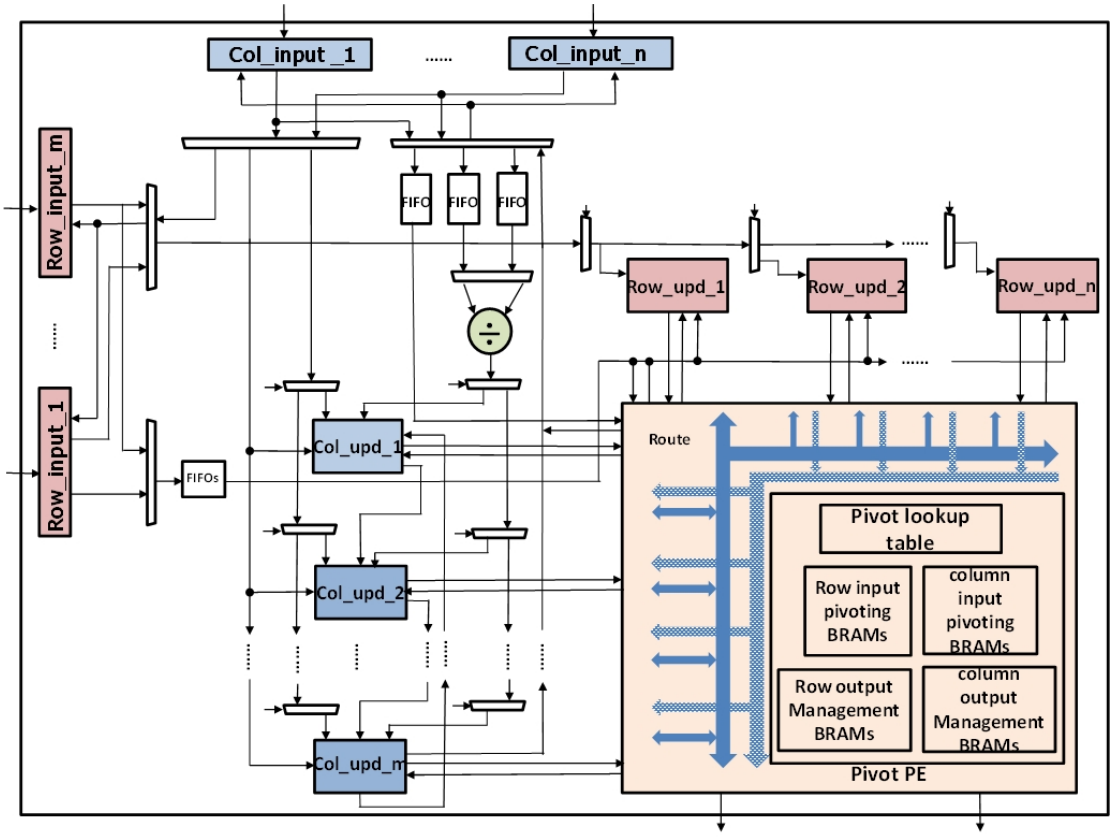


Figure 6.4: The block diagram of our sparse LU Decomposition architecture

the lower triangular part of the matrix have been updated, the Division processing element normalizes these entries with respect to the matrix's diagonal element associated with that column. The Pivot processing element directs movement between Update processing elements in a manner that results in the pivoting of matrix elements. Additionally, the Pivot processing element manages the output data stream.

The number of processing elements used for our architecture is configurable based on three factors 1) amount of on-chip resources, 2) matrix block partitioning strategy used, and 3) matrix properties (e.g. matrix dimensions, sparsity rate).

6.6.1 Input

The Input PEs have two variations. One is for initially processing columns from the lower triangular part of the matrix, and one is for processing rows from the upper triangular part of

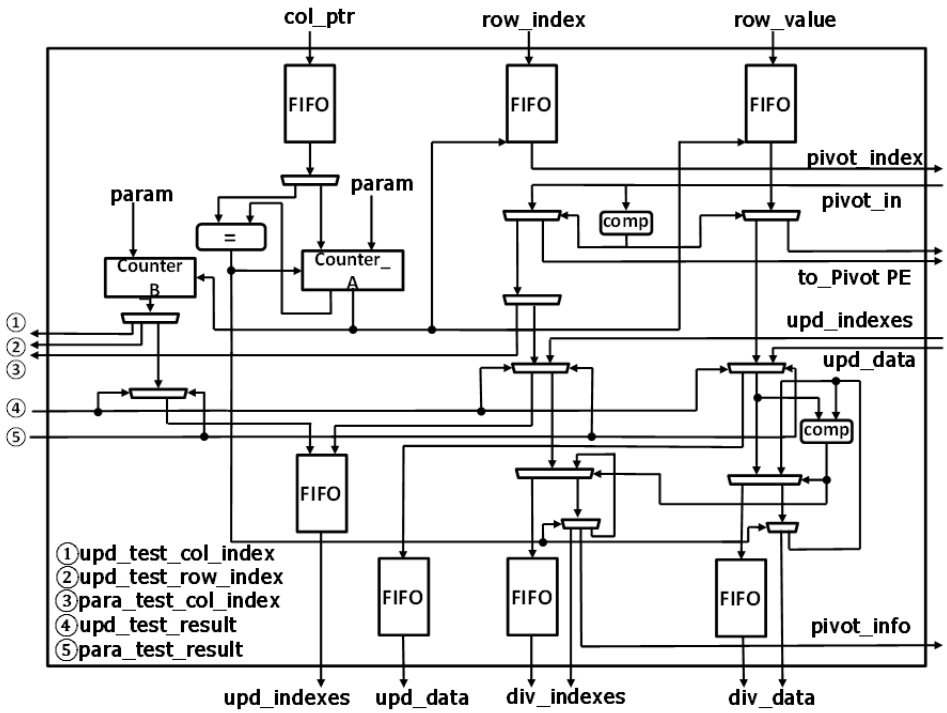


Figure 6.5: Input PE architecture.

the matrix. As shown in Fig. 6.5, the architecture of these two are similar, with the column version having some additional logic.

The additional logic for column processing serves three purposes: 1) it reduces the amount of processing by detecting if the update of an element involves multiplication by zero, 2) it determines if updating the current element will impact updating elements to be processed in adjacent columns from the upper triangular part of the matrix, and 3) it obtains the pivot information for each column by locating the element with the largest value.

The functionality that the two variations of the Input PE share is that based on the row (column) index information received from the input, either 1) the element of the row (column) read in will be routed to a row (column) Update processing element, or 2) if the element is to be pivoted from the lower triangular part of the matrix to the upper triangular part of the matrix or visa-verse, then the element is directed to the Pivot PE, which will be responsible for forwarding the element to the proper Update PE in the future. Additionally, both Input PE types take as input matrix row (column) information formatted as CSR (CSC).

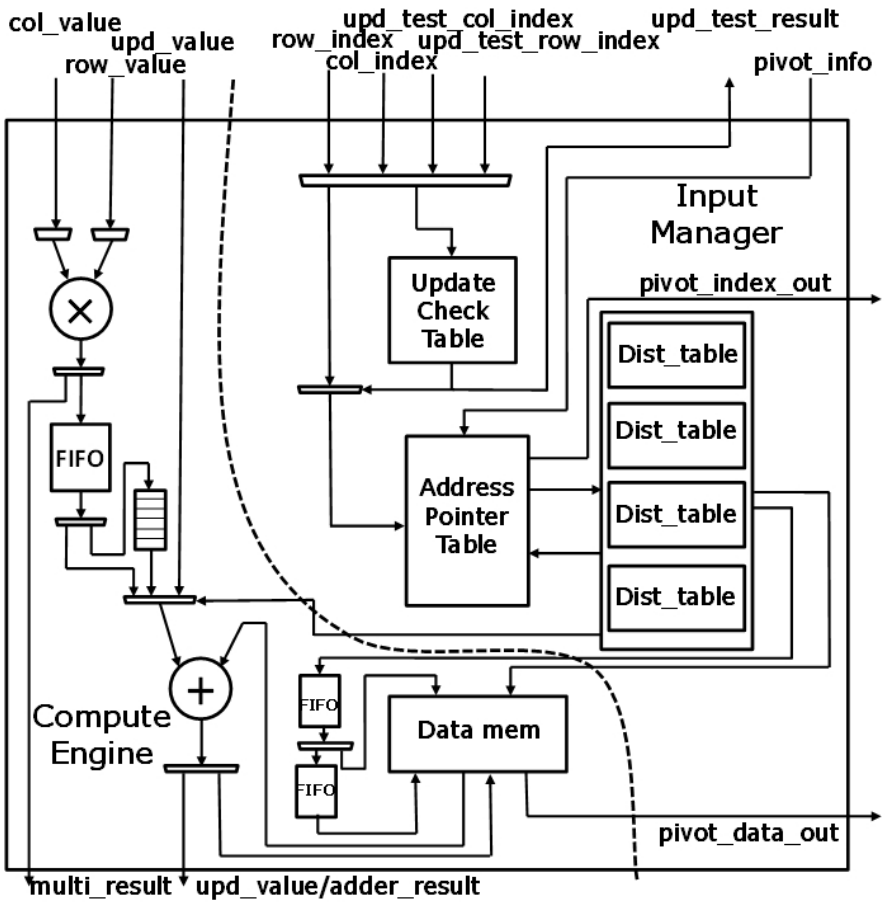


Figure 6.6: Update PE architecture.

6.6.2 Update

The Update PEs are responsible for helping compute the core update equation of the Crout method, $A(i,j) = A(i,j) - L(i,k) * U(k,j)$, where $k = 1 \cdots \min(i,j)$. It is composed of two primary components: 1) an Update Compute Engine, and 2) an Update Input Manager. Fig. 6.6 gives a block-level diagram of the Update PE.

The Update Compute Engine performs multiply accumulate operations over rows (columns) that are currently being updated. It is fed appropriate values from the Update Input Manager. The Update Input Manager provides two services. Firstly, it manages what we call an "Update Check Table" to indicate if a given element of a row or table needs to be updated. Secondly, it maintains a data structure (using Address Pointer Table and $Dist\_Table$) that keeps track of addresses of non-zero values that are stored in Data mem of Fig. 6.6. These are the values fed to the Update Compute Engine from Data mem.

Once a matrix element has been updated by the Update PE, then if it is associated with the lower triangular part of the matrix its value is normalized with respect to matrix's diagonal element associated with that element's column.

6.6.3 Pivot

As indicated in Section 6.3.1, pivot operations are required to ensure numerical stability during sparse LU Decomposition. The Pivot PE (see lower right side of Fig. 6.4) performs pivots by acting as a router between the lower triangular and upper triangular part of the matrix. Based on an element's ⟨row, column⟩-index information, when read into an Input PE, the Pivot PE determines if that element should pivot (i.e. be transferred from the lower to upper triangular part of the matrix or visa-verse). Lookup tables within the Pivot PE are used to store and evaluate ⟨row, column⟩-index information to determine if and where an element should be pivoted. The Pivot PE is also responsible for buffering and sending elements to off-chip storage that have been completely processed. In other words, an element's value is stored off-chip when it can no longer be affected by the processing of future elements.

6.7 Experiments and evaluations

6.7.1 Implementation and experimental setup

Our architecture was evaluated using a Convey Computer HC-2 system [Convey Computer HC-2 (2012)]. The HC-2 is a hybrid-core system that couples a standard Intel based server with a co-processor board that hosts four Xilinx Virtex-5 XC5VLX330 FPGAs and a special-

ized memory subsystem for high throughput random memory access. We implemented our architecture for one of the HC-2's Virtex-5 FPGAs, referred to by Convey as an Application Engine (AE).

All floating-point operations were performed using double-precision computing cores generated with Xilinx's Coregen [Xilinx Inc. (2012)]. The pipeline latency for each type of core used in our design was: 9, 14, and 57 clock cycles for multiplication, addition, and division, respectfully. The modular structure of our design allows us to easily customize the number and types of PEs implemented to best suit the properties of the matrix being processed. For example, for a "skinny" and "tall" matrix we implement more Update PEs for processing columns than rows. The largest design that fits on the Virtex-5 LX330 consisted of 64 Update PEs. The FPGA resource utilization was 76.4% LUTs, 48.4% DSPs, and 87.5% BRAMs. It could be run at a maximum frequency of 176 MHz, which easily meets the 150 MHz frequency typically used for designs run using the HC-2.

Benchmarks were selected from the University of Florida sparse matrix collection [Davis and Hu (2011)] as workloads for our performance evaluation. As can be seen in Table 6.1, the selected benchmarks cover a wide range matrix types in terms of Dimension, Element pattern (e.g. symmetric, asymmetric, rectangular), and Sparsity rate. All matrices were split into upper triangular and lower triangular matrices and stored using CSR and CSC formats respectively.

6.7.2 Performance analysis

In this section, we first profile the sequential sparse LU Decomposition implementation on selected benchmarks. The profiling results (see Fig. 8.10) demonstrate that the percentage of time consumption for pivoting operations compared to entire execution time is determined by the matrix property and input data pattern. Comparably, higher percentage of execution time is used for pivoting operations in sparse LU Decomposition on symmetric matrix compared to that with asymmetric or rectangular datasets.

To analyze our design, we investigate how different architectural configuration parameters impact our design's performance. Then we evaluate the performance of our approach against

Table 6.1: Experimental benchmark matrices and their properties.

Matrix	Dimensions	Sparse rate	Pattern	Application domain
494_bus	494×494	0.68%	sym	Power network
Powersim	15838×15838	0.03%	sym	Power network
Msc01050	1050×1050	2.38%	sym	Structural problem
problem1	415×415	1.61%	sym	FEM
qc2354	2354×2354	7.22%	sym	Electromagnetic
West1505	1505×1505	0.24%	asym	Chemical process
CAG_mat1916	1916×1916	5.34%	asym	Combinatorial problem
lpi_chemcom	288×744	0.74%	rec	Linear programming
Well1850	1850×712	0.66%	rec	Least square problem
photogrammetry	1388×390	2.18%	rec	Computer vision

an existing FPGA implementation and against several software implementations, including Matlab (see Table 6.2).

Table 6.2: Experimental benchmark matrices and their performance.

Matrix	Dimensions	nnz(L+U)	Matlab performance (ms)	Our FPGA performance (ms)
494_bus	494×494	13,425	1.92	0.359
Powersim	15838×15838	136,472	19.7	12.3
Msc01050	1050×1050	61,234	5.89	1.20
problem1	415×415	12,242	1.33	0.533
qc2354	2354×2354	926,209	652	107
West1505	1505×1505	42,688	31.3	13.7
CAG_mat1916	1916×1916	2,542,599	3920	279
lpi_chemcom	288×744	1,878	0.203	0.112
Well1850	1850×712	122,104	50.2	4.90
photogrammetry	1388×390	213,891	74.3	6.28

With respect to the impact of parameter settings on performance, we chose to examine two parameters: 1) the multiply-accumulate blocks size of the Update PE, where block size refers to the number of columns or rows processed as a batch while a previous column is being normalized using the division operator, and 2) the number of partitions the input matrix is

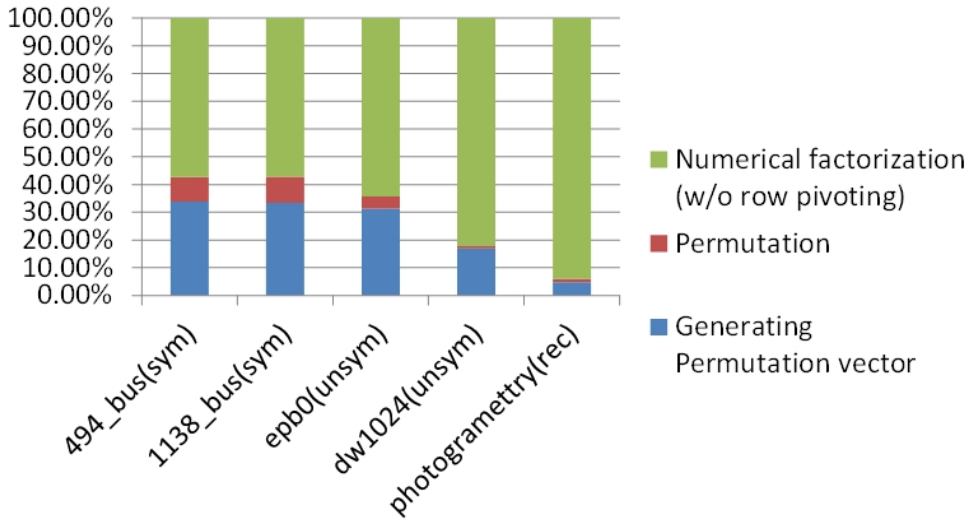
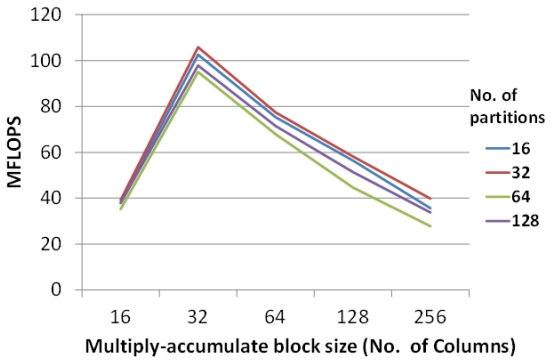


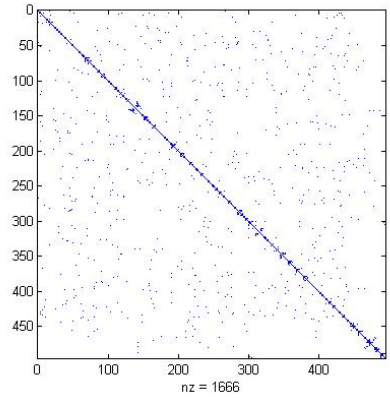
Figure 6.7: The profiling results of sparse LU Decomposition on selected benchmarks.

divided into. As Fig. 6.8 illustrates, multiply-accumulate block size has a significant impact on performance. Choosing a block size that is too big or too small causes up to a 2.5× difference in performance for the representative benchmarks shown. Deeper analysis showed that when block size was too large, the Update PE spends more time searching the block space for non-zero values, thus increasing multiply-accumulate stalls. When block size was too small, the multiply-accumulate engine of the Update PE will finish before the normalization process and will need to block until normalization completes. An interesting observation is that different benchmark matrices have different optimal multiply-accumulate block sizes. With respect to the number of partitions used to split up the input matrix, we observed a much smaller impact. In Fig. 6.9, the numbers of row pivoting operations are demonstrated by normalizing them to respective row dimensions of selected benchmarks, in which less row pivoting operations are required to maintain the numerical stability for symmetrical matrix compared to asymmetric or rectangular datasets, and a single row are more likely to be pivoted multiple times in operating rectangular matrix.

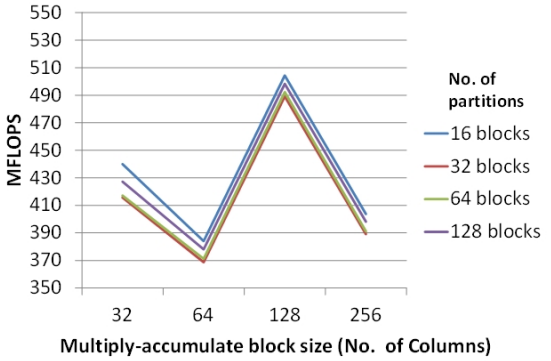
Figure 6.10 and 6.11 compares the performance of our architecture against others. When compared against the FPGA architecture of [Wu et al. (2012)], our throughput is 1.2 to 5× better. Additionally, while both architectures target acceleration across arbitrary application



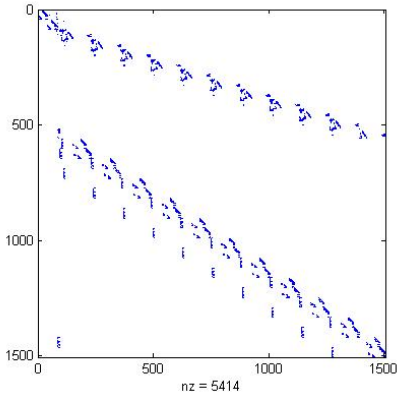
(a) 494_*bus* throughput



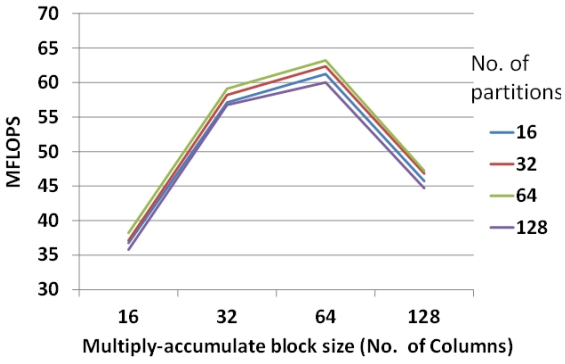
(b) 494_*bus* Sparcity pattern



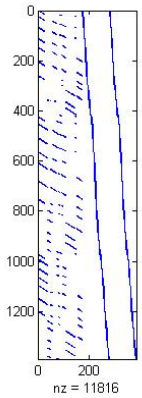
(c) *West*1505 throughput



(d) *West*1505 Sparcity pattern



(e) *photogrammetry* throughput



(f) *photogrammetry* Sparcity pattern

Figure 6.8: Impact of multiply-accumulate block size and No. of Input matrix partitions on performance.

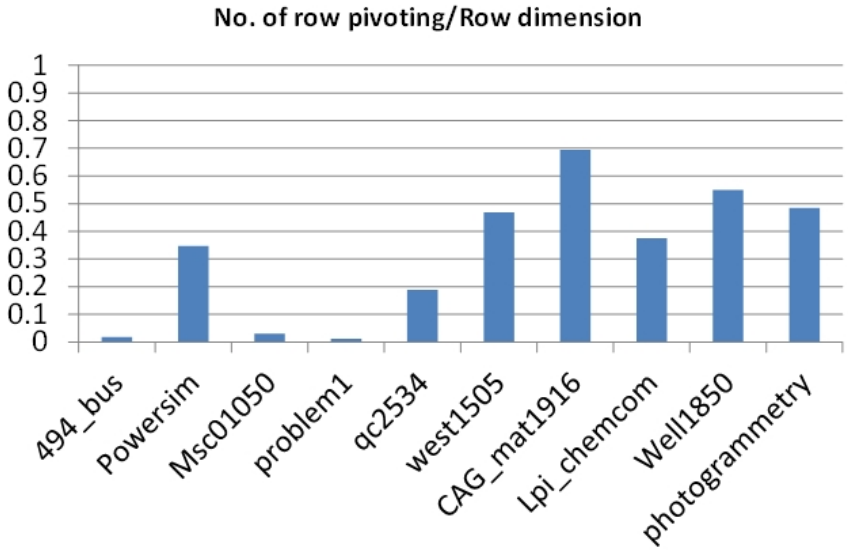


Figure 6.9: The No. of row pivoting operations compared row dimensions of selected benchmarks.

domains, our architecture does not require the input matrix to be reordered into a diagonally dominate matrix. When compared to the software approaches in [Wu et al. (2012)], where optimized linear solvers UMFPACK [Davis (2004)] and PARDISO [Gårtner (2004)] were used on a single-core (UMFPACK [Wu et al. (2012)]), single-core (PARDISO [Wu et al. (2012)]), and 4-core (PARDISO2 [Wu et al. (2012)]) CPU, we show a 1.2 to 75× speedup. A comparison with Matlab's LU Decomposition function (Fig. 6.11) run on a 2.2 GHz dual core Intel Xeon processor with 16GB of memory shows a speedup of 1.6 to 14×.

6.8 Conclusions

An FPGA-based architecture is presented that performs sparse LU Decomposition using a modified version of the Crout method. As opposed to targeting matrices with domain-specific sparsity patterns, it efficiently processes input matrices with arbitrary sparsity patterns, without requiring pre-ordering of the input matrix. For sparse matrix benchmarks having a wide range of properties, our experimental results show a 1.6 to 14× speedup over an optimized software solution.

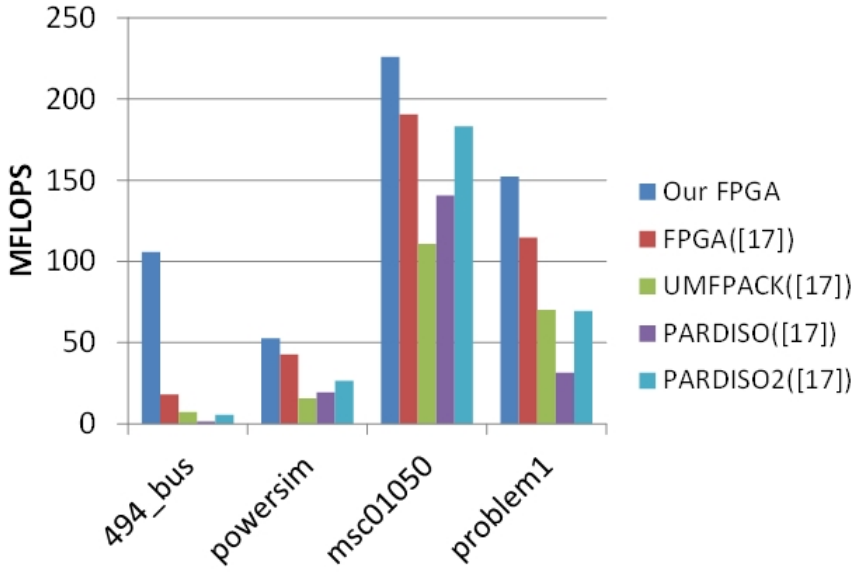


Figure 6.10: Throughput comparison between our architecture, and the FPGA and software implementations in [Wu et al. (2012)].

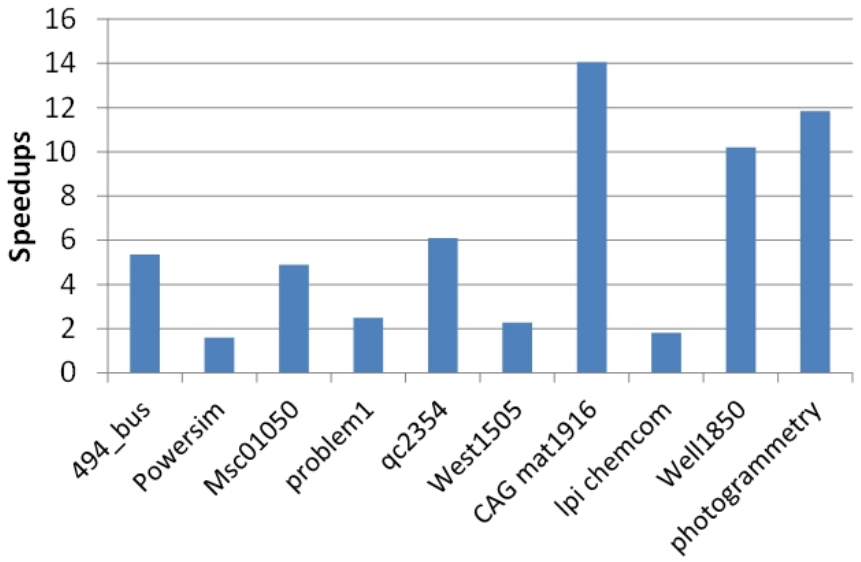


Figure 6.11: Speedups of our FPGA implementation normalized to Matlab's LU Decomposition routine.

CHAPTER 7. PARALLELIZING LATENT SEMANTIC INDEXING USING FPGA

Modified from a conference paper under submission

Xinying Wang[1] and Joseph Zambreno[2]

7.1 Abstract

Latent Semantic Indexing (LSI) has played a significant role in discovering patterns on the relationships between the query terms and unstructured documents. However, the inherent characteristics of complex matrix factorization in LSI make it difficult to meet stringent performance requirements, especially when analyze large-scale datasets. In this paper, we present a deeply pipelined reconfigurable architecture for LSI, which parallelizes the matrix factorization and dimensionality reduction, computation of cosine similarity between vectors, and the ranking of documents. Our architecture implements the reduced Singular Value Decomposition with Hestenes-Jacobi algorithm, in which both singular values and orthogonal vectors are collected, and its components can be reconfigured to update query vector coordinate and calculate query-document similarity. In addition, an ordered tree structure is used to reduce the matrix dimension and rank the documents. Analysis of our design indicates the potential to achieve a performance of 8.9 GFLOPS with dimension-dependent speedups over an optimized software implementation that range from 3.8× to 10.1× in terms of computation time.

[1]Primary researcher and author
[2]Correspondence author

7.2 Introduction

In many scientific and engineering applications (e.g., text processing [Gee (2003)], information retrieval [Maletic and Marcus (2000)], and bioinformatics [Vanteru et al. (2008)]), Latent Semantic Indexing (LSI) has been widely used as an information analysis technique to identify the relationships between the query terms and the content of unstructured documents. LSI commonly implements a linear factorization tool named reduced Singular Value Decomposition (rSVD) [Golub and Van Loan (1996)] at its core, followed by ranking process according to the cosine similarity between query terms and documents. LSI is considered as computationally expensive; as the data volume is growing continuously, the performance of sequential LSI implementation can not satisfy the expected efficiency of many applications, which requires the throughput of semantical retrieval over millions of documents within a few milliseconds per query [Majumdar et al. (2011)].

Software tools such as Gensim [Řehůřek and Sojka (2011)], SEMILAR [Rus et al. (2013)], LSAfun [Günther et al. (2014)] have been developed to perform Latent Semantic Analysis (LSA). However, the efficiency of sequential software implementations is relative low due to the inherent computational complexity of matrix factorization. To accelerate LSA, GPUs [Byna et al. (2010)] and multi-processor system with distributed memory [Okša and Vajteršic (2009)] are used to parallelize matrix factorization and/or vector operations. However, compared to multi-core or multi-processor systems, FPGAs have shown the promise to provide fine-grained parallelism [Herbordt et al. (2008)], which is more compatible with highly data-dependent transformations in SVD.

FPGA-based accelerators have been proposed for LSA [Majumdar et al. (2011); Eick et al. (2006); Cadambi et al. (2010); Graf et al. (2009)]. However, those implementations reformulated the computationally intensive portion of LSA as matrix or vector operations (e.g., multiplication, addition), instead of directly parallelizing SVD computation. Although speedups are achieved, the reconfigurability and flexibility of FPGAs have shown potentials to further improve the performance by ranking the documents in parallel with pipelined SVD computing and relative vector coordinates updates.

In this paper, we propose an FPGA-based architecture to accelerate LSI, which parallelizes the SVD computing, vector coordinates updates, cosine similarity calculation, and the process of ranking selected documents in an order of the calculated query-document cosine similarity. Our deeply pipelined reconfigurable architecture implements the reduced SVD with Hestenes-Jacobi algorithm [Wang and Zambreno (2014b)] that collects both the singular values and orthogonal vectors in the reduced k-dimensional space. Besides, the individual components of our architecture can be dynamically configured to update the query vector coordinates and calculate query-document cosine similarity. Additionally, an ordered tree structure is used to perform dimensionality reduction and rank the documents. Analysis of our design indicates the potential to achieve a performance of 8.9 GFLOPS with dimensional dependent speedups over an optimized software implementation that range from 3.8× to 10.1× in terms of computation time.

7.3 Theoretical background

7.3.1 Latent Semantic Indexing (LSI)

Latent Semantic Indexing [Deerwester et al. (1990)] is a mathematical technique to analyze the correlation between query terms and a collection of documents. It has been extensively used in many search engine applications to optimize the information retrieval. Traditionally, information retrieval is processed through lexically matching query terms with concepts in documents. However, its accuracy is impaired by synonymy that a given concept can be expressed in multiple ways, and polysemy that a word is able to convey many different meanings. To solve this problem, LSI introduces a method to retrieve information through matching the context of query terms and documents. The documents are ranked by query-document cosine similarity, which has no direct relationship with the number of shared terms.

To perform LSI, the documents and query terms are modeled by term-document matrix D and query vector q. D is an $m \times n$ matrix and q is a vector with a length of m, where m and n are the number of selected terms (key words) and documents respectively. An example of term-document matrix and query vector is given by Fig. 7.1.

1. ALU performs arithmetic and logic operations.
2. What operation the ALU to perform is told by the operation code.
3. ALU stands for arithmetic logic unit.
4. ALU is part of a computer processor.
Query terms: ALU, arithmetic, logic, operation(s).

(a) Example documents and query terms

	d1	d2	d3	d4		q
ALU	1	1	1	1		1
perform(s)	1	1	0	0		0
arithmetic	1	0	1	0		1
and	1	0	0	0		0
logic	1	0	1	0		1
operation(s)	1	2	0	0		1
what	0	1	0	0		0
the	0	2	0	0		0
to	0	1	0	0		0
is	0	1	0	1		0
told	0	1	0	0		0
by	0	1	0	0		0
code	0	1	0	0		0
stand(s)	0	0	1	0		0
for	0	0	1	0		0
unit	0	0	1	0		0
part	0	0	0	1		0
of	0	0	0	1		0
a	0	0	0	1		0
computer	0	0	0	1		0
processor	0	0	0	1		0

(b) Constructed term-document matrix D and query vector q

Figure 7.1: Example term-document matrix and query vector

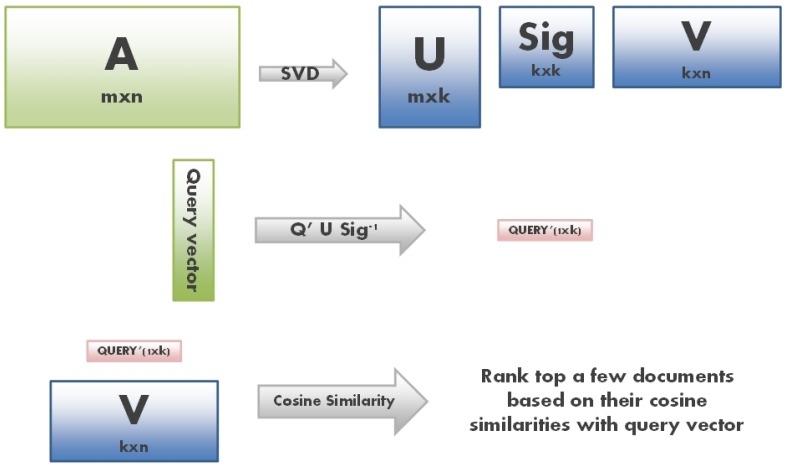


Figure 7.2: The process of Latent Semantic Indexing

To reduce the dimensions of the semantic space, the reduced Singular Value Decomposition (rSVD) is performed on the term-document matrix D in the form given by eq. (7.1)

$$D_{m\times n} \approx U_{m\times k}\Sigma_{k\times k}V^T_{k\times n} \qquad k < \min(m,n) \tag{7.1}$$

where U and V are orthogonal matrices and Σ is a diagonal matrix with singular values as its diagonal elements. Matrix V contains n documents coordinates in the reduced k-dimensional space. After rSVD, query vector q is updated to the new coordinates in the reduced k-dimensional space in the form of eq. (7.2).

$$q' = q^T U\Sigma^{-1} \tag{7.2}$$

The query-document cosine similarity between query vector and every document vector in matrix V in the reduced k-dimensional space is calculated in the form given by eq. (7.3)

$$sim(q,d_i) = \frac{q\cdot d_i}{\|q\|\cdot\|d_i\|} \tag{7.3}$$

where the values of $sim(q,d_i)$ are used to rank the documents for their association with query terms. A high cosine similarity indicates the close relationship between query term and document. However, it does not necessary mean a large number of terms shared by them. A diagram of general process for LSI is given by Fig. 7.2.

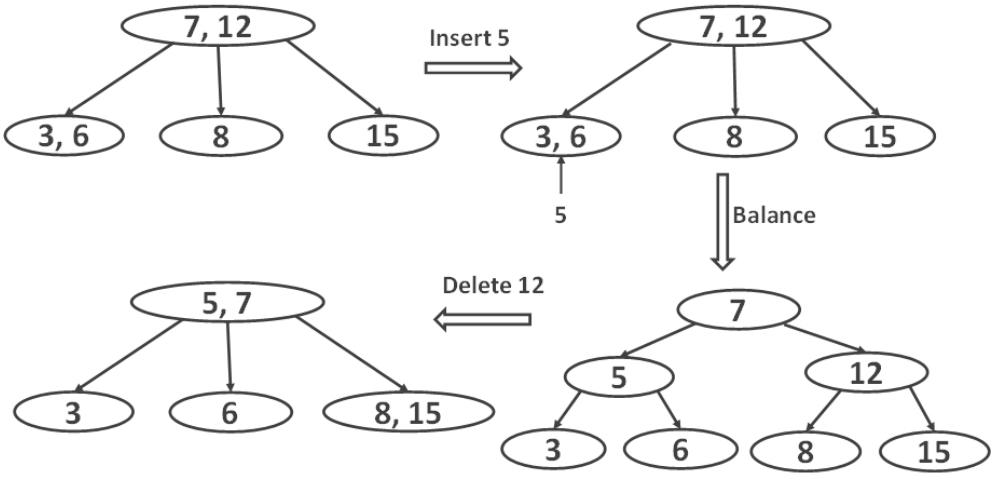


Figure 7.3: The 2-3 tree structure and operations

7.3.2 2-3 tree structure

Sorting is necessary as the final step of LSI to rank the documents based on their calculated cosine similarities with query vectors. Efficient sorting algorithms (e.g. merge sort, quick sort) or binary search trees (e.g. red-black tree, AVL tree) can be used for this purpose. In our case, we select $2-3$ search tree algorithm [Cormen et al. (2001)] to sort documents since it is considered as a balanced search tree that can be easily parallelized [Yang and Prasanna (2010)].

The $2-3$ tree data structure (see Fig. 7.3) is featured as a balanced and ordered search tree, whose node either has one element with two children or two elements with three children attached [Yang and Prasanna (2010)]. It can be seen as an equivalence of a binary search tree when every node has only one element. When an internal node has two elements p and q, the elements in its left and right children nodes are smaller and greater than both p and q respectively, while its middle children node contains the elements with the values between p and q ($p \leq q$).

Typical operations on $2-3$ tree include searching, insertion and deleting. As an ordered data structure, item can be located in the similar manner as searching in a binary search tree. Insertions are always taken place at the appropriate leaf nodes, and relative elements are repetitively moved from children to their parent node until all the 2-3 tree properties are satisfied. Meanwhile, operations that are required after removing an element depend on the position of that element in a 2-3 tree. Simple operations are presented in Fig. 7.3.

7.4 Related work

Latent Semantic Analysis (LSA) is a mathematical technique to extract the meaning of words with the contextual usage, and analyze relationships between documents and terms [Landauer et al. (1998)]. In the scope of LSA, Latent Semantic Indexing (LSI) [Deerwester et al. (1990)] is a popular solution for learning and ranking documents with query terms by modeling them in the semantic space. LSI and its alternatives pLSI [Hofmann (1999)] and LDA [Blei et al. (2003)] model the concepts of terms in a low dimensional semantic space, and then identify the correlations between them according to the cosine similarity (LSI) or probability (pLSI, LDA). In [Bai et al. (2009)], the supervised Semantic Indexing is presented with models are trained with (query, document) pairs of text documents.

Latent Semantic Analysis (LSA) is computationally expensive, and parallel implementations have been proposed to accelerate LSA computation. In [Cavanagh et al. (2009)], Cavanagh et al. presented a GPU-based implementation of LSA, in which they used a Lanczos algorithm to improve the efficiency of SVD through optimizing a series of matrix-vector operations. Byna et al. [Byna et al. (2010)] proposed a parallel implementation of supervised Semantic Indexing with customized best-effort computing strategies [Meng et al. (2009)] and data dependency relaxation on GPU. Okša et al, in [Okša and Vajteršic (2009)], introduced an LSI implementation on parallel computer with distributed memory, in which parallel two-sided block-Jacobi algorithm with dynamic ordering is used. Although GPUs and other modern parallel computer system have demonstrated the capability to accelerate LSI, FPGAs show potentials in further performance improvement, especially when the application has relatively sparse datasets.

Majumdar et al. proposed an FPGA-based accelerator for supervised Semantic Indexing [Majumdar et al. (2011)], in which FPGA provides a solution to parallelize a huge amount of dot products with fine granularity. Eick et al. [Eick et al. (2006)] use FPGA to accelerate Latent Semantic processing by mapping three compute-bound operations (document vector tokenization and calculation, documents and concept vectors mapping, and document vector scoring for ideal match retrieval) onto highly parallel platform. To improve the scalability, a parallel programmable learning and classification accelerator was presented by Cadambi et al. [Cadambi et al. (2010)], which uses on-chip memory for intermediate data, and banked off-chip memory with independent processing elements group assigned. Graf et al. [Graf et al. (2009)] implemented arrays of variable-resolution arithmetic vector processing elements (VPEs) on FPGAs to accelerate learning process, in which each group of VPE is connected to independent memory bank, and main data flows are kept locally for the purpose of low power dissipation. Most previous FPGA-based designs for Latent Semantic processing mainly focus on parallelizing simple matrix or vector computation. However, the flexibility and reconfigurability of FPGAs provide the potentials to parallelize advanced matrix decomposition, vector coordinates updates, and sorting operation simultaneously.

7.5 Algorithm for Latent Semantic Indexing

The LSI algorithm (see Algo. 4) takes the document-term matrix and query vector as the input, and output the ranked document sequence in the order of their association with query terms. To reduce the dimension of the input dataset, reduced SVD is performed on the document-term matrix with both left and right orthogonal matrices are collected (see Algo. 3). In our proposed LSI solution, the Hestenes-Jacobi algorithm is employed since it is featured with highly parallelizable vectored "reduce" and "map" operations [Wang and Zambreno (2014b)].

Hestenes-Jacobi algorithm is introduced based on the principle that annihilates individual element is equal to orthogonalize their respective column vectors. Thus, to conduct column-wise orthogonalization, squared column vector norms $N_{i,i}$ and the covariances among columns $N_{i,j}$ are first calculated, which is followed by iterative Jacobi rotations to zero out covariances. In this process, the calculated Jacobi rotation parameters are used to update affected squared

Input: matrix A and its dimensions m and n, matrix V
Output: Singular value vector Σ_i, orthogonal matrices U and V
`/* Calculate the squared 2-norms and covariances */`
for $i \leftarrow 1$ ***to*** n **do**
  for $j \leftarrow i$ ***to*** n **do**
    $N_{i,j} \leftarrow A_i^T * A_j$
  end
end
repeat
  for $i \leftarrow 1$ ***to*** $n-1$ **do**
    for $j \leftarrow i$ ***to*** n **do**
      `/* Calculate Jacobi rotation angle parameters */`
      $norm_1 \leftarrow N_{j,j} \quad norm_2 \leftarrow N_{i,i} \quad cov \leftarrow N_{i,j}$
      $\rho \leftarrow (norm_2 - norm_1)/(2 * cov) \quad t \leftarrow sign(\rho)/(|\rho| + \sqrt{1+\rho^2})$
      $\cos \leftarrow 1/\sqrt{1+t^2} \quad \sin \leftarrow cos * t$
      `/* Update the squared 2-norms */`
      $N_{j,j} \leftarrow N_{j,j} + t * cov \quad N_{i,i} \leftarrow N_{i,i} - t * cov$
      `/* Update the covariances */`
      for $h \leftarrow 1$ ***to*** $i-1$ **do**
        $N_{h,i} \leftarrow N_{h,i} * \cos - N_{h,j} * \sin \quad N_{h,j} \leftarrow N_{h,i} * \sin + N_{h,j} * \cos$
      end
      for $h \leftarrow i+1$ ***to*** $j-1$ **do**
        $N_{i,h} \leftarrow N_{i,h} * \cos - N_{h,j} * \sin \quad N_{h,j} \leftarrow N_{i,h} * \sin + N_{h,j} * \cos$
      end
      for $h \leftarrow j+1$ ***to*** m **do**
        $N_{i,h} \leftarrow N_{i,h} * \cos - N_{j,h} * \sin \quad N_{j,h} \leftarrow N_{i,h} * \sin + N_{j,h} * \cos$
      end
      `/* Update the left orthogonal matrix */`
      $$(A_{1:m,i}\ A_{1:m,j}) = (A_{1:m,i}\ A_{1:m,j}) \begin{bmatrix} \cos & \sin \\ -\sin & \cos \end{bmatrix}$$
      `/* Update the right orthogonal matrix */`
      $$(V_{1:n,i}\ V_{1:n,j}) = (V_{1:n,i}\ V_{1:n,j}) \begin{bmatrix} \cos & \sin \\ -\sin & \cos \end{bmatrix}$$
    end
  end
until *convergence reached*
for $i \leftarrow 1$ ***to*** $\min(m,n)$ **do**
  $\Sigma_i \leftarrow A_{i,i}$
end

Algorithm 3: Modified Hestenes-Jacobi SVD algorithm with orthogonal matrices collected

vector norms $N_{i,i}$, covariances $N_{i,j}$, and the related entries of the input matrix A and the right orthogonal matrix V. By running numerous iterations, the convergence can be acheived as the absolute value of covariance is smaller than the product of their respective column vector norms. Then, singular values can be collected as the squared root of those squared column norms. Left orthogonal matrix can be obtained with the updated input matrix A and singular values Sig as $U = A * Sig^{-1}$, while the resulted matrix V is the wanted right orthogonal matrix.

Input: Term-document matrix D and query vector q
Output: Ranked document sequence Rd
$[m\ n] \leftarrow size(D)$
`/* Perform SVD on matrix` D `(`$[U\ \Sigma\ V] \leftarrow$`svd`$(D)$`) (see Algo. 3) */`
$A \leftarrow D \quad V \leftarrow I$
$[A\ \Sigma\ V] \leftarrow$svd$(A)$
`/* Reduce to low rank k-SVD and update the query vector` q `*/`
$q' \leftarrow qA\Sigma^{-1}$
`/* Calculate cosine similarity among query and document vectors */`
for $i \leftarrow 1$ ***to*** n **do**
  $sim_i = \frac{q \cdot v_i}{\|q\| \cdot \|v_i\|}$
end
`/* Sort the resulted similarity sequence */`
$Rd \leftarrow sort(sim)$

Algorithm 4: Latent Semantic Indexing Algorithm

First k column and row vectors with largest associate singular values are selected from the left and right orthogonal matrices respectively. Then, query vector is updated with the dimensionality reduced left orthogonal matrix and singular values. As both the computation of left orthogonal matrix U and update of query vector q requires the inverse of square root operations, they can be eliminated by a single division. Cosine similarity between document vectors from the right orthogonal matrix and updated query vector q' are computed with the vector norms and covariances. The final resulted ranked sequence is generated by the sorting operation with calculated similarity, and in our design, we use a $2-3$ tree structure [Cormen et al. (2001)] to perform sortings.

In this algorithm, all the vectored computations can be parallelized, and they are shared either "reduce" structure to obtain the product of vectors for norms and covariances or "map" structure to update vector elements. Meanwhile, the $2-3$ tree structure [Cormen et al. (2001)]

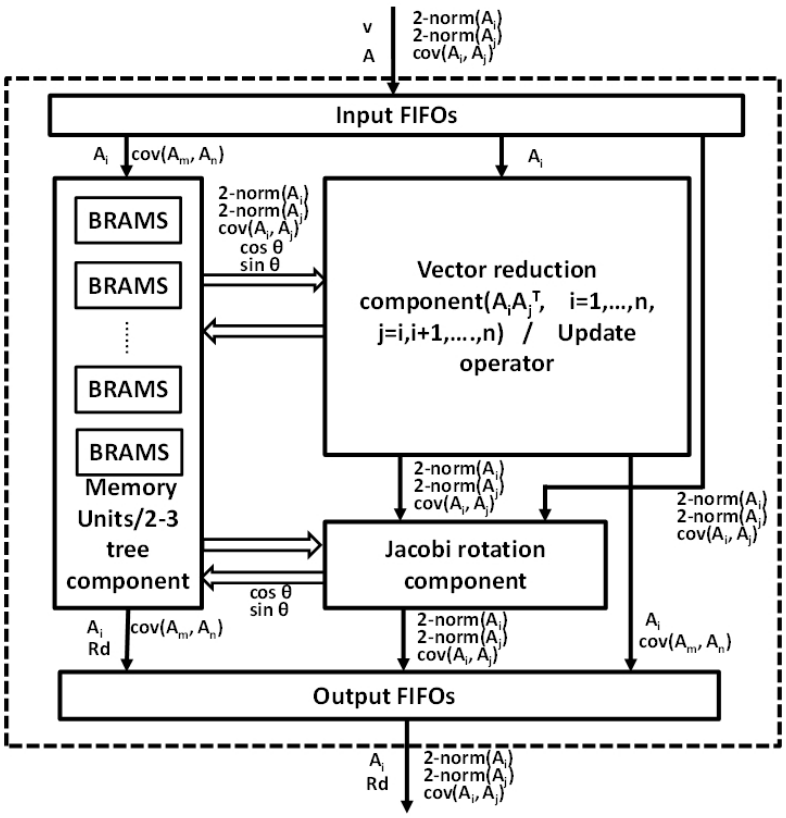


Figure 7.4: The proposed architecture for Latent Semantic Indexing.

offers a great opportunity to parallelize the sorting operation, which can be conducted simultaneously with other calculations if there is no data dependencies among them.

7.6 Proposed architecture for Latent Semantic Indexing

LSI primarily consists of four computational or logic operations: (1), calculating the squared norms of vectors and the covariances between vector pairs for the SVD and cosine similarity computation; (2), conducting Jacobi rotations on paired squared norms and their respective covariances; (3), updating the matrix elements, the newly generated right orthogonal matrix elements, and the covariances affected by rotation; (4), sorting the singular values for dimensionality reduction and the calculated cosine similarity for the final output.

In our architecture for LSI (see Fig. 7.5), we created four fully pipelined components: the *Vector reduction component*, the *Jacobi rotation component*, the *Update component*, and the *2-3 tree sorting component*, in which the Vector reduction component and the Update component reuse the same computational resources. The Vector reduction component is responsible for computing the squared vector norms and the associate covariances of SVD and cosine similarity computing. The Jacobi rotation component is used to perform plane rotation with squared vector norms to annihilate their related vector covariance. The update component is employed to update the affected elements including the input matrix elements, created right orthogonal matrix elements, and the vector covariances. The final result of this architecture is produced by the $2-3$ tree sorting component, which sorts the document vectors according to their cosine similarity with query vector.

7.6.1 Vector reduction component

The Vector reduction component extends the Hestenes Preprocessor of [Wang and Zambreno (2014b)]'s architecture to compute vector dot products for the norms and covariances computing, the query vector coordinates updates, and cosine similarity calculation. In the process of LSI, both the SVD computing and vector cosine similarity computation require the squared vector norms and covariances among vectors. Thus, the Vector reduction component compute squared column 2-norms and covariance between column vectors through $A_i^T * A_j$ with a design of multiple layers of pipelined multiplier-arrays (shown in Fig. 7.5). In this design, a multiplier-array is responsible to calculate the partial results of different squared norms and their related covariances, and operands are reused by all the multipliers in a pipelined manner. The "reduce" process is performed through summing up the calculated product of a multiplier with the results of its corresponding multiplications across all the layers, who share the same matrix column indexes. For instance, as shown in Fig. 7.5, the matrix elements of $A_{i,j+1}$, $A_{i+1,j+1}$, $A_{i+2,j+1}$, and $A_{i+3,j+1}$ are multiplied with $A_{i,j+2}$, $A_{i+1,j+2}$, $A_{i+2,j+2}$, and $A_{i+3,j+2}$ at first, second, third, and forth layer respectively, and their results are summed up as partial result of the covariance between the $(j+1)^{th}$ and $(j+2)^{th}$ columns. At the mean time, operands as $A_{i,j+1}$ are successively applied to the adjacent multipliers for multiplications

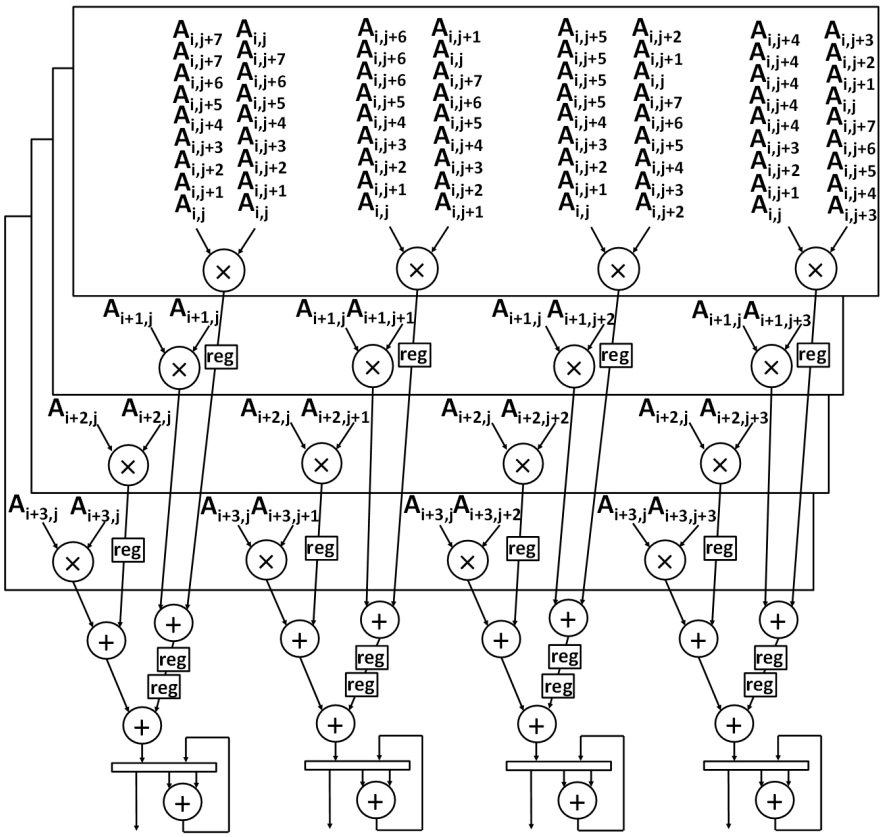


Figure 7.5: The architecture of Vector reduction component.

such as $A_{i,j+1}{}^{*}A_{i,j+3}$, whose product is used to compute the covariance between $(j+1)^{th}$ and $(j+3)^{th}$ columns.

7.6.2 Jacobi rotation component

$$t = \frac{|2 * c_{i,j}|}{|n_j - n_i| + \sqrt{(n_j - n_i)^2 + 4 * c_{i,j}^2}} \tag{7.4}$$

$$cos = \sqrt{\frac{(n_j - n_i)^2 + 2 * c_{i,j}^2 + |n_j - n_i| * \sqrt{(n_j - n_i)^2 + 4 * c_{i,j}^2}}{(n_j - n_i)^2 + 4 * c_{i,j}^2 + |n_j - n_i| * \sqrt{(n_j - n_i)^2 + 4 * c_{i,j}^2}}} \tag{7.5}$$

$$sin = (sign)\sqrt{\frac{2 * c_{i,j}^2}{(n_j - n_i)^2 + 4 * c_{i,j}^2 + |n_j - n_i| * \sqrt{(n_j - n_i)^2 + 4 * c_{i,j}^2}}} \tag{7.6}$$

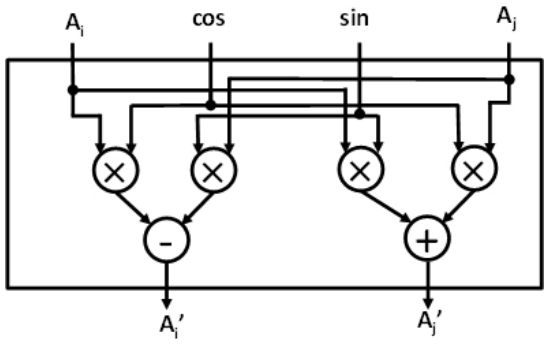


Figure 7.6: The architecture of Update component.

Similar with the Jacobi rotation component of [Wang and Zambreno (2014b)]'s architecture, this Jacobi rotation component is mainly responsible for performing orthogonal transformation between column vectors. As described in Algo. 3, the orthogonalization between column vectors is calculated through a series of operations on paired squared column 2-norms and the associate covariance. By expanding the Jacobi rotation process, its equations can be represented as eq. (7.4), eq. (7.5), eq. (7.6), where n_i and n_j represents the squared 2-norms of column vectors, while the covariance between columns is represented by $cov_{i,j}$. The calculated parameter t is an update factor that is used to update its corresponding vector 2-norms, and annihilate covariance between the rotated vectors. As floating-point arithmetic has become increasingly popular in many scientific and engineering applications for its wider value range support, in our architecture, we use floating-point operators to perform the additions, subtractions, multiplications, divisions, and square-root operations, and all of which can be performed concurrently if independent operands are applied. Although fully parallelize all the independent operations can achieve high throughput, the floating-point computational cores are featured with expensive cost of resources. To balance the efficiency both in speed and resources usage, the floating-point computational cores are reused among those calculations. Besides, the divider is also reused in later computations for query vector coordinates updates and vector cosine similarity computation.

7.6.3 Update component

The Update component performs element-wise update on matrix column entries and covariances which are affected by the processed Jacobi rotations in the SVD computing. Generated rotation angle parameters cos and sin are employed to update the matrix column covariances before they are used by later rotations, and calculate matrix column entries of both left and right orthogonal matrices. The updates of original matrix entries affected by rotations iteratively resulted in the left orthogonal matrix, while the iterative collection update of generated rotation angle parameters create the right orthogonal matrix, both of which consists of simple multiplications, additions and subtractions. The architecture of a update component is demonstrated in Fig. 7.6, in which pipelined multipliers and adder are employed. In our architecture, multiple update components are equipped to perform the all the update operations, whose efficiency dominates the system performance for large-scale matrices, and the number of Update component that can be allocated on a single chip is determined by the availability of on-chip resources. To optimize the hardware resource usage, the multiplier-arrays and their connected adders of computational operators of Vector reduction component are reused as the a series of Update components through runtime reconfiguration.

7.6.4 2-3 tree sorting component

The $2-3$ tree sorting component is responsible for producing the results of the LSI process by ranking the documents according to their cosine similarity with the query vector. The $2-3$ tree is a sorted tree data structure, whose node has either one or two data elements, and each node has either at most two or three children nodes. The example of $2-3$ tree structure is shown in Fig. 7.3, in which a node has at most two elements (v_1 and v_2) and up to three children with element values as l (left children), c (center children), and r (right children) in an order of $l \leq v_1 \leq c \leq v_2 \leq r$. In our architecture, the calculated the series of cosine similarity values are streamed into the $2-3$ tree sorting component, and they are continuously inserted into the appropriate position of the sorted tree.

The $2-3$ tree sorting component organizes BRAMs into unit to store tree node information, and uses a group of BRAM units to maintain a full tree structure. The description of the BRAMs usage is introduced in Table 7.1, in which the BRAM $DataMem$ is used to store the floating-point numerics, and each entry of which has corresponding entries in BRAMs $LeftNode$, $MidNode$, $RightNode$, and $ParentNode$ to store the address pointers to its parent, left child, center child, and right child entries. Meanwhile, the BRAM $SortedLink$ keeps the address pointers of the last and next elements in the sorted order, whose access produces the final sorted result. To help the tree search and insertion operation, BRAM $LevelandNeighbor$ and $StatusRegisters$ are implemented, in which BRAM $LevelandNeighbor$ is used to manage the tree node level number and the address pointer of the other data element in the same tree node, while $StatusRgisters$ keeps current tree structural parameters. Besides, $PathRegisters$ are functioned as flags to indicate on which path the data movement is stalled due to this path is affected by a tree structure update. Additionally, FIFOs are used to synchronize the data movement among BRAMs.

The $2-3$ tree is a self-balanced search tree data structure, and in our architecture, the operations are performed on the $2-3$ tree mainly includes searching, insertion, update and deletion. When a new data element arrives, the process starts with searching the proper position for insertion based on its key value. The search function starts with examining the root node, and then be directed to the proper subtree based on its key value. By recursively performing comparisons from the root node to the leaf node level by level, the search process for insertion is ended when leaf node is reached, and the insertion operation normally takes place at the leaf level. If only one element exist in a leaf node, the new data element can be directly added into this node, otherwise update operation is started due to the maximum capacity of a single node is violated with new element inserted. The update operation splits a tree node into two tree node, and move the elements with middle key up to the parent node. Update operation is performed upwards recursively if parent tree node capacity is exceeded, and new root node is generated if the old root node has more than two data elements. Similar with data insertion, to delete a node, the iterative updates from child node to parent node are performed with data move down until the $2-3$ tree property is satisfied.

Table 7.1: Local memory design for the management of 2-3 tree structure.

BRAM or Register(s)	Entry Field	Description
Data Mem	Data value	store the data values continuously
Left Node	Addr1 Addr2	Addresses of two elements in its left children node
Mid Node	Addr1 Addr2	Addresses of two elements in its mid children node
Right Node	Addr1 Addr2	Addresses of two elements in its right children node
Parent Node	Addr1 Addr2	Addresses of two elements in its parent node
Sorted Link	Next Last	Addresses of two elements which are right before and after in sorted order
Level and Neighbor	Level Neighbor	Its level index Address of its neighbor element within a node
Group ID	Connected Nodes	Memory group indexes of all connected nodes or elements
Status Registers	Status	Addresses of root elements The number of tree levels
Path Registers	Paths	Path Status indicate if any any update in its moving path
FIFOs	value related info	Buffers to move and synchronize data and corresponding info

To parallelize the $2-3$ tree operations, numerous search process can be executed concurrently, and the key value comparisons at different level can be performed in parallel. As search the tree does not alter its structure, the parallelism of the search operation is straightforward. When proper tree node is located, the new element is added or an old element is deleted, and the tree structure is needed to be updated when tree property is violated. In most cases, the search, insertion, deletion and update operations can be simultaneously performed, since, in this component, we maintain the tree structure and update the tree nodes connections locally with out affecting the operations on the rest part of the tree.

An example 2-3 tree operation process is demonstrated in Fig. 7.7, in which independent comparisons are performed concurrently at different level except the stall happens at the second

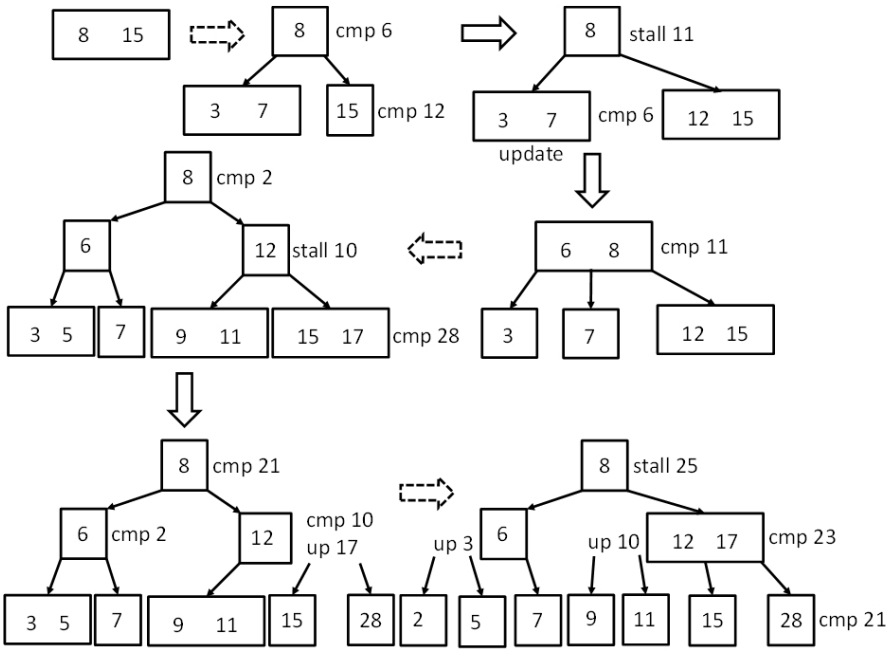


Figure 7.7: The example of parallel processing with new elements insertions and 2-3 tree structure updates.

level tree node when 10 arrives due to its right children leaf node starts an update to recover the tree property violation at this cycle. Each tree node can perform the comparisons with up to 4 elements include one element move up from lower level as the result update operation, and one element move down for locating proper tree node. When an update take place, the data element stall will be placed at every level of tree node in the path to the root, and no operation will be performed at these tree nodes at this cycle. The example result of the $2-3$ tree with all data elements inserted is demonstrated in Fig. 7.8, in which dashed line represents the partition of the $2-3$ tree into numerous groups of the BRAM units, which can be operated in parallel. Input element starts operations with the BRAM unit who has the root tree node, and communications of data elements and their associated parameters take place among BRAM units as needed.

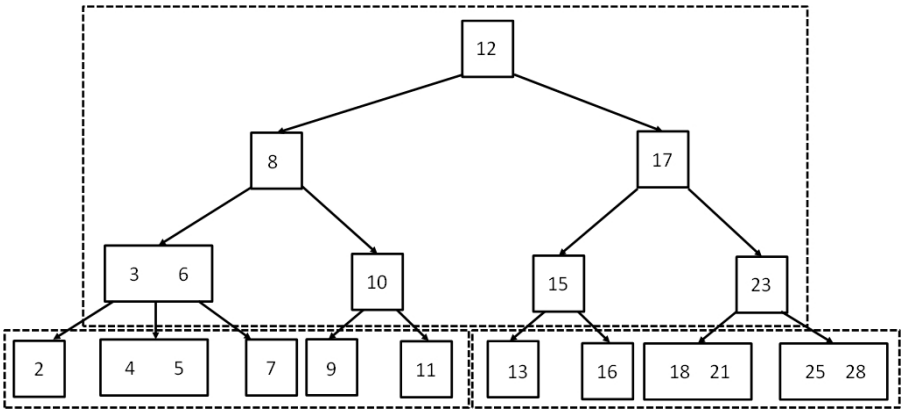


Figure 7.8: The resulted 2-3 tree with all data inserted.

7.7 Implementation and Experimental Evaluation

7.7.1 Implementation and experimental setup

To evaluate the performance of our Latent Sematic Indexing design, we implement our architecture on a single Xilinx Virtex-5 XC5VLX330 FPGA on the Convey HC-2 system [Convey Computer HC-2 (2012)]. In our implementation, we generate the double-precision floating-point computational cores by using Xilinx Coregen generator [Xilinx Inc. (2012)]. In the Vector reduction component, eight layers of multiplier-array are implemented, in which 32 multipliers and 32 adders are used. Dynamic reconfiguration is performed on the Vector reduction component to the Update components with 32 multipliers and 16 adders at the first level. In the Jacobi rotation component, 1 multiplier, 2 adder, 1 divider and 1 square-root operator are employed, which initializes 8 independent Jacobi rotations in pipeline in every 64 clock cycles. In our system, our generated computational cores are configured with default latencies as 9, 14, 57, 57 clock cycles for multiplier, adder, divider and square-root calculator respectively. To synchronize the input and output, two groups of eight 64-bit width FIFOs are used, and another group of FIFOs are employed for the data communication between the Vector reduction component, the Jacobi rotation component, and the Update components. To temporarily cache the rotation angle parameters and covariances, simple dual port RAMs are used. Four groups of eight simple dual port RAMs are employed for the $2-3$ tree sorting component, in which Data Mem is configured with 64 bit entries, while 16 bit bandwidth is used for other

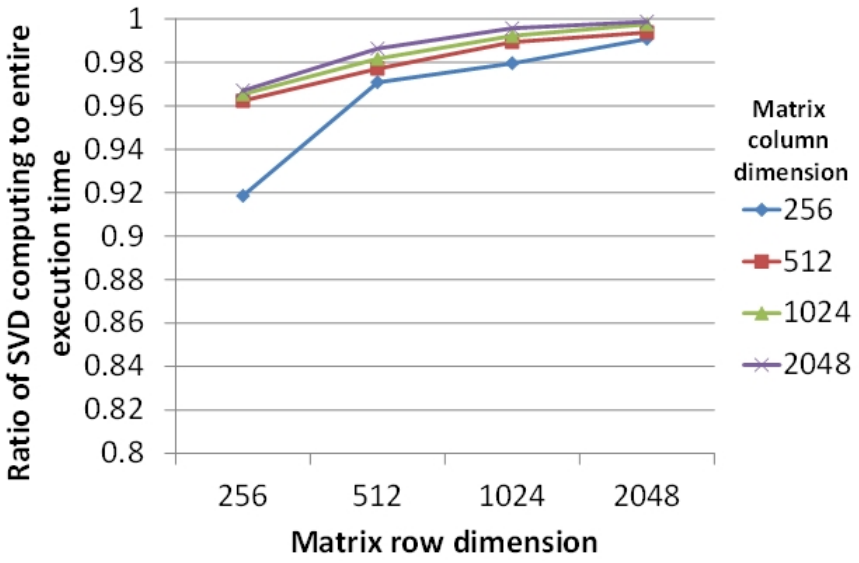


Figure 7.9: The percentage of time consumption for SVD computing in the entire LSI execution (k is 64).

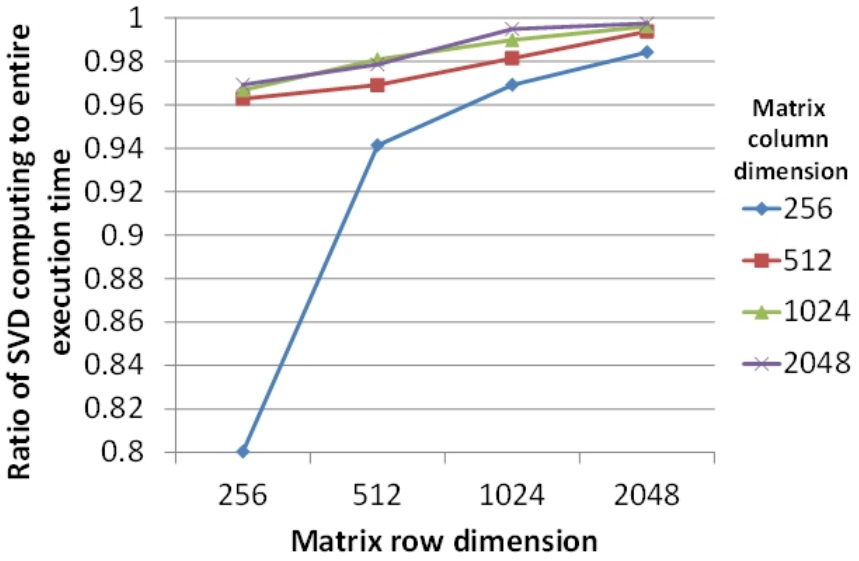


Figure 7.10: The percentage of time consumption for SVD computing in the entire LSI execution (k is 128).

local pointer storages. The system is evaluated by executing at 150 Mhz, in which the SVD computing is an iterative process, and each element is rotated by 6 times, which is believed sufficient for achieving reasonable convergence.

7.7.2 Performance analysis

By profiling the sequential LSI implementation, the SVD computing normally takes more than 90% of LSI execution time (see Fig. 7.9 and Fig. 7.10). In both Fig. 7.9 and Fig. 7.10, the experiment results demonstrate higher percentage of execution time is consumed by SVD computing in LSI operation as matrix dimensions grow. Although, the value of reduced subspace dimension k has comparably trivial impact on the profiling results, the percentage of SVD computing in LSI operation slightly increases as the subspace dimension k grow.

Our experiment are applied with both square and rectangular matrices with different dimensions of reduced k-subspace, the performance for matrices with dimensions from 256 to 1024 is demonstrated in Fig. 7.11, in which the dimension of the subspace is set at 128. The experimental results demonstrate that the execution time grows significantly as the number of documents increases, due to the amount of updates for the right orthogonal matrices, and matrix vector covariances is determined by the quantity of documents. Comparably, the number of key words, who determines the dimensions of the left orthogonal matrix, has smaller impact on the overall performance. However, the LSI process requires the usage of both left and right orthogonal matrices, and the matrices elements updates after each rotation usually dominates the performance of the SVD computing for medium to large sized matrices. In our design, we can temporarily store matrices with dimensions under 128, and when the matrix size grows over 128, the performance is increasingly affected by the I/O bandwidth for the matrix elements and covariances communicating between our LSI architecture and the off-chip memory.

In Fig. 7.12, the performance of LSI computation on different dimensional matrices with different subspace dimensions is demonstrated. In the LSI process, the dimensionality reduction is performed after the SVD computing based on the calculated singular values. In this experiment, we have the observation that the number of k has comparably trivial impact on the overall performance of the LSI process, due to the overall the performance of LSI is heavily

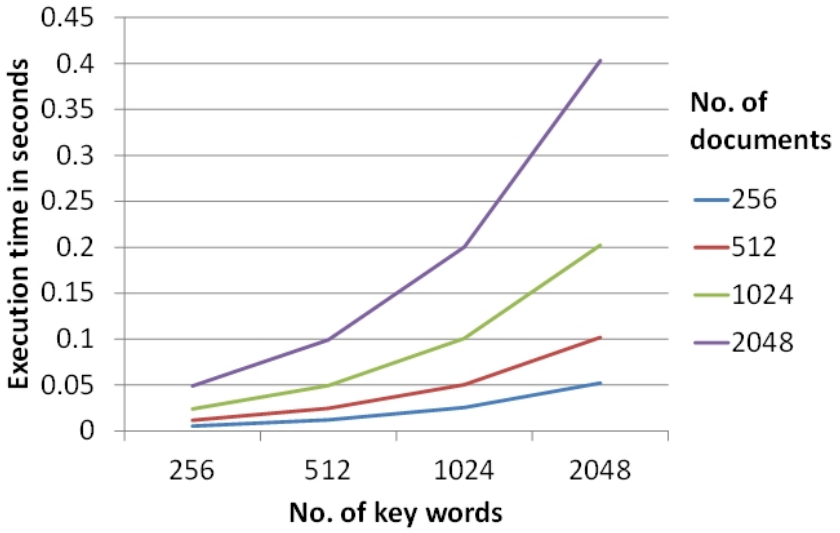


Figure 7.11: LSI computation time (in seconds) for matrices with different dimensions (k is 128).

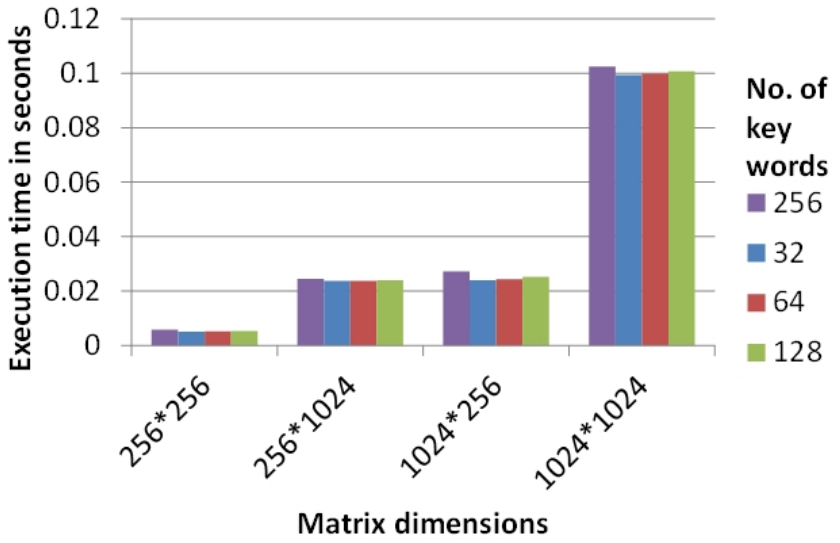


Figure 7.12: LSI computation time (in seconds) for different dimensional matrices with different k-subspace.

determined by the SVD computation. For rectangular matrix, as the dimension of subspace grows, the number of key words is a more significant factor that affect the performance of the LSI process, compared to the number of documents.

When the input dataset has a column dimension over 256, the I/O bandwidth starts affecting the speed of LSI execution. Due to the on-chip storage resources are limited, and the amount of communications between the LSI engine and off-chip memory increases as the dimension of input dataset grows. Fig. 7.13 demonstrates the average number of double-precision floating-point I/O requests per clock cycle for input matrix with various column dimensions. The I/O bandwidth becomes a significant factor to affect the efficiency of our design as the number of double precision floating point operands communication requests per clock cycle grows over 16, which number is the optimized I/O bandwidth that is provided by our experimental platform.

Comparisons of execution times have been made between our implementation and Matlab LSI program, and in Fig. 7.14, the dimensional speedups of our design compared to the Matlab 7.10.0 LSI program running on a 2.2 GHz dual core Intel Xeon processor are demonstrated, in which Matlab program uses the SVD and sorting routines. By analyzing those data points in Fig. 7.14, our architecture shows better efficiency than Matlab implementations, with a dimensional speedups that can be achieved range from 3.8× to 10.1× for matrices with dimensions from 256 to 2048. As the our experimental platform's on-chip memory is limited, and I/O throughput has increasingly significant impact on the performance as matrix dimension grows. The speedup decrease as the I/O requests start to affect the overall performance, and the speedups then gradually increase after the dimensions have further growth due to the efficiency of pipelined computational cores usage is improved.

We have also compared the performance of SVD computing in our design with the published results of SVD solutions implemented with Intel MLK 10.0.4 or on NVIDIA 8800 GPU with 128 stream processors [Lahabar and Narayanan (2009)]. Although MKL-based and GPU-based solution have demonstrated better efficiency when matrix size grows over thousands, our design is more efficient for processing matrices with dimension up to $2k$. In our design, we are able to achieve 8.9 GFLOPs with 78% slice LUT, 89% BRAM, and 67% DSP used.

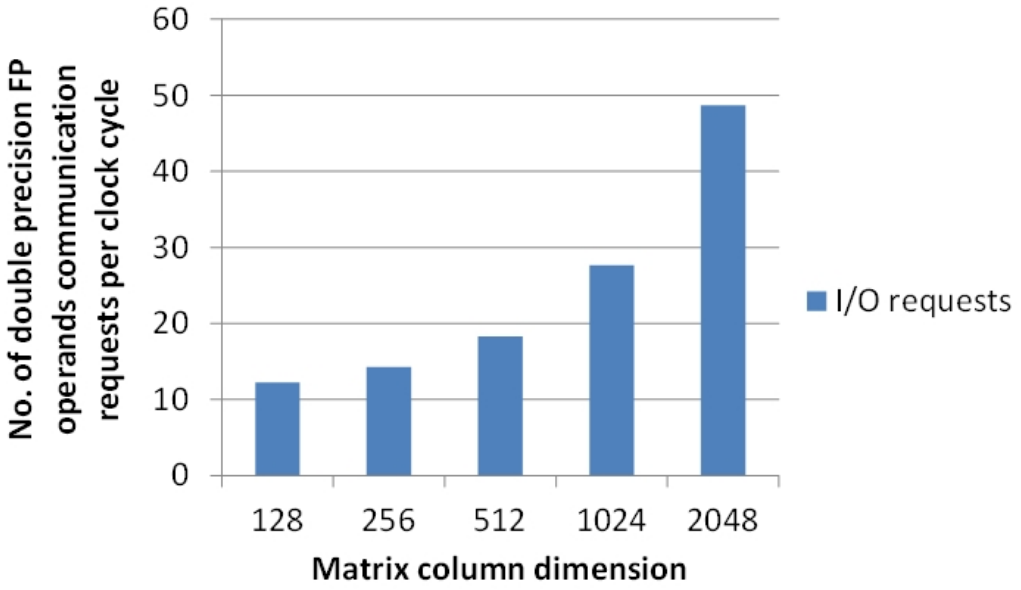


Figure 7.13: The average No. of double precision floating point operands communication requests per clock cycle for input matrix with various column dimensions (k is 128).

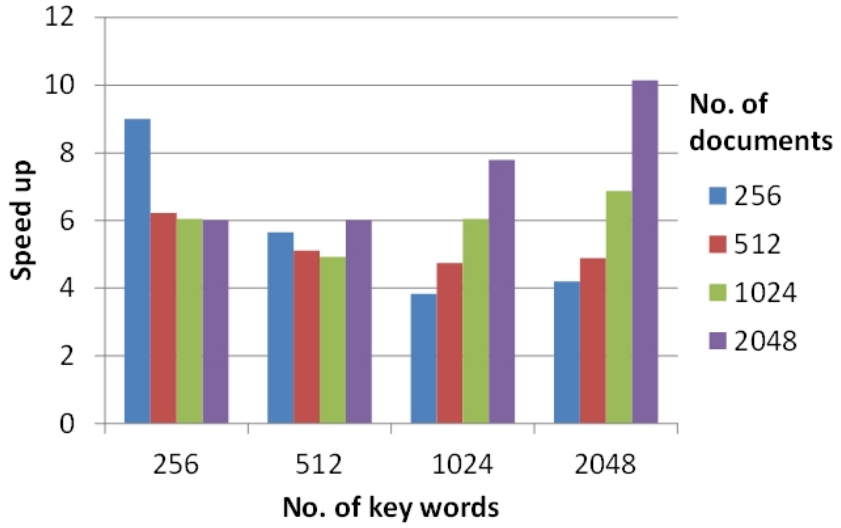


Figure 7.14: Speedups of our LSI process compare to Matlab LSI program execution.

7.8 Conclusion

An FPGA-based hardware architecture is proposed and implemented to perform Latent Semantic Indexing, which parallelizes the Hestenes-Jacobi SVD computation, the vector computing, the calculation of cosine similarity, and the sorting of the resulted vector. The performance analysis indicates our design has better performance than standard software solutions, and better efficiency for matrices with small to medium dimensions compared to GPU and MKL implementations. Further optimization are planned to merge the dimensionality reduction process within the SVD iterative process as the future work.

CHAPTER 8. A CONFIGURABLE ARCHITECTURE TO ACCELERATE HOMOTOPY ℓ_1-MINIMIZATION

Modified from a Journal paper under submission

Xinying Wang[1] and Joseph Zambreno[2]

8.1 Abstract

ℓ_1-norm minimization has played a vital role in many scientific and engineering applications to solve the underdetermined system of equations, especially in compressed sensing, which area acquires and accurately reconstructs signals with significantly reduced dimensions. However, ℓ_1-norm minimization is considered computationally complicated. Although a few approximate algorithms were proposed to improve the performance of ℓ_1-norm minimization in speed, their accuracy are limited. To balance the speed and accuracy, hardware accelerators (GPUs, FPGAs) are used to explore parallel solutions to accelerate exact methods of ℓ_1-norm minimization. However, the existing implementations are either parallelizing slow algorithms or limiting the parallelism in simple matrix-vector computations. Homotopy algorithm is the fastest exact method for ℓ_1-norm minimization, which constructs a decreasing sequence of regularization parameter, and "break points" are identified along the path with the variable associated support set updated by adding or removing components. The classical Homotopy algorithm performs rank-1 update every time when a single element entering or leaving the support set. However, it requires executing a large number of iterations to reach the convergence. In this paper, we modify the Homotopy algorithm to allow only a single update of the search direction of Homotopy path to be performed after a few qualified elements added to the support set by

[1]Primary researcher and author
[2]Correspondence author

performing Cholesky Decomposition, which reduces the number of iterations for convergence. To accelerate Homotopy ℓ_1-norm minimization, we propose an FPGA-based highly pipelined architecture, which can be dynamically configured to perform either Cholesky Decomposition or rank-1 update. By evaluating our architecture with randomly generated dataset and benchmark from the application of robust face recognition, our experimental result demonstrate dimensional and application-dependent speedups from 3.6× to 9.9× compared to optimized software solution with reasonable accuracy achieved.

8.2 Introduction

The pursuit of the minimum ℓ_1-norm solution has been considered as an effective way to solve the underdetermined system of linear equations, which has fewer linear equations than the number of unknown variables [Chen et al. (2001)]. To recover the unknown vector $x_0 \in \mathbb{R}^n$ from a linear transformation $y = Ax_0$ with the known resulted vector $y \in \mathbb{R}^m$ and matrix $A \in \mathbb{R}^{m \times n} (m < n)$, ℓ_1-norm minimization are normally transformed into a convex optimization problem as basis pursuit denoising (BPDN) [Chen et al. (2001)]. Solving the ℓ_1-norm minimization problem or BPDN is considered as a time consuming process, and classical approaches such as primal-dual interior-point methods are challenged by high time complexity [Kojima et al. (1990)]. In the past a few years, several methods have been proposed to improve the performance of ℓ_1-norm minimization. However, the state-of-the-art are still suffered from performance issues in terms of speed and accuracy for exact and approximate approaches respectively [Yang et al. (2010)].

Solutions for ℓ_1-norm minimization have been categorized into two groups that exact methods and approximate methods. Among the cutting-edge ℓ_1-norm minimization solutions, Gradient Projection [Figueiredo et al. (2007)] and Homotopy [Malioutov et al. (2005)] are the most representative exact methods, while Soft Shrinkage-thresholding [Wright et al. (2009)], Proximal Gradient [Becker et al. (2011)], and Alternating Direction [Yang and Zhang (2011)] are the widely used approximate methods. Although approximate ℓ_1-min solutions demonstrate an improvement of the efficiency compared to exact methods, their achievement of accuracy is limited. An investigation on the performance of latest ℓ_1-norm minimization solvers for

the application of robust face recognition demonstrates that exact methods generally outperform their approximate alternatives in terms of recognition accuracy especially when the pixel corruption rate grows [Yang et al. (2010)]. Among the exact methods, Homotopy algorithm has demonstrated the capability to achieve the best performance both in speed and accuracy. However, it is running one order of magnitude slower than approximate solutions [Yang et al. (2010)].

The Homotopy algorithm solves the objective function of BPDN by decreasing a regularization parameter to follow a Homotopy path from the ℓ_2 constraint to the ℓ_1 objective in a piece-wise manner [Donoho and Tsaig (2008); Efron et al. (2004)]. The Homotopy algorithm constructs a decreasing sequence of regularization parameter, and identifies "break points" along the path with the variable associated support set updated by adding or removing components [Yang et al. (2010)]. In this process, the performance is dominated by determining the direction of the path at every "break point" through the subdifferentiation of the convex function, which is normally calculated by conducting matrix decomposition, matrix inverse or rank-1 update [Yang et al. (2010)](such as QR [Wang et al. (2014)] or symmetric LU Decomposition [Wang et al. (2014)]). Fortunately, the matrix decomposition, matrix inverse or rank-1 update can be easily parallelized, which provides the opportunities to accelerate the Homotopy ℓ_1-norm minimization.

Many research has been conducted to explore novel algorithms for optimizing both the speed and accuracy of ℓ_1-norm minimization [Figueiredo et al. (2007); Malioutov et al. (2005); Wright et al. (2009); Becker et al. (2011); Yang and Zhang (2011)]. However, the improvement of the performance is still limited due to the inherent high computational complexity especially for exact methods [Yang et al. (2010)]. Parallel implementations were proposed for ℓ_1-norm minimization by using hardware accelerators such as GPU [Shia et al. (2011); Fang et al. (2011)] and FPGA [Blache et al. (2012); Bai et al. (2012); Rabah et al. (2015); Ang and Kumar (2013)]. However, most of them were designed to accelerate the BPDN with Orthogonal Matching Pursuit (OMP) [Pati et al. (1993)], a traditional ℓ_1-min solver who is considered with poor recovery accuracy, questionable scalability, and conditional failures [Ang and Kumar (2013); Plumbley (2006)]. Although several GPU [Shia et al. (2011)] or FPGA-based [Ang and Kumar

(2013)] designs were implemented to optimize advanced solvers, their parallelism are limited to simple matrix-vector computations, which leaves additional space for further performance improvement.

Homotopy algorithm is an iterative process to achieve the convergence through tracing the Homotopy path. In this paper, we modify the Homotopy algorithm to allow multiple elements to be added to the support set together followed by a single direction update, which can significantly reduce the number of iterations required for convergence. We propose an FPGA implementation for ℓ_1-norm minimization with Homotopy algorithm, which can be dynamically configured to perform either parallel Cholesky Decomposition or rank-1 update. By analyzing the performance of our proposed hardware solution, our architecture shows dimensional and application-dependent speedups from 3.6× to 9.9× compared to optimized software solution with reasonable accuracy achieved.

8.3 Theoretical background

8.3.1 ℓ_1-norm minimization problem

ℓ_1-norm minimization has been considered as an effective factorization tool to solve the underdetermined system of equations, especially in the applications of compressive sensing. Given a measurement vector $b \in \mathbb{R}^n$ and a known matrix $A \in \mathbb{R}^{p \times n}$, an equation of $b = Ax$ can be built with an unknown vector of interest $x_0 \in \mathbb{R}^n$ $(p < n)$. Thus, the equation can be solved through minimizing ℓ_1-norm with formula (8.1), a computational tractable linear program.

$$\underset{x}{\text{minimize}} \ \|x\|_1 \text{ subject to } b = Ax \tag{8.1}$$

In this underdetermined system, n and p represent the number of variables and equations of this linear system, and they can be used to determine the sparsity of the solution x_0. In the sense of Compressive Sensing, x_0 can be exactly recovered or reasonably approximated when the unknown vector x_0 is sufficient sparse and matrix A has incoherent columns [Bruckstein et al. (2009)].

To solve ℓ_1-norm minimization, the white noise are normally taken into account to relax the equation $b = Ax$. By adding the error measurement n, the equality turns into $b = Ax + n$ ($\|n\|_2 \leq \epsilon$), which approximation is named as basis pursuit denoising (BPDN) (see (8.2)) [Chen et al. (2001)].

$$\underset{x}{\text{minimize}} \ \|x\|_1 \text{ subject to } \|b - Ax\|_2 \leq \epsilon \tag{8.2}$$

By solving this regularization problem, optimal solution for ℓ_1-norm minimization can be pursued through the process of balancing the trade-off between minimizing the ℓ_2 distance of the residual $(y - Ax)$ and simplifying the ℓ_1-norm of unknown vector x. In the past decades, ℓ_1-norm minimization problem had attracted broad attention in the community of signal processing, and many general solvers were proposed such as interior points method [Yang et al. (2010)], Gradient Projection (GP) [Figueiredo et al. (2007)], Homotopy [Malioutov et al. (2005)], Iterative Shrinkage-Thresholding (IST) [Wright et al. (2009)], Proximal Gradient (PG) [Becker et al. (2011)], and Augmented Lagrange Multiplier (ALM) [Yang and Zhang (2011)]. Among them, Homotopy ℓ_1-norm minimization method is recognized as one of the most efficient approach to produce the exact solution of the ℓ_1-norm minimization problem.

8.3.2 Homotopy method for ℓ_1-norm minimization

To perform ℓ_1-norm minimization, the optimization problem (8.2) is normally transformed into a more convenient and unconstrained form as shown in (8.3) [Donoho and Tsaig (2008)].

$$\underset{x}{\text{minimize}} \ \lambda\|x\|_1 + \frac{1}{2}\|b - Ax\|_2 \tag{8.3}$$

The Homotopy ℓ_1-norm minimization method normally initializes the solution x as 0, and λ is set as infinity. Then, λ decreases gradually in a piecewise linear manner, and changes are computed accordingly. In this process, the solution x are built successively by adding or removing components to or from its active set. The algorithm is terminated when the constraint is reached that $\|b - Ax\|_2 = \epsilon$. In many cases, the use of a pre-calculated initial solution can

shorten the path to achieve the optimal solution with Homotopy ℓ_1-min algorithm, and the sparsity rate of the solution determines the number of steps in this path to reach the solution.

While the entire path of Homotopy ℓ_1-min algorithm maintains the optimality of (8.3), the relationship (8.4) can be derived from the Karush-Kuhn-Tuchker (KKT) optimality conditions [Kuhn and Tucker (1951)] by subdifferentiating the objective function of (8.3).

$$\frac{\partial f_\lambda(x)}{\partial x} = -A^T(b - Ax) + \lambda u = 0 \tag{8.4}$$

where u is the subdifferential of $\|x\|_1$, and its values are defined as the set of (8.5) [Dai et al. (2015)].

$$\begin{cases} u_i = sgn(x_i), x_i \neq 0 \\ u_i \in [-1, 1], x_i = 0 \end{cases} \tag{8.5}$$

Therefore, the eq. (8.4) can be rewritten as eq. (8.6).

$$\begin{cases} |A_\Sigma^T(b - Ax)| \leq \lambda \\ A_\Phi^T(b - Ax) = \lambda \cdot sgn(x_\Phi) \end{cases} \tag{8.6}$$

where Σ and Φ represent the sparse support sets, whose elements are the column indexes of matrix A. By satisfying eq. (8.6), the optimality condition of eq. (8.3) can be guaranteed. By further taking the derivative on both sides of eq. (8.6) with λ, the Homotopy path search direction can be obtained as that $x_i' = 0$ when $i \in \Sigma$, and $x_i' = -(A_\Phi^T A_\Phi)^{-1} sgn(x_\Phi)$ when $i \in \Phi$, with the assumption that limited change is applied to λ, and the matrix $A_\Phi^T A_\Phi$ is of full rank [Dai et al. (2015)]. The Homotopy algorithm is designed to fit its entire path solution to eq. (8.3) for every small change of the regularization parameter λ in a piecewise manner. Thus, the regularization parameter λ and variable x can be updated as $\lambda^{d+1} = \lambda^d + \Delta\lambda$ and $x^{d+1} = x^d + \Delta\lambda x'$, where d indicates the sparse support after d times of updates along the homotopy path.

The updates of regularization parameter λ and variable x may lead to either adding or dropping event occur that either add element into or drop event from active support set. When the scenario happens that $A_i^T(b - A(x^{(d)} + \Delta x')) = \pm(\lambda^d + \Delta\lambda)$, where $i \in \Sigma$, an index

$i \in \Sigma^{(d)}$ will be added to the active support set $\Phi^{(d)}$, and the index i is determined by the eq. (8.7) [Dai et al. (2015)].

$$\gamma^{+} = \min_{\forall i \notin \Phi} \Big\{ \frac{A_i^T(b - Ax^{(d)}) - \lambda^{(d)}}{A_i^T Ax' + 1}, \frac{A_i^T(b - Ax^{(d)}) + \lambda^{(d)}}{A_i^T Ax' - 1} \Big\} \tag{8.7}$$

Meanwhile, another scenario may happen that an index $i \in \Phi$ with its value crosses zero, which violates the sign agreement. When this scenario occurs, the index i will be removed from active set Φ, and the index i can be obtained through eq. (8.8).

$$\gamma^{-} = \min_{\forall i \in \Phi} \big\{ -x_i^{(d)} / x_i' \big\} \tag{8.8}$$

In constructing the Homotopy path with decreasing regularization parameter λ, "break points" are identified at the occurrence of either adding element into active set or dropping element from active set, and the operation at each "break points" is determined by $\min \{\gamma^{+}, \gamma^{-}\}$. The Homotopy path maintains the optimality conditions of eq. (8.3) at each "break points", and variable x converges to the solution of eq. (8.3) as the regularization parameter λ approaches zero [Yang et al. (2010)].

8.4 Related work

ℓ_1-norm minimization has been widely used in compressed sensing as an efficient approach to recover the sparse estimate of compressed dataset [Chen et al. (2001)]. However, it is identified as a computationally expensive operation. In the past a few years, new algorithms were proposed to improve the performance of ℓ_1-norm minimization both in speed and accuracy. Primal-Dual Interior-Point Algorithm (PDIPA) is considered as a classical solution for solving ℓ_1-norm minimization as a linear programming problem. However, a total of $\mathcal{O}(\sqrt{n})$ iterations and $\mathcal{O}(n^3)$ operations in each iteration are required to solve this linear system [Yang et al. (2010)].

Figueiredo and etc. introduced Gradient Projetion (GP) algorithm, which reformulated the linear programming implementation in PDIPA as a bounded quadratic programming problem [Figueiredo et al. (2007)]. By selecting appropriate search parameter, significant improvement of

performance is achieved as better convergence speed is demonstrated [Figueiredo et al. (2007)]. But, when the problems are of large scales, approximate solution should be used as it is computationally impractical for solving GP directly. To further improve the performance, Malioutov and etc. discovered the fact that the objective function of the Quadratical Programming follows a homotopy path from the ℓ_2 constraint to the ℓ_1 objective, while their regularization parameter decreases. Based on this fact, Homotopy method was proposed to solve ℓ_1 norm minimization with a more efficient and greedy optimization path selected compared to previous two, and its performance is dominated by updating direction in $\mathcal{O}(n^2)$ operations [Malioutov et al. (2005)]. However, in case of its sparsity grows proportionally with the dimension, the upper-bound of Homotopy method becomes $\mathcal{O}(n^3)$ [Yang et al. (2010)].

In the meantime, numeric approximate methods were proposed such as Soft Shrinkage-thresholding [Wright et al. (2009)], Proximal Gradient [Becker et al. (2011)], and Alternating Direction [Yang and Zhang (2011)]. Instead of directly solving unconstraint BPDN problem as Homotopy method, in Soft Shrinkage-thresholding approach [Wright et al. (2009)], Wright and etc. replaced hessian matrix with an approximated diagonal matrix, and used light-weight vector operations or vector-matrix multiplications to keep track of the updates for advanced matrix decomposition (QRD, LUD). As a representative of another class of approximate algorithm, Proximal Gradient was presented by Becker and etc., which iteratively form quadratic approximations to the original cost function at carefully selected points, and minimize those approximations for closer solution sets to the exact ones [Becker et al. (2011)]. In the recent years, Yang and etc. proposed a novel procedure, named as Alternating Direct method, to solve the (8.3) through alternating between optimizing the ℓ_1-norm of the sparse signal and ℓ_2-penalty of the residual term [Yang and Zhang (2011)]. Although the investigations of approximate methods demonstrates oblivious performance advantages in speed, their accuracy is limited especially when the dataset is highly corrupted [Yang et al. (2010)]. Thus, it is necessary to explore high performance implementations for exact solutions.

To improve the computational speed, hardware accelerators (e.g. GPU, FPGA) are used to parallelize the ℓ_1-norm minimization. Among all the advanced algorithms, Orthogonal Matching Pursuit (OMP) [Pati et al. (1993)] is the most frequently selected algorithm for parallel

implementation due to its low complexity and acceptable accuracy for many applications. In [Fang et al. (2011)], Fang and etc. parallelize the Fujimoto's matrix-vector multiplication algorithm and the matrix-inverse-update algorithm on GPU. In [Blache et al. (2012)], Blache and etc. identified three computational kernels of OMP process, and proposed high level parallel strategy for matrix-vector operations and modified Cholesky Decomposition-based matrix inverse with FPGA. However, FPGAs are able to provide further accelerations with finer grained designs. In [Bai et al. (2012)], Bai and etc. implemented vector multiplication unit to parallelize the vector-matrix multiplication in the incremental QRD of OMP process, and used look-up table to help accelerate the processing of square root reciprocal on a Virtex FPGA. However, memory access bandwidth becomes a major limits for the efficiency of parallel implementation. To optimize the memory usage, Rabah and etc. proposed a pipelined architecture for OMP algorithm, which includes Newton-Raphson iteration-based inter-product computation and inversion, and scalable Moore-Penrose pseudoinversion [Rabah et al. (2015)]. Although quite a few researches have been conducted to explore high performance OMP implementation for ℓ_1-norm minimization, the limited accuracy and conditional failures of OMP demands the researchers to accelerate more advanced algorithms. Shia and etc. presented a fast ℓ_1-norm minimization implementation on GPU for robust face recognition, which parallelized Augmented Lagrangian Method (ALM), whose kernel involves vector algebra and matrix-vector multiplication [Shia et al. (2011)]. Ang and etc. proposed an FPGA-based embedded-friendly BPDN solver, whose computational intensive kernels are implemented by pipelining and parallell processing [Ang and Kumar (2013)]. However, their implementations limited the parallelism to simple matrix-vector operations, and it fails to achieve the FPGAs provided potentials of accelerating more advanced matrix computation.

8.5 Modified Homotopy algorithm for ℓ_1-norm minimization

Homotopy algorithm for ℓ_1-norm minimization traces a solution path to solve the quadratic optimization of eq. (8.3) in a piecewise manner. To improve the accuracy and unity, the Homotopy algorithm for ℓ_1-norm minimization is reformulated as eq. (8.9) [Asif and Romberg (2013)], where W is a diagonal matrix with positive weights as the diagonal elements, and u

is defined as $-W\hat{z} - A^T(A\hat{x} - b)$. In this equation, $\hat{x}$ is an initial solution of this optimization problem, and $\hat{z}$ is the sign sequence vector of $\hat{x}$. In this way, the optimization solution path is determined by moving the Homotopy parameter ϵ from 0 to 1. The convergence can be reached faster if an initial value of $\hat{x}$ is provided, otherwise $\hat{x}$ is given as 0. The first active support set is determined by calculating the index of $\arg\max_{j=1}^{n} \|A_j^T b\|$ if no initial active support set is provided, and in practice u is computed through matrix decomposition such as QR Decomposition or LU Decomposition.

$$\underset{x}{\text{minimize}} \ \|Wx\|_1 + \frac{1}{2}\|b - Ax\|_2^2 + (1-\epsilon)u^T x \tag{8.9}$$

The Homotopy ℓ_1-norm minimization iteratively identifies "break points", calculates moving direction and step size, tracks the change of the homotopy parameter and solution, and updates the active support set until the convergence is reached when ϵ no less than 1. The computation of the Homotopy path direction update and step size is identified as the performance dominant component in each iteration, and it is typically calculated through rank-1 update on matrix factor when an index is added or removed to or from the active support set. To optimize the number of iterations for approaching the convergence, the active support set can be updated with numerous indexes in a single iteration. However, a full matrix decomposition has to be performed. Both the rank-1 update of matrix factor or the full matrix decomposition leave the space for the acceleration with parallel and pipelined implementations.

Each iteration of Homotopy ℓ_1-norm minimization is started with computing the subdifferential of x by either matrix factor rank-1 update or new matrix decomposition. Then, the optimality parameters p and q are calculated mainly with matrix-vector operations, and they can be performed partially in parallel with the subdiffrential of x depends on their data dependencies. Parallel comparisons are performed afterwards to determine the step size to next "break point". After determining the step size, stopping criteria is evaluated, and convergence is considered reached as ϵ passes over 1. Unless the stop criteria is met, the x and ϵ are updated with the resulted direction value and step size, and the active support set is updated accordingly.

Input: A full rank matrix $A \in \mathbb{R}^{m \times n}$ $(m < n)$, and a vector $b \in \mathbb{R}^m$, an optional diagonal matrix $W \in \mathbb{R}^{m \times m}$ with positive weights as elements

Output: x^*

Initialization: $x \leftarrow 0$ (or $\hat{x}$), $\epsilon \leftarrow 0$, the first support index: $i \leftarrow \arg\max_{j=1}^{n} \|A_j^T b\|$, $\Gamma = \{i\}$, u is the decomposition factors of $A_\Gamma^T A_\Gamma$.

repeat

- Compute $\partial x = (A_\Gamma^T A_\Gamma)^{-1} u_\Gamma$ on the support of Γ, otherwise 0 `/* Update the direction */`
- Compute optimality condition parameters p and q $p \leftarrow A_i^T(Ax^* - b) + (1 - \epsilon)u$
- $q \leftarrow \delta(A_i^T A \partial x - u)$
- Compute δ^+ and δ^- `/* Compute the step size */`
- $\delta^+ \leftarrow \min_{i \in \Gamma^c}(\frac{w-p}{q}, \frac{-w-p}{q})_+$
- $\delta^- \leftarrow \min_{i \in \Gamma}(\frac{-x^*}{\partial x})_+$
- $\delta^* \leftarrow \min_(\delta^+, \delta^-)$
- `/* Evaluate the stopping criterion */`
- **if** $\epsilon + \delta^* > 1$ **then**
  - $\delta^* \leftarrow 1 - \epsilon$
  - $x^* \leftarrow x^* + \delta^* \partial x$ `/* Solution for the output */`
  - **break**
- **end**
- $x^* \leftarrow x^* + \delta^* \partial x$ `/* Solution update */`
- $\epsilon \leftarrow \epsilon + \delta^*$ `/* Homotopy parameter update */`
- **if** $\delta^* = \delta^+$ **then**
  - $\Gamma \leftarrow \Gamma \cup \{i\}_+$
  - `/* Add (a) new element(s) to the support */`
- **end**
- **else**
  - $\Gamma \leftarrow \Gamma \setminus \{i\}_-$
  - `/* Remove (a) element(s) from the support */`
- **end**

until $\epsilon = 1$

Algorithm 5: Modified Homotopy algorithm for ℓ_1-norm minimization

The update of active support set is either adding new element(s) to the support set if the optimality condition is violated or dropping the element(s) from the support set once the element(s) in the active support set cross zero. If only one element is updated with the active support set, the computing of direction can be performed through rank-1 update on matrix factor, whose time complexity is $O(n^2)$ if the matrix is sufficient dense. However, large number of iterations are required to reach the convergence along the path. Full matrix decomposition, whose time complexity is $O(n^3)$, will be needed for the occurrence of numerous elements enter the active support set in a single iteration.

In the sequential Homotopy algorithm, only one element is either added into or removed from the active support set, and rank-1 updates are performed on the matrix factor, due to the better efficiency of a rank-1 update than a full matrix decomposition. However, parallel implementation can highly improve the performance of the matrix decomposition. The update with one or multiple elements on active support set in one iteration may lead to better efficiency. Therefore, the algorithm can be modified to parallelize both the rank-1 update and matrix decomposition, and either of them is used in the update process depending on the requirement of efficiency and accuracy.

8.6 Configurable architecture for ℓ_1-norm minimization

The configurable architecture for ℓ_1-norm minimization primarily consists of two processing components: *Matrix-vector computation component* and *Matrix factorization component*, in which the Matrix-vector computation component performs matrix-vector operations (e.g. multiplication, addition, and subtraction) at different phases of the minimization process, while the Matrix factorization component is used to calculate Cholesky Decomposition, rank-1 Cholesky update, and the matrix inverse. Fig. 8.1 provides a high-level view of how these components are related within our architecture, in which additional control and storage elements are used.

8.6.1 The Matrix-vector computation component

The Matrix-vector computation component (shown in Fig. 8.2) consists of numerous layers of pipelined multiplier arrays, in which operand can be reused by multipliers in the same

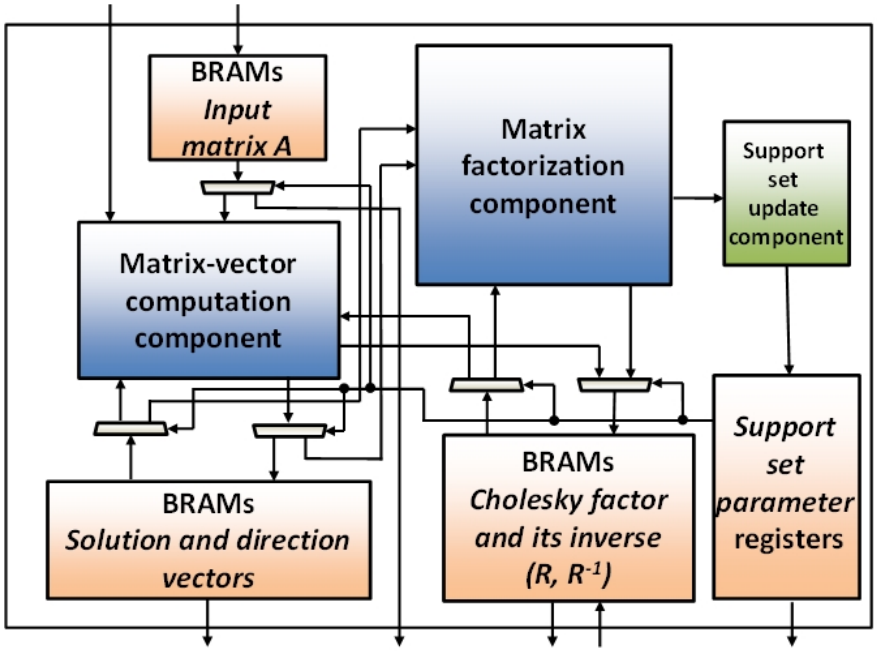


Figure 8.1: Block diagram of the proposed architecture for ℓ_1-norm minimization.

layer. The resulting product of a multiplier is summed up with the results of its corresponding multiplications across layers, after which all the resulted sums are further added up. The Matrix-vector computation component is reconfigurable to perform different matrix-vector operations through collecting the outputs for this component at different levels of computational cores (see the dashed lines and arrows of Fig. 8.2). The first level output produces a series of simple vector element-wise multiplication-addition results, while multiple products of independent vector-vector multiplications can be obtained at the second level output simultaneously. If all the multipliers are applied with the elements from same vector, the vector-vector multiplication result is collected at the third level output, which produces the smaller number of results with a shorter latency compared to the configuration with second level outputs used.

8.6.1.1 Initial setup process

ℓ_1-norm minimization initialized by matrix vector multiplication of $-A^T * b$ with initial solution sets collected (refer B.1). In this case, matrix elements from the same vector are applied to multipliers in the same layer simultaneously, and to multipliers across layers in pipelining. The architecture is configured to produce the results at the third output level,

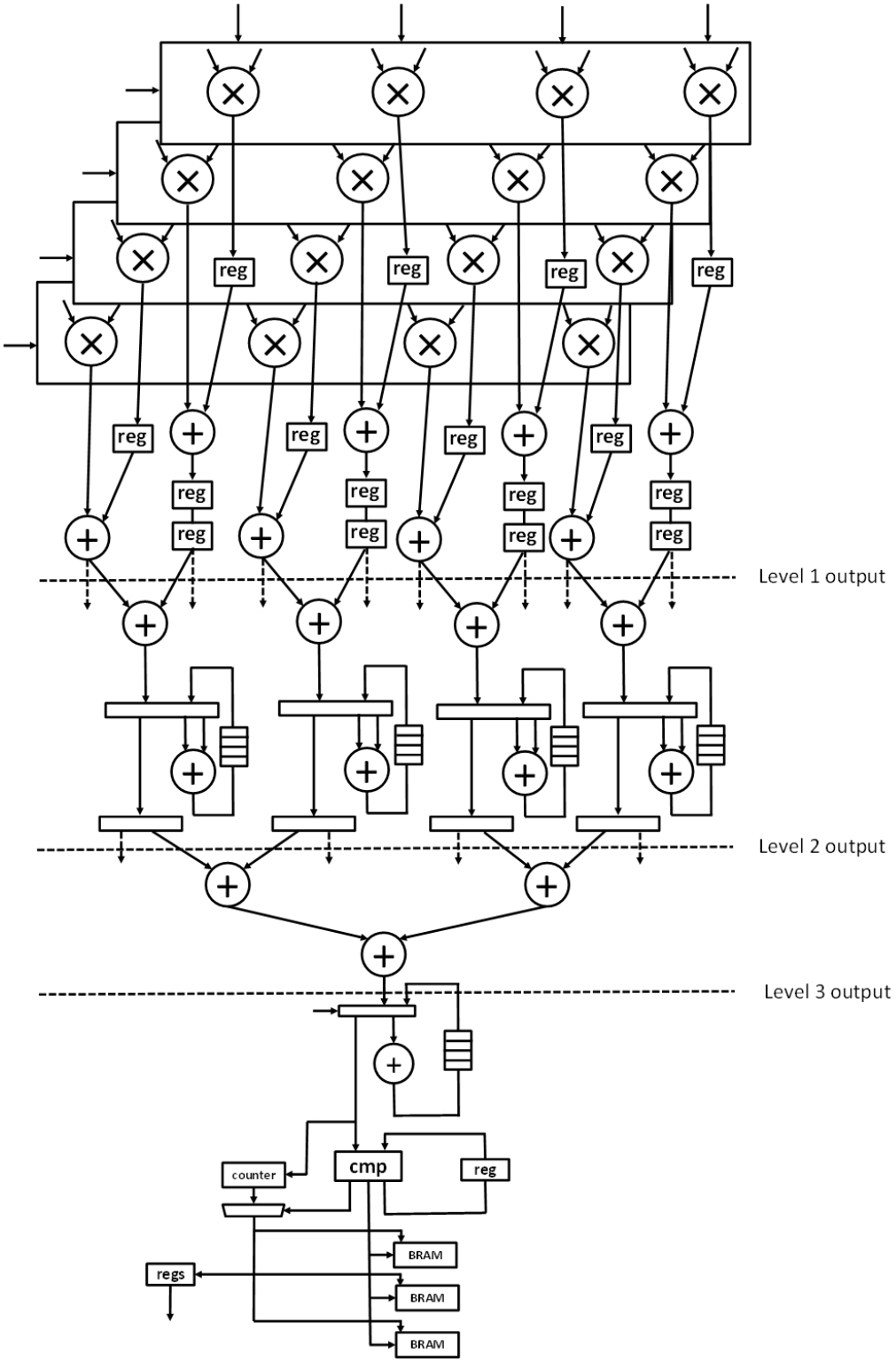


Figure 8.2: The architecture of the Matrix-vector computation component.

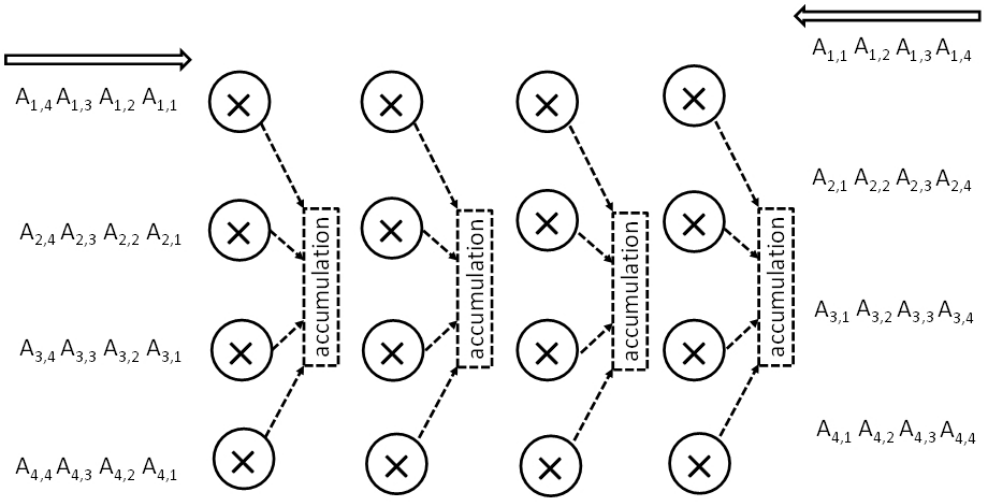
which is followed by using floating comparison logic to determine the index of the resulted vector element with largest value. BRAMs are used to hold the values of the calculated the defined parameter vector u, the primal solution vector ps, and its respective sign vector.

8.6.1.2 Update direction computation process

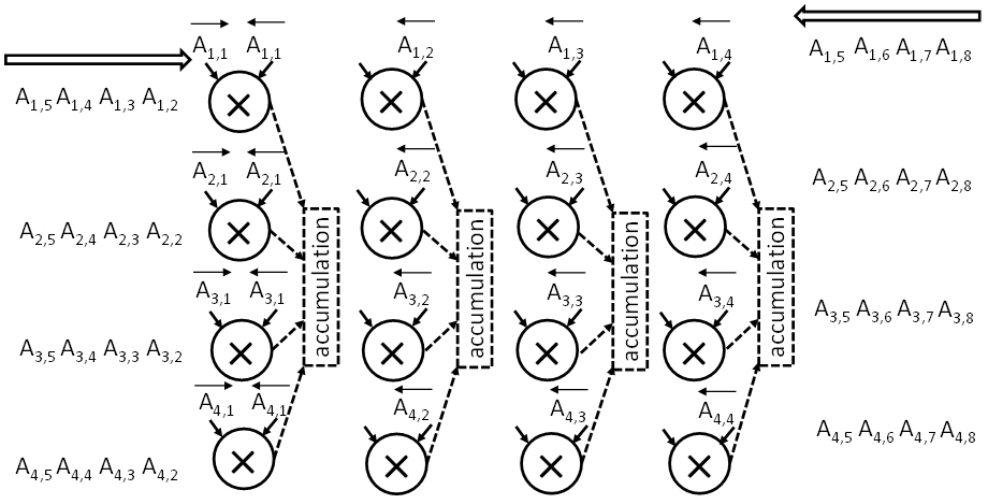
The update direction is computed by Cholesky Decomposition (refer B.2), which requires the input matrix to be symmetric. The matrix symmetrization is performed by the architecture of the Matrix-vector computation component (shown in Fig. 8.2), in which matrix row elements are streamed into this component from the two sides of the architecture, and this multi-layer multiplier-array is functioning as a systolic array. The elements applied to the multipliers have one clock cycle delay from layer to layer as the partial products between matrix vectors is collected across layers. Due to the resulted matrix is symmetric, only the elements of a triangular matrix are needed to be calculated. Registers are used among the multipliers in the same layer to synchronize the movement of matrix element and restrict the computations to be performed for either the upper or lower triangular part of the resulted matrix. Fig. 8.3, 8.4 and 8.5 demonstrate the first a few states of performing matrix symmetrizing process by using the Matrix-vector architecture, in which Fig. 8.3a-8.4b show the example data applying to all four layers of the architecture, while Fig. 8.5 only demonstrates the subsequent data movements in a single layer.

8.6.1.3 Step size computation process

After the completion of the initial setup and update direction computation, the optimization enters the iterative process until the convergence is reached. To compute the step size, solution vectors are used to determine the step size(s) for next "break point", and the detail computation can be referred at B.3. In our hardware solution, ds are computed by the Matrix-vector computation component.

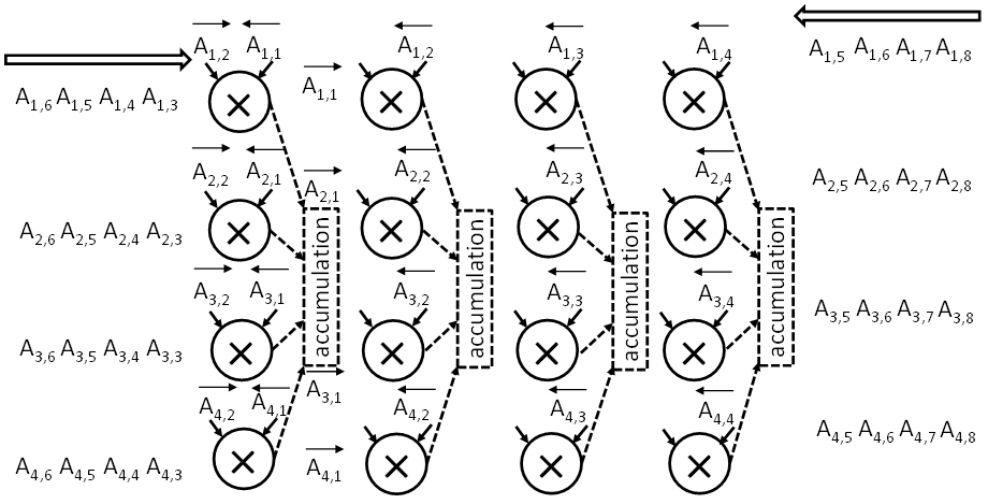


(a) The initial state

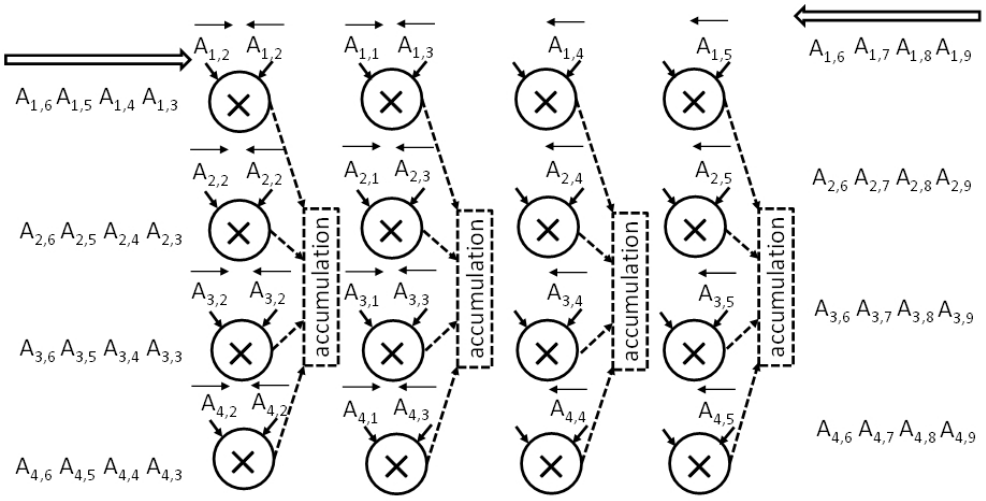


(b) The first computing state

Figure 8.3: The process of the matrix A symmetrization with the Matrix-vector architecture.



(a) The second computing state



(b) The third computing state

Figure 8.4: The process of the matrix A symmetrization with the Matrix-vector architecture (continued).

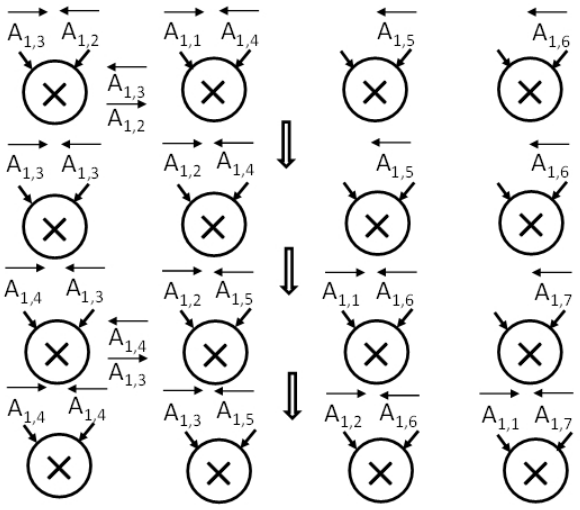


Figure 8.5: The process of the matrix A symmetrization with the Matrix-vector architecture (continued).

8.6.1.4 Support set update process

When an appropriate step size is determined, the step ϵ is updated as $\epsilon = \epsilon + \delta$, and the largest δ value will be used if the step size is a vector. Then, solution xs will be updated as $xs = xs + \delta * dirx$, which is performed on the Matrix-vector multiplication component with level-1 output used. The registers signify the support set will be updated as the bit turn from 0 to 1 indicates the corresponding element enter the support set, otherwise, the element leaves. The update of support set registers can be performed in parallel with other computations.

8.6.1.5 Cholesky rank-1 update process

Cholesky Decomposition, and the matrix symmetrization are computationally expensive with a time complexity of $O(n^3)$. To alleviate the computational burden, rank-1 update on the Cholesky factor is used as an alternative method to keep track of the direction along the Homotopy path with better efficiency. The Cholesky rank-1 update uses $O(n^2)$ operations. However, it requires to be performed every time after a single element change of the support set. Both the Cholesky Decomposition and the Cholesky rank-1 update are the performance dominant of the entire minimization process, and they have their own advantages in determining the di-

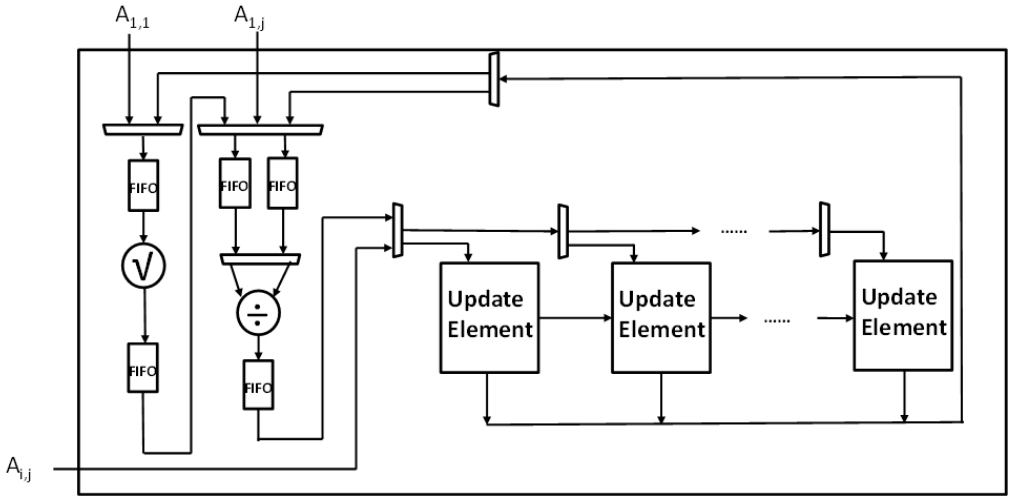


Figure 8.6: The architecture of the Matrix factorization component.

rection vector $dirx$. Cholesky Decomposition are able to be performed once on solving matrix after a few elements either entering or leaving the active support set, which can largely reduce the number of iterations for achieving convergence, especially when the support set is small or sparse. However, significant advantages on execution speed can be observed by Cholesky rank-1 update when the support set grows to a large scale. Our architecture can be configured to perform either Cholesky Decomposition and Cholesky rank-1 update dynamically with the same hardware resources. The Cholesky rank-1 update (refer B.5) includes the processes for support set element insertion and deletion [Sjöstrand (2005)], and they are calculated by both the Matrix-vector computation component and the Matrix-factorization component, in which the Matrix-vector computation component is used to perform matrix-vector calculations required in the update process.

8.6.1.6 Stop criterion evaluation process

The minimization process will be terminated as ϵ reaches 1. Then, the final step is updated as Algo. 11, after which the Matrix-computation component is used to perform simple vector operations for producing the final output solution vector xs.

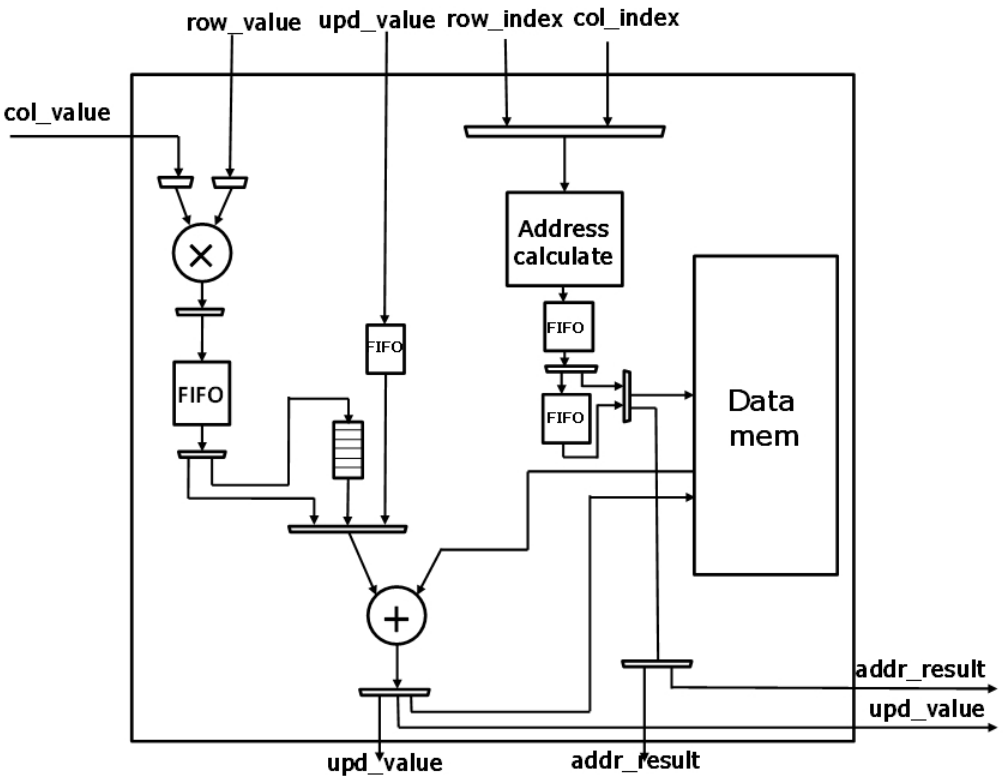


Figure 8.7: The architecture of update element in the Matrix factorization component.

8.6.2 The Matrix factorization component

The Matrix factorization component provides hardware solutions for full Cholesky Decomposition or Cholesky rank-1 updates on symmetric matrix, and matrix inverse on triangular matrix. Both of Cholesky Decomposition and Cholesky rank-1 updates mainly involves with element-wise operations of division, square roots, and multiplication-addition, while matrix inverse on triangular matrix is computed by parallel backward substitution. The architecture of the Matrix factorization component is introduced in Fig. 8.6, which contains division, square root operators, and a series of update elements. The update element (see Fig. 8.7) performs updates on vector elements if they are affected by computations on the elements from the same row or column which they belong to. As all the computational operators are fully pipelined with floating-point operations, each update element is responsible for the calculations on elements in a vector or matrix partition. BRAMs are used to temporarily hold matrix or vector elements in each update element, and finite state machines are defined to manage the move-

ment of the element and update parameters. To synchronize the operations between various computational component within this architecture, FIFOs are used to transfer the data and its respective matrix or vector indexes.

8.6.2.1 Cholesky Decomposition

To determine the update direction, Cholesky Decomposition is performed. In the update direction computation process, after the matrix is symmetrized, the resulted matrix data are sent to the Matrix factorization component for Cholesky Decomposition. The Cholesky Decomposition updates the matrix diagonal and off-diagonal elements with other processed matrix elements from the same column and row vectors before they are applied to the regularization process with square roots and division. Eq. 8.10 and Eq. 8.11 give the detail calculations of Cholesky Decomposition, which process is conducted vector by vector with diagonal element calculated first, and followed by the calculations of vector off-diagonal elements. At runtime, vector elements are averagely distributed to the *Update elements*, in which locally stored *Update parameters* are used to prepare the elements for regularization operations. After the completion of updates, vector elements are transferred to designated FIFOs, the buffers to pipelined square roots and division operators. Then, the calculated elements are further coupled with other processed elements to modify the value of the Update parameters, which are hold locally in the BRAMs of the Update elements. For example, matrix element $A_{3,7}$ is first preprocessed by the Update parameter with indexes of $(3, 7)$, and then divided by its corresponding diagonal element. After the $A_{3,7}$ is factored, it is used to be paired with other factored element in the same vector such as $(A_{3,8})$ to change the value of corresponding Update parameter at $(7, 8)$.

$$R_{i,i} = \sqrt{A_{i,i} - \sum_{k=1}^{i-1} R_{k,i}^2} \tag{8.10}$$

$$R_{i,j} = \frac{A_{i,j} - \sum_{k=1}^{i-1} R_{k,i} * R_{k,j}}{R_{i,i}} \tag{8.11}$$

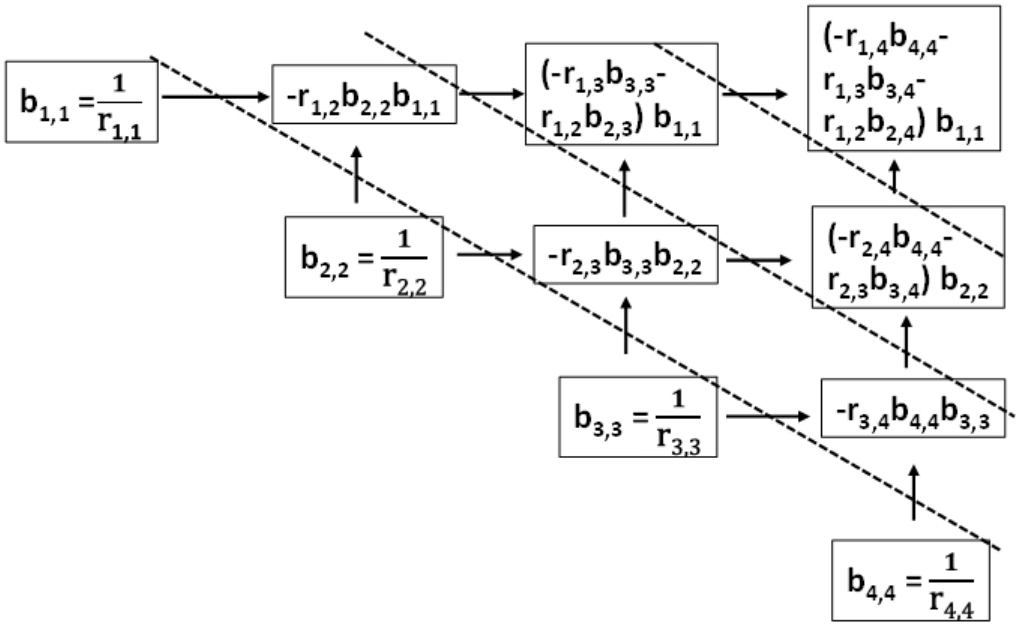


Figure 8.8: The diagram of the matrix inverse process.

8.6.2.2 Matrix inverse

Cholesky Decomposition produces an upper triangular matrix, which is then applied by backward substitution for a matrix inverse. The matrix factorization component is used for matrix inverse, in which all the diagonal elements are first performed with division operations in pipelining. Then, all the remaining off-diagonal elements are averagely distributed to the Update Elements diagonally, and update elements on every diagonal vector by vector with data movements take place among the Update Elements accordingly. Fig. 8.8 provides a diagram of the matrix inverse.

As the matrix size grows larger than a threshold such as 64, the matrix vectors are partitioned, which every sub-vector is calculated with symmetrization, decomposition, and inverse subsequently. After the matrix inverse is obtained, they are sent to the Matrix-vector multiplication component to produce the updated direction vector with matrix-vector multiplications. The overlap of matrix symmetrization, decomposition, inverse, and multiplications of various sub-vectors help improve the parallelism.

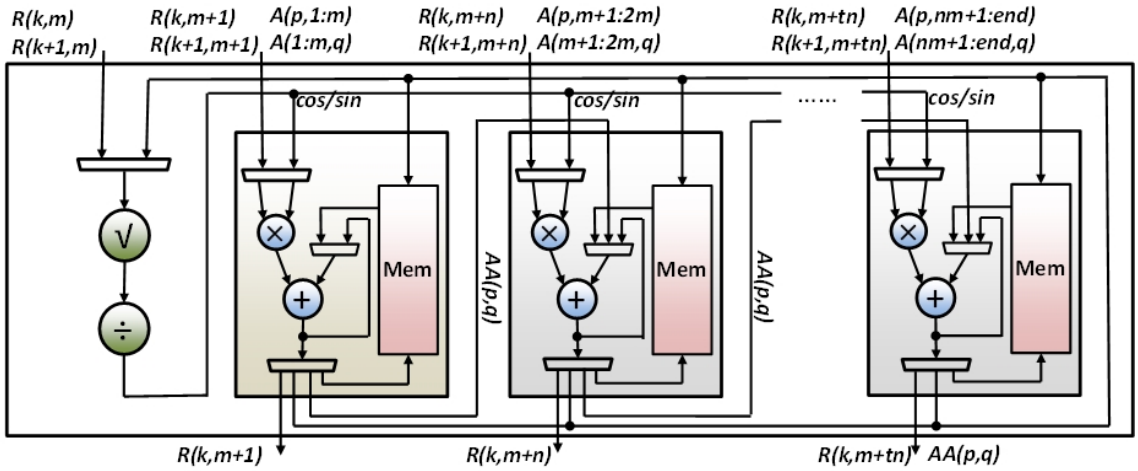


Figure 8.9: The diagram of Matrix factorization component with reconfiguration to Cholesky rank-1 update.

8.6.2.3 Cholesky rank-1 update process

In this proposed hardware solution, the Matrix-vector computation component is used to conduct the matrix-vector multiplication, while the update of the inverse of Cholesky factor R, and vector-vector operations for calculating the new diagonal elements of Cholesky factor R and direction vector $dirx$ is performed by the Matrix-factorization component. When compute parallel vector-vector multiplication, the Matrix factorization component is reconfigured to a 1-dimensional systolic-array, which is demonstrated in Fig. 8.9. Vector elements are averagely distributed to Update element, and local memory in each Update element hold partial results of vector operations. After partial results of an Update element is calculated, it will be moved rightwards to be added up with other partial results, and the final result is produced once the accumulation reaches the last Update element.

In the Cholesky rank-1 deletion update, the Givens rotation is performed by the Matrix factorization component, in which division and square root operators are used to calculate the rotation angle parameters cos and sin, while Update elements are used to update affected vector elements with rotation angle parameters (see Fig. 8.9). The vector elements are moved leftwards for rotation iteratively.

8.7 Implementation and Evaluation

8.7.1 Implementation and Experimental Setup

We evaluate our proposed hardware solution for Homotopy Algorithm ℓ_1-norm minimization with Convey Computer HC-2 system [Convey Computer HC-2 (2012)], a hybrid-core system coupled standard Intel based server with Xilinx Virtex-5 XC5VLX330 FPGAs as co-processor. To optimize the speed, we generate the double-precision floating-point computational cores (adder, multiplier, divisor, square roots, floating-point comparator) by using Xilinx Coregen generator [Xilinx Inc. (2012)]. In the Matrix-vector computation component, four layers of multiplier-array are implemented, in which 16 multipliers and 20 adders are used, among which 5 adders are employed as floating-point accumulators. In the Matrix factorization component, 1 divider, 1 square-root and 8 Update elements are implemented, and in each Update element, 1 adder, 1 multipliers, and simple dual port memory are equipped. In our design, the pipeline latency of each type of core are configured as 9, 14, 57, 57 clock cycles for multiplication, addition, division, and square roots, respectfully. BRAMs are used to store the part of input matrix A, solution vectors, Cholesky factor R and its inverse locally. To improve the local memory bandwidth and throughput, BRAMs are organized into groups for the storage of on-chip matrices. By performing the synthesizing and implementation with Xilinx design tools, our placed and routed design consumes 67.8% of the slice LUTs, 43.75% of DSP48Es and 83.3% of BRAMs with a maximum frequency of 157 MHz. Re-synthesis is necessary if the property of input matrices and architectural configuration are changed. We evaluate our system efficiency by running our implementation with both random datasets of [Asif and Romberg (2013)] and Yale face matrix [Georghiades et al. (2001); Lee et al. (2005)] at 150 MHz, which is a typical frequency used in HC-2.

8.7.2 Performance analysis

In this section, we first profile the ℓ_1-norm minimization, in which input matrices are of various dimensions. As demonstrated in Fig. 8.10, more than 70% and 20% of execution time for ℓ_1-norm minimization is used to perform iterative Cholesky rank-1 update and compute

step size respectively. Meanwhile, the percentage of time consumption for Cholesky rank-1 update in entire ℓ_1-norm minimization increases as the ratio of input column dimension over row dimension grows. In practice, the Cholesky rank-1 update and step size computing are the performance dominant for ℓ_1-norm minimization.

To analyze the performance, we evaluate input underdetermined system with the dimensions from 512 to 4096 with certain sparsity level and a signal noise ratio of 40. In this paper, we name the single Cholesky Decomposition that is performed after n (n¿1) elements entering or leaving the active support set as rank-n update. We have also configured the system to allow rank-n update with full Cholesky Decomposition in one iteration. The performance for input matrices with row dimension as 512 and various column dimensions is demonstrated in Fig. 8.11, in which the number of elements that is allowed to add to the support set in a single iteration can be configured. By configuring the number n for direction update by rank-n, n elements will move into the support set if there is a continuous n element that satisfy the condition to become active, otherwise iterative rank-1 update will be performed. The experimental results indicate that the execution time grows significantly as the column dimensions increasing due to more iterations are needed for convergence. With the configuration of n elements are allowed to be added into the support set, the performance is improved as the number of iterations that need to be executed are largely reduced. However, many input dataset properties affect the system performance include the matrix dimensions, sparsity level, and data pattern. When we configure the experimental datasets with rank-n update and the n grows, the performance is degraded due to the number of qualified rank-n update iterations decrease.

In Fig. 8.12, the performance of ℓ_1-norm minimization with column dimension as 4096 and various row dimensions is demonstrated. Also, the rank-n configuration is performed. When the column dimension grows, the execution time increases significantly, and the column dimension determines the maximum dimension of the support set, which affect the scales of intermediate matrix computation and operation (e.g. matrix factorization, matrix inverse) within the ℓ_1-norm minimization process. Besides, the on-chip storage is limited, and the column dimension is a major factor in determining the dimensions of Cholesky factor R. If Cholesky factor R can be fully maintained on-chip, the I/O burden will be highly alleviated. Similarly, when rank-n

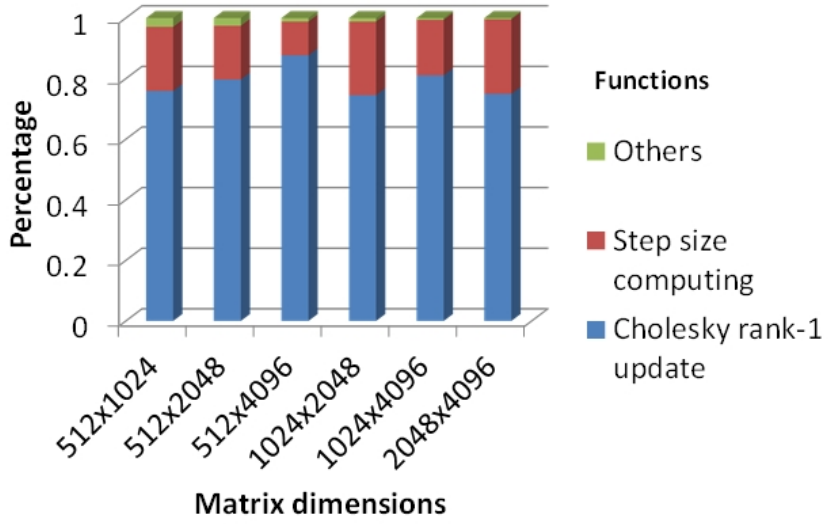


Figure 8.10: The profiling results of ℓ_1-norm minimization with various dimensional input datasets.

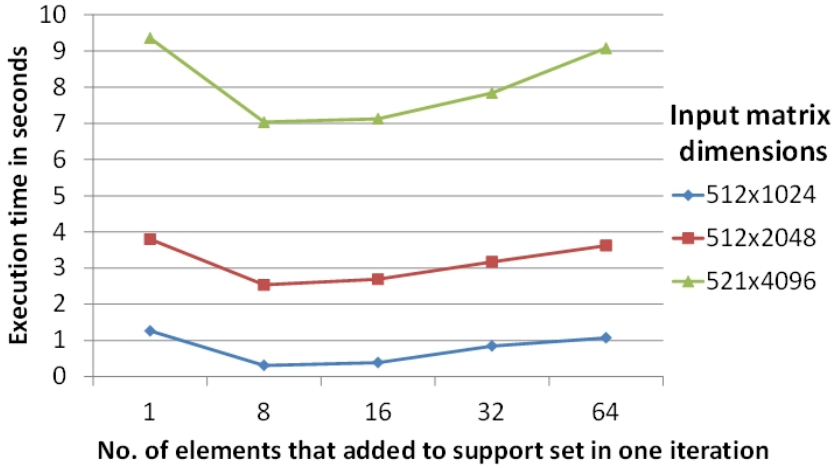


Figure 8.11: ℓ_1-norm minimization computation time (in seconds) for matrices with row dimension as 512 and various rank-n update configuration.

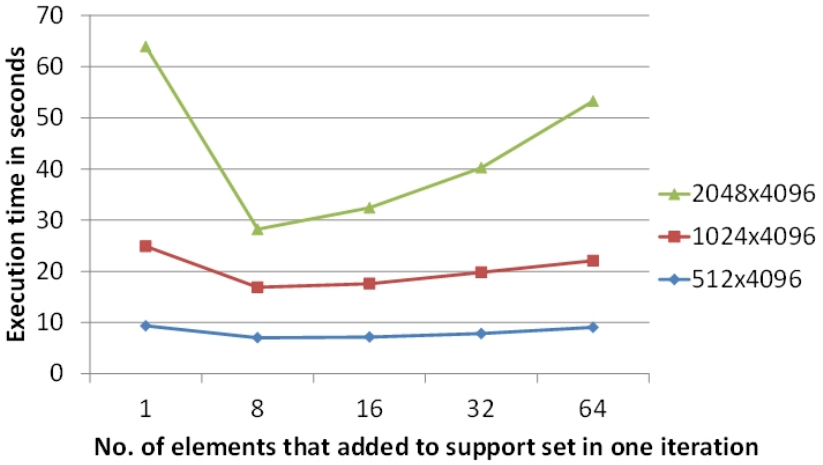


Figure 8.12: ℓ_1-norm minimization computation time (in seconds) for matrices with column dimension as 512 and various rank-n update configuration.

is configured, the performance will be improved as the number of iterations for convergence is decreased. The configuration of the number of n for best performance is determined by many factors such as the properties of input data.

In Fig. 8.13, we have demonstrated the number of iterations required for convergence for matrices with various dimensions and update configurations. After the configuration of the rank-n update in one iteration (n is shown in the parenthesis after the dimensions), there will be three types of iterations: rank-1 add, rank-1 delete, and rank-n add. In the same random data pattern and sparsity level, the total number of iterations will be reduced as the more than 1 elements are allowed to be added into the support set in one iteration. However, when the configuration parameter n increases over a threshold, the number of qualified rank-n update iteration will be decreased, and the total number of iteration grows. In our architecture, we implemented both parallel Cholesky Decomposition and Cholesky rank-1 update, and the number of iterations have significant influence on the performance. We have also demonstrated the accuracy and potential maximum size of the support set based on the rank-n update configuration in Table 8.1, which indicates limited accuracy sacrificed with the rank-n update allowed for the input data with similar property and pattern.

The comparisons of execution time between our implementation and Matlab ℓ_1-norm minimization program can be seen in Fig. 8.14, in which dimensional speedups of our design

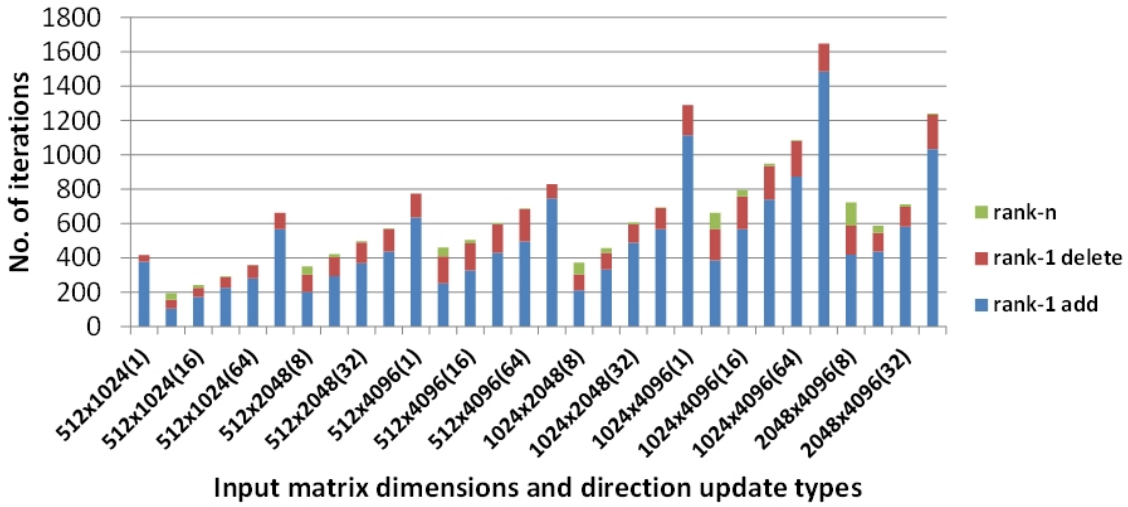


Figure 8.13: No. of iterations for convergence with different matrix dimensions and update configurations.

compared to the Matlab 7.10.0 ℓ_1-norm minimization program running on a 2.2 GHz dual core Intel Xeon processor is demonstrated. We have shown the speedups of our ℓ_1-norm minimization implementation compared to both the Matlab ℓ_1-norm minimization program with rank-1 update only and with rank-n update configured. By analyzing those data points in Fig. 8.14, our architecture shows better efficiency than Matlab implementations, with a dimensional speedups that can be achieved range from 3.6× to 9.9× for matrices with dimensions from 512 to 4096.

To further analyze the performance of our hardware solution, we have compared the implementations with the real-world dataset. We have used Yale Face dataset of [Georghiades et al. (2001); Lee et al. (2005)] as our benchmark that Yale Face database contains 165 grayscale images in GIF format of 15 individuals. By configuring our architecture with rank-8 update allowed, our experimental results shows 3.89× speedup for the input matrix with dimensions of 165×1024 without significant accuracy loss.

8.8 Conclusion and Future Work

In this paper, we modify the Homotopy algorithm of ℓ_1-norm minimization for better efficiency, which provide the flexibility to add one or more elements into the support set in one

Table 8.1: Experimental matrices, properties, and their accuracy.

Dimensions	rank-n configuration	Max. size of support set	Error rate
512×1024	1	339	0.004093
512×1024	8	325	0.004774
512×1024	16	348	0.005412
512×1024	32	352	0.008545
512×1024	64	345	0.01058
1024×2048	1	665	0.003806
1024×2048	8	672	0.004166
1024×2048	16	668	0.004471
1024×2048	32	671	0.005062
1024×2048	64	654	0.00829
2048×2048	1	1323	0.003896
2048×2048	8	1310	0.004068
2048×2048	16	1345	0.004352
2048×2048	32	1323	0.004635
2048×2048	64	1325	0.004906

iteration. We propose an FPGA implementation for ℓ_1-norm minimization with Homotopy algorithm, which can be dynamically configured to perform either parallel Cholesky Decomposition or rank-1 update. By analyzing the performance of our proposed hardware solution, our architecture shows dimensional and application-dependent speedups from 3.6× to 9.9× compared to optimized software solution with reasonable accuracy achieved.

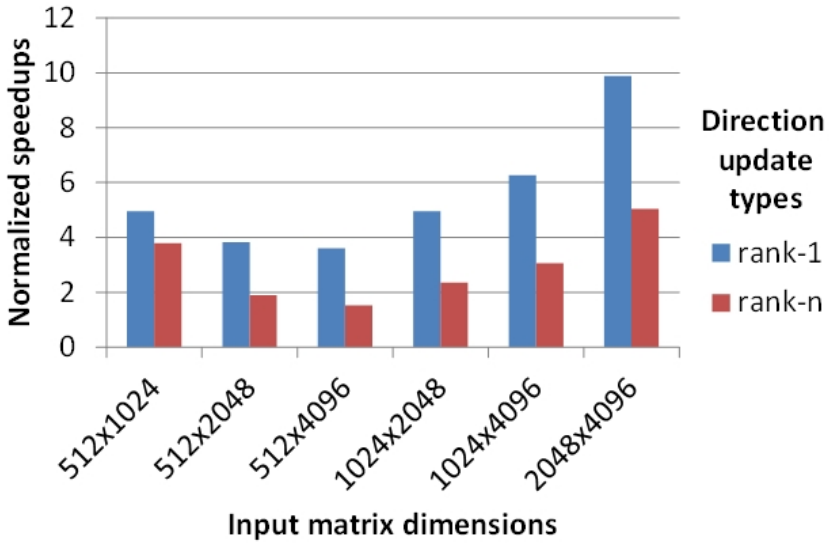


Figure 8.14: The dimensional speedups of our hardware ℓ_1-norm minimization solution over Matlab implementation.

CHAPTER 9. FUTURE WORK DISCUSSION

Previous chapters have introduced our work of proposing efficient FPGA-based architectures for matrix decomposition (e.g., EVD, SVD, QRD, Sparse LUD) and applications (e.g.,Latent Semantic Indexing and ℓ_1-norm minimization). The future research work aims to explore further optimization by leveraging the characteristics of the FPGAs. The potential future work will be primarily developed in two directions: 1), explore hybrid architectures, the FPGA-based solution to implement multiple algorithms with same hardware resources, which can be configured to execute the most appropriate algorithm at runtime; 2), explore application-specific architecture to customize the architecture according to the demands of specific applications for better performance both in speed and power consumption.

9.1 Hybrid architecture

The reconfigurability of FPGAs provides the opportunity to implement multiple algorithms with the same set of hardware resources. In mathematics, matrix decomposition problems normally can be solved by various methods, and each of which has its own advantages. Besides, some of matrix decomposition are computed iteratively until convergence is reached, and better performance can potentially be achieved if the most proper algorithm is applied in each iteration. For example, to compute SVD, classic algorithms include Householder transformation, two-sided Jacobi rotation and Hestenes-Jacobi rotation. The Householder transformation is featured with efficient vectored operations, while Jacobi rotations are easily mapped to parallel platforms. In our research, we explored the FPGA-based SVD architectures for two-sided Jacobi rotation and Hestenes-Jacobi rotation separately. However, as the dataset changed to a triangular matrix after the first iteration of Hestenes-Jacobi operation, the systolic array

structure we used for two-sided Jacobi rotation would bring benefits in efficiency for the rest iterations. Thus, a hybrid architecture that can be dynamically configured to the multi-layer arrays (see Fig. 4.2) and the one-dimensional systolic array (see Fig. 3.4) would potentially offer additional performance improvement.

9.2 Application-specific architecture

Applications are various in terms of the properties of the input datasets. The properties of the input data such as dimensions, sparsity and data pattern have significant impact on the performance of the entire system. The flexibility of FPGAs offer the opportunity to customize the hardware solution to a more application-specific architecture, which deeply optimize the performance to meet the requirement of a specific application. By understanding a specific application features, the limited hardware resources of FPGAs can be arranged accordingly for better performance. For example, in model reduction [Constantine et al. (2014)], the input dataset are featured as tall-and-skinny matrix, a rectangular matrix who has far more rows than columns ($m \gg n$). A customized application-specific architecture targeting the tall-and-skinny matrix can largely reduce the burden of I/O usage, which has become the wild recognized bottleneck for performance improvement on FPGAs in many scientific and engineering applications.

CHAPTER 10. CONCLUSION

Matrix decomposition is computationally expensive, and it has been widely used in data mining and signal processing applications. In this dissertation, FPGA-based architectures have been proposed to perform Eigenvalue Decomposition, Singular Value Decomposition, QR Decomposition, and sparse LU Decomposition. By using systolic array architecture, an efficient architecture for floating-point Eigenvalue Decomposition is demonstrated, while an FPGA implementation of the Hestenes-Jacobi algorithm is presented to perform Singular Value Decomposition. In addition, a reconfigurable architecture for QR Decomposition is proposed, which can be dynamically configured to perform either Householder transformation or Givens rotation in a manner that takes advantage of the strengths of each. To improve the efficiency of analyzing sparse dataset, a configurable architecture is introduced for sparse LU Decomposition, which can analyze both symmetric and asymmetric sparse matrices with arbitrary sparsity patterns.

This dissertation also applies the architectures of matrix decomposition to perform Latent Semantic Indexing and ℓ_1-norm minimization. By extending the Hestenes-Jacobi SVD architecture, a deeply pipelined reconfigurable architecture is developed, which performs the matrix factorization, dimension reduction, computation of vector cosine similarity, and the ranking of documents simultaneously. Besides, in this research, FPGAs are further used to implement ℓ_1-norm minimization with Homotopy algorithm, in which symmetric LU Decomposition is the performance dominant. Our experimental results using an FPGA-based hybrid acceleration system indicate the efficiency of the proposed architectures that offer one or two orders of magnitude dimension and application-dependent speedups over optimized software implementations. The analysis of the experimental results indicates that, for some applications, the I/O bandwidth starts to limit the performance improvement as the dimensions of input matrices

grow. In addition, FPGAs have shown the advantages in handling datasets with special data patterns such as tall-and-skinny or sparse. In the future work, further optimization can be accomplished through exploring hybrid architectures and application-specific architectures with the help of the reconfigurability and flexibility of FPGAs.

APPENDIX A. HIGH PERFORMANCE COMPUTING PROCESSORS AND CONVEY HYBRID-CORE COMPUTING PLATFORM

As the volume of information is increasing continuously at an astonishing rate, the runtime of many computations grow dramatically, and conventional sequential processing unit is unable to satisfy the performance requirements of many real-world applications. Many-core processors have shown the promise in providing high performance solutions for many computing problems. With the help of the parallel programming technology, problems are partitioned into subproblems, and multiple light-weight threads can be executed concurrently on a single chip. To improve the systematic performance, multi-core processors commonly act as co-processors to assist the conventional processing unit in tackling the computational-intensive problems with parallel processing. In this appendix, an overview of the most popular high performance accelerators, such as GPUs [Keckler et al. (2011)], Xeon Phi coprocessor [Rahman (2013)], FPGAs [Hauck and DeHon (2007)], are presented. Besides, an FPGA-based hybrid-core computing platform–Convey system [Brewer (2010)] is introduced.

A.1 High performance accelerators

A.1.1 GPUs

A graphic processing unit (GPU) is a dedicated circuit implementing integrated transform, lighting, triangle setup/clipping, and rendering engines on a single-chip processor [Keckler et al. (2011)]. GPUs were originally designed to perform the simplest pixel-drawing functions, and they are progressed to building transforms, lighting, rasterization, texturing, depth testing, and display with full 3D pipeline [Heinecke et al. (2012)]. To accelerate many real-world applications, the general purpose GPU (GPGPU) emerges that these graphics-optimized ar-

chitectures, which are massively parallel that consist of thousands of small and efficient cores designed to handle multiple tasks simultaneously, are used to perform non-graphics processing. As commodity data-parallel processors, GPUs are featured with tremendous computational capacity and rapid growth curve, and they have shown the promise in performing domain-specialized data-parallel computing, whose performance far outstrip traditional CPUs [Luebke and Humphreys (2007)].

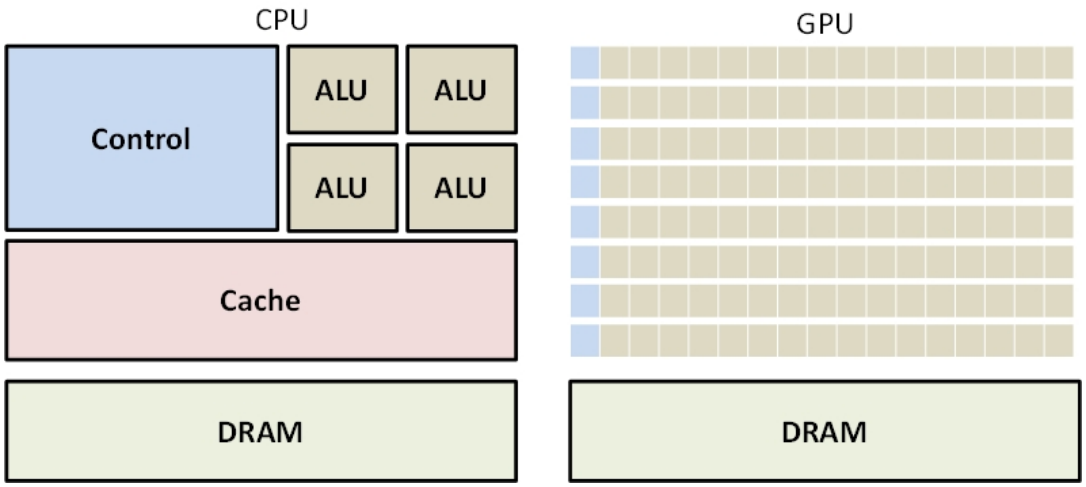


Figure A.1: Design philosophies behind CPUs and GPUs [Kirk and Hwu (2010)]

To achieve high throughput, GPUs perform stream processing to solve problems that tolerate high latencies. In Fig. A.1, the design philosophies behind CPUs and GPUs are illustrated, and compare to traditional CPU, GPUs uses more transistor area for computing units and less area for cache [Glaskowsky (2009)]. Fig. A.2 demonstrated an example GPU architecture with CUDA-capable, which consists of an array of highly threaded streaming multiprocessor (SMs), whose quantity could be vary from GPU to GPU [Kirk and Hwu (2010)]. GeForce 8800 is a CUDA-capable GPU, and it has 16 stream multiprocessors, each of which contains 8 unified streaming processors (cores). A streaming processor (core) is a fully pipelined and multithreaded computational engine that support 96 threads, couple with a register file including 1024 scalar 32-bit registers. It implements all 32-bit and 64-bit integer arithmetic, comparison, conversion, logic instructions, as well as IEEE-754 single-precision floating-point operations. Besides, a stream multiprocessor contains two special function unit and a IEEE-754

double-precision function unit to perform the computations of special functions such as the transcendental functions, reciprocals, square roots, and IEEE-754 double-precision floating-point operations, respectively. During runtime, workloads are partitioned into blocks of threads, and a group of 32 threads are named as a warp, which is the primitive unit of scheduling and execute the same instruction at a time [Lindholm and Oberman (2007)].

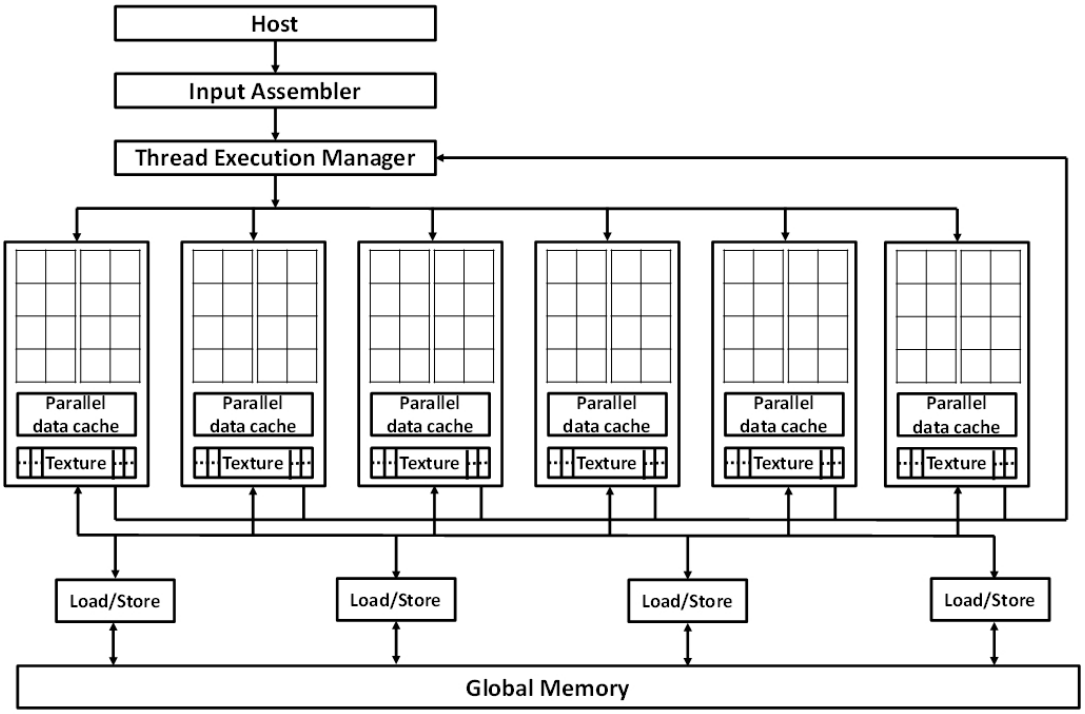


Figure A.2: An example of CUDA-capable GPU architecture [Kirk and Hwu (2010)]

GPUs implement multithreading technology, and they rely on sufficient threads to hide latency instead of using multi-level cache. With the extensive parallelism, GPUs are often used as a powerful computational engine to acheive high performance. Compared to conventional CPU, the main memory of GPU is designed for high bandwidth rather than low latency [Kirk and Hwu (2010)].

A.1.2 Xeon Phi coprocessor

Xeon Phi is a coprocessor introduced by incorporating Intel Many Integrated Core Architecture [Rahman (2013)]. The Intel Xeon Phi coprocessor provides up to or more than 50

in-order cores, in which individual cores are associated with each other by passing data through fully coherent caches, and communicate in a bidirectional ring. The processor core is shown in Fig. A.3, which consists of instruction decoder, scalar unit, vector unit, related registers and hierarchical caches [Jeffers and Reinders (2013)]. The scalar unit is pipelined and Intel Pentium processor based, and it is enabled with dual issue of scalar instructions with throughput of one-per-clock cycle. A fully functional multi-threaded vector unit is included in the Xeon Phi coprocessor, and the existing vector unit supports 512-bit SIMD Instructions with numerous 512-bit wide vector registers are available [Rahman (2013)]. In a single core, there are four hardware threads, each of which issues instruction in turn, and execute instruction in round-robin to hide the latency. Fully-coherent L1 and L2 caches are equipped with each processor core [Jeffers and Reinders (2013)].

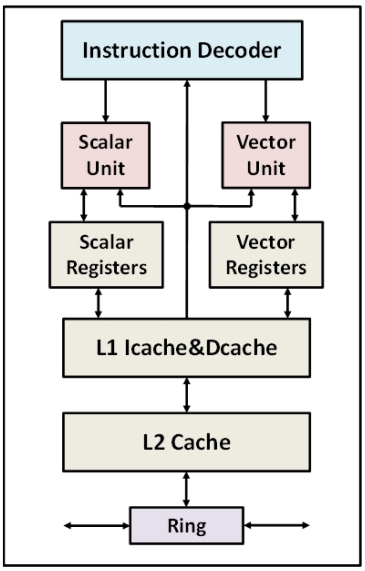


Figure A.3: The architecture of Intel Xeon Phi Coprocessor core [Jeffers and Reinders (2013)]

A.1.3 FPGA and reconfigurable computing technology

FPGA is an integrated circuit composed of configurable logic blocks (CLB) and programmable interconnections (as shown in Fig. A.4) [Hauck and DeHon (2007)]. A configurable logic block (CLB) can be programmed individually to perform unique function. It normally consists of numerous logic cells, each of which typically includes a 4-input LUT, a full adder

and a D-type flip-flop. As shown in Fig. A.5, a 4-input LUT is made up by two 3-input LUTs. By using multiplexers, CLB can be operated in normal mode, which is defined mainly for general logic applications and combinational functions. Another mode is named as arithmetic mode, and it implements adders, counters, accumulators, and comparators [Hauck and DeHon (2007)]. Recent FPGA platforms employ 6-input LUT with fully independent inputs instead of conventional 4-input LUT, which optimizes trade-off between critical path delay and design die size [Cosoroaba and Rivoallon (2006)].

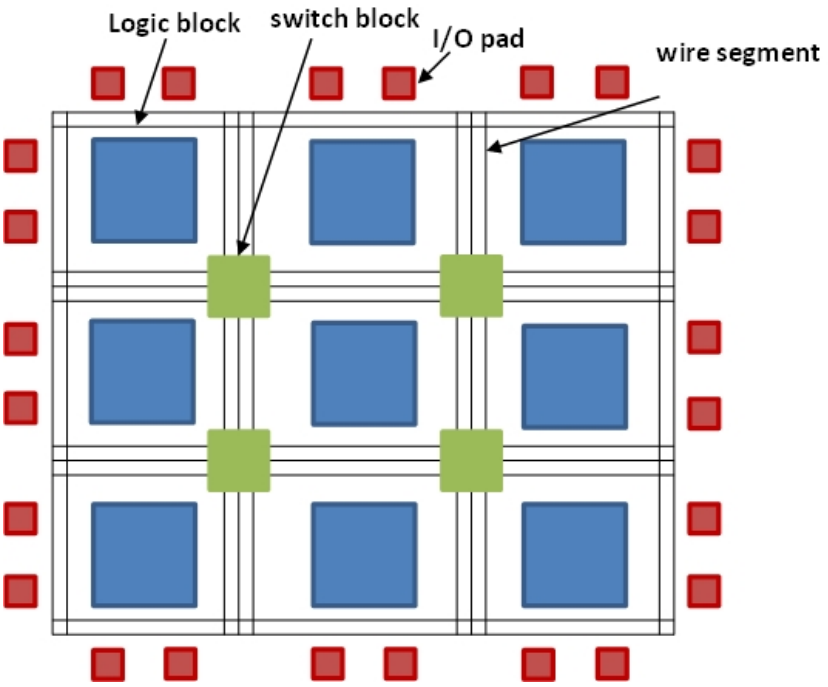


Figure A.4: The general architecture of FPGAs [Kuon et al. (2008)]

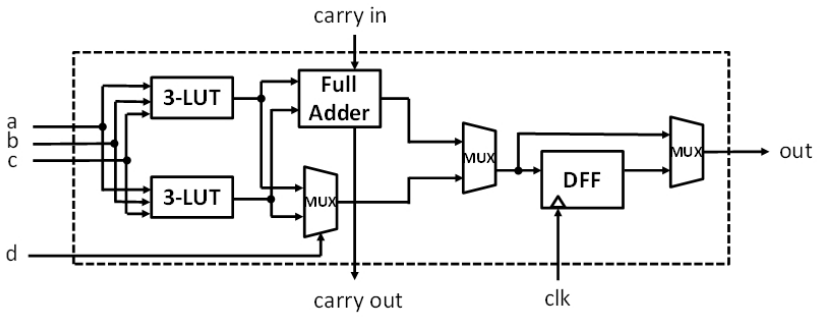


Figure A.5: An example architecture of FPGA logic block [Kuon et al. (2008)]

Each input pin has a connection to one side of the logic block, and the output channels to the right and below of the logic block connect to routing wires. Both the input and output pins of logic block are able to be connected to any wiring segment in the adjacent channels. In the similar way, the I/O pad can also communicate with adjacent channels by establishing connections between I/O pad to any wiring segment in them [Kuon et al. (2008)].

The routing of FPGA is unsegmented that a wiring segment connects one logic block to a switch box only. An example topology of switch box is demonstrated in Fig. A.6, in which longer paths can be built by configuring programmable switches in a switch box. In practice, to archive higher speed interconnection, longer routing wires that span numerous logic blocks are used in many of today's FPGA implementations [Kuon et al. (2008)].

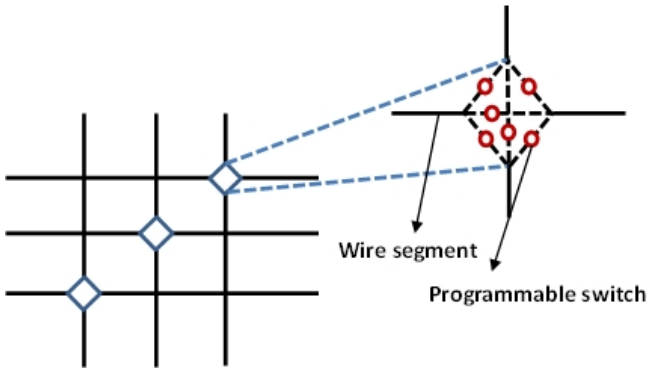


Figure A.6: The example topology of switch box [Kuon et al. (2008)]

Researches were conducted to evaluate the potentials of using the input pin doglegs to connect more than one routing wire segment to the same logic input pin. To establish more efficient connections, the actual routing architecture of SRAM-based FPGAs implement a buffer between routing tracks and input pins, instead of using the input pin doglegs. Similarly, a set of independent pass transistors is replaced by multiplexers to build connections between wire segments and input pins for the sake of area optimization [Hauck and DeHon (2007)].

A.2 Convey hybrid-core computing platform

Hybrid-core is considered as a breakthrough technology in solving the issue of power limitation happen to conventional processor. The Convey's hybrid-core platform implements het-

erogeneous architecture by combining the commodity processors with an application-specific hardware design. To expand the capabilities of the commodity processors, the Convey hybrid-core system implements reconfigurable coprocessor, by which the instructions are executed as extensions to the x86 instruction set, and customizable for the requirement of application-specific computations [Convey Computer (2012)].

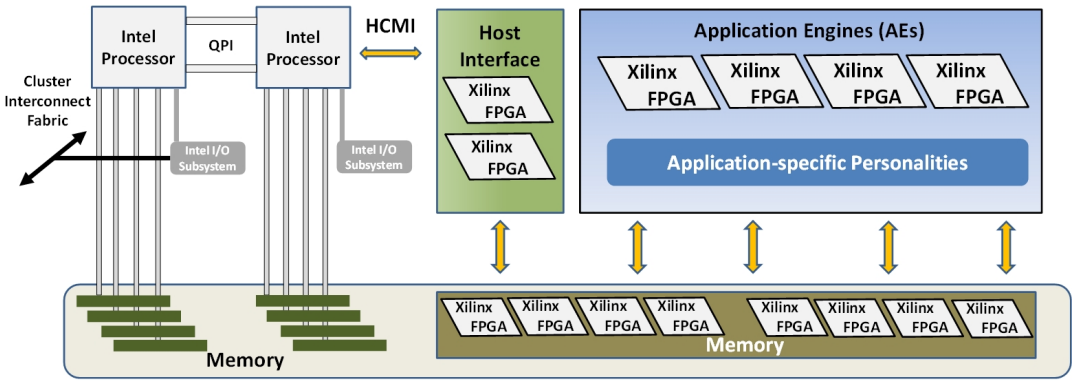


Figure A.7: The Convey Hybrid-Core Architecture [Convey Computer HC-2 (2012)]

As shown in Fig. A.7, the Convey computer implements commodity FPGAs as coprocessor coupled with standard multi-core Intel Xeon processors. The host processor and coprocessor have their own local memories physically. However, all the local memories are globally addressable with their coherence protected. The Convey computer outperforms conventional processors with improved application performance, enhanced functionality, and increased operational efficiency. It accelerates the computations by parallelizing the computational-intensive kernels of the application with the FPGA-based coprocessor. According to the user manual of Convey computer, it provides a wide range of configurations, with which developers can select from a combination of models, memory configuration, and I/O devices to meet the application-specific demands [Convey Computer (2012)].

HC-2 and HC-2^{ex} are the latest models of Convey computer, both of which consist of host system and coprocessors. The host system is equipped with Intel processors, on which industry-standard Linux is running. Meanwhile, host processor supports standard networking and interconnect fabrics capable for clustering. The coprocessor system contains multiple FPGAs that perform application-specific operations (called personalities) [Convey Computer

HC-2 (2012)]. The Convey platform supports different instruction sets in a common hardware platform largely reduce the time for developing the application or algorithm-specific instruction sets. At runtime, the personalities are wrapped into components, which can be reloaded dynamically, and customize the supported instruction set based on the application-specific requirement. A personality contains the base instruction set for scalar operations and execution control, and a set of extended instructions that are designed for application-specific manipulations [Convey Computer PDK (2012)].

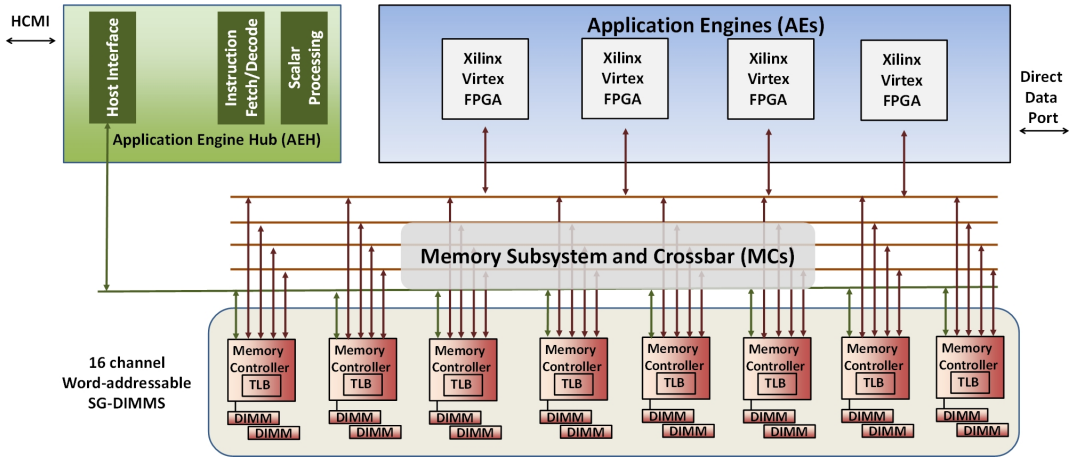


Figure A.8: The Convey HC-2 memory subsystem [Convey Computer HC-2 (2012)]

As shown in Fig. A.8, the coprocessor consists of three major components that the Application Hub (AEH), the Memory Subsystem (MCs), and the Application Engines (AEs). Among them, AEH acts as the control center for the coprocessor, and it is responsible for the implementation of hybrid-core memory interconnect (HCMI) to the host system, the management of communications between the host and coprocessor, and the routing of coprocessor memory access requests to proper MCs [Convey Computer HC-2 (2012)].

In both of the Convey HC-2 and HC-2^{ex} computer, eight MCs with a total of 16 DDR2 memory channels are equipped, which optimizes the communications between AEs and the physical memory of coprocessor with highly parallel channels and large bandwidth. By leveraging Convey designed Scatter-Gather DIMMs, the system support random memory access, which can largely improve the performance for applications [Convey Computer HC-2 (2012)].

The application-specific computational operations are implemented on the Application Engines (AEs), which are considered as the core of the coprocessor. On the Convey platform, multiple AEs are equipped in the coprocessor. AEs are connected to the AEH via a command bus, who is responsible for the communication of the opcodes and scalar operands. AEs are also connected to every MCs through a network with point-to-point linkage [Villarreal et al. (2010)]. AE instructions are delivered to all of the AEs, and their executions are controlled by the personality according to its design. In a single AE, the FPGA can operate many functional components concurrently, and the performance optimization of coprocessor has been transformed into the pursuit of high degree of parallelism [Convey Computer HC-2 (2012)].

APPENDIX B. HOMOTOPY ℓ_1-NORM MINIMIZATION ALGORITHM

The Homotopy algorithm is an iterative process that construct a decreasing sequence of regularization parameter, and identify "break points" along the path with the variable associated support set updated by adding or removing components [Yang et al. (2010)]. In this chapter, the modified Homotpy ℓ_1-norm minimization algorithm is described [Asif and Romberg (2014)], which introduces more flexibility of adding one or more active elements to the support set in one iteration for better efficiency.

B.1 Initial setup process

Input: A full rank matrix $A \in \mathbb{R}^{m \times n}$ $(m < n)$, and a vector $b \in \mathbb{R}^m$

`/* Compute the initial primal and dual solution */`

$ps \leftarrow -A^T * b$

$ds \leftarrow 0$

$xs \leftarrow 0$

`/* Initialize support set setup */`

$[value\ gamma_x] = max(abs(ps))$

$sign_x = zeros(N, 1)$

$sign_x(gamma_x) = -sign(ps(gamma_x))$

$idelta \leftarrow gamma_x(1)$

$flag \leftarrow 1$

`/* Add element to active support set when flag equal to 1, otherwise remove element */`

`/* Update defined parameters (`u `is defined as` $-Weight * sign(ps) - A' * (A * ps - b)$ `) */`

$u = -Weight. * sign_x - ps$

$ps = ps + u$

`/* Initialize the step */`

$epsilon = 0$

Algorithm 6: Solution and support set initialization

The *Initial setup process* is responsible for computing the initial primal and dual solution, and determine the first active element in the initial support set. By assuming no initial solution is provided, the ℓ_1-norm minimization process is started by the matrix vector multiplication of $-A^T * b$. Comparisons are performed to locate the first element to be added in the active support set. The detail process of solution and support set initialization is described in Algo. 6.

B.2 Update direction computation process

Input: A full rank matrix $A \in \mathbb{R}^{m \times n}$ $(m < n)$, and a vector $b \in \mathbb{R}^m$
`/* Cholesky Decomposition on` A `*/`
$R \leftarrow chol(A(:, gamma_x)^T * A(:, gamma_x))$
`/* Update of direction with triangular matrix inverse */`
$dirx \leftarrow R^{-1} * ((R^{-1})^T) * u(gamma_x))$

Algorithm 7: Initial computation of update direction

After the completion of the Inital setup process, to determine the starting update direction, the computation of the update direction is performed by Cholesky Decomposition and triangular matrix inverse (see Algo. 7). Due to Cholesky Decomposition requires the dataset to be symmetric, the support set based submatrix of A has to be symmetrized first, after which Cholesky Decomposition is applied to. Then, the resulted upper triangular matrix R is calculated for its inverse through backward substitution. The Update direction computation process is ended by matrix-vector multiplications between the triangular matrix inverse R^{-1} and the defined parameter u. The Homotopy algorithm for ℓ_1-norm minimization starts with empty active support set, and the initial update direction computation only involves with numerical elements operations. However, as the size of support set increases iteratively, the update direction computations become the performance dominator for the entire process of ℓ_1-norm minimization. The full Cholesky Decomposition has to be performed to determine the moving direction of the Homotopy path if multiple new active elements are added to the support set in one iteration. Otherwise, iterative running rank-1 Cholesky update is required every time after a single support set element change. The appropriate usage of Cholesky Decomposition can largely reduce the number of iterations for convergence, and save computational time with sacrificing of reasonable accuracy.

B.3 Step size computation process

When the initial solution, support set, and optimization direction along Homotopy path is available, then the ℓ_1-norm minimization enters an iterative process until the convergence is reached. The step sizes are determined by primal, dual and original solution vectors ps, ds, xs through simple matrix-vector operations (see Algo. 8). The step sizes are determined by selecting one or more step sizes with minimum positive values. If one step size is chosen, one register is used to always hold currently minimum positive value, after which rank-1 update will be performed. Otherwise, multiple continuous step sizes are selected.

Input: A full rank matrix $A \in \mathbb{R}^{m \times n}$ $(m < n)$, the direction vector $dirx$, the defined vector u, and the solution vectors ps, ds, xs

`/* Update dual solution with direction vector */`

$ds \leftarrow A' * A * dirx - u$

`/* Compute step size candidate vectors */`

$\delta_1 \leftarrow (Weight - ps)./ds$

$\delta_2 \leftarrow (-Weight - ps)./ds$

$\delta_3 \leftarrow -xs(gamma_x)./dirx(gamma_x)$

`/* Determine the action and step sizes */`

if $\min(\min_{pos}(\delta_1), \min_{pos}(\delta_2)) < \min_{pos}(\delta_3))$ **then**

| $flag \leftarrow 1$

end

`/* Add element to active support set when flag equal to 1, otherwise remove element */`

else

| $flag \leftarrow 0$

end

if $flag == 1$ **then**

| $\delta \leftarrow min_n(min_{pos}(\delta_1), min_pos(\delta_2))$

end

`/* n elements should be smaller than the minimum positive of` δ_3 `*/`

else

| $\delta \leftarrow min_n(min_{pos}(\delta_3))$

end

`/* n elements should be smaller than the minimum positive of` δ_1 `and` δ_2 `*/`

Algorithm 8: The step size computation

B.4 Support set update process

After appropriate step size is determined, the step ϵ is updated as $\epsilon = \epsilon + \delta$, and the largest δ value will be used if the step size is a vector. Then, solution xs will be updated as $xs = xs + \delta * dirx$.

B.5 Cholesky rank-1 update process

Input: A full rank matrix $A \in \mathbb{R}^{m \times n}$ $(m < n)$, the vector index of added element add_{index}, the previous calculated Cholesky factor R, and the direction vector $dirx$

$add_x \leftarrow A(:, add_{index})$
$diag \leftarrow add_x^T * add_x$
$AA_k \leftarrow (A(:, gamma_x)^T * A(:, gamma_x))(:, k)$
`/* k is the support set element number after 1 insertion */`
$R_k \leftarrow (R^T)^{-1} * AA_k$
`/* New vector of Cholesky factor R */`
$R_{diag} \leftarrow \sqrt{diag - R_k^T * R_k}$
`/* New diagonal element of Cholesky factor R */`
$R(:, k) \leftarrow [R_k; R_{diag}]$
$R(k, :) \leftarrow [zeros(k-1)\ R_{diag}]$
$upd \leftarrow R^{-1} * [zeros(k-1); 1]$
$dirx \leftarrow [dirx; 0] + (upd^T * u([gamma_x; add_x])) * upd$
`/* Update direction vector with updated Cholesky factor R */`
$gamma_x \leftarrow [gamma_x; add_x]$
`/* Add new element to support set */`

Algorithm 9: Cholesky rank-1 update as one element added

The Cholesky rank-1 update includes the processes for support set element insertion and deletion [Sjöstrand (2005)]. With a new element added into the support set, its corresponding vectors in original matrix A, the symmetrized matrix $A^T * A$, and the current Cholesky factor R are used to calculate the new vector elements for R. Then, the newly constructed Cholesky factor R is used to update the direction vector $dirx$. This update process involves with simple matrix-vector operations, and the detail computations can be seen in Algo. 9.

Compared to element insertion, the Cholesky rank-1 update after element deletion requires more complicated operations. After the removal of an arbitrary vector, iterative Givens rotations [Bindel et al. (2002)] are performed from this column to the end of the matrix to annihilate

Input: The previous calculated Cholesky factor R, the direction vector $dirx$, and the vector index of delete element $delete_{index}$,

$n \leftarrow size(R)$

$R(:, delete_{index}) \leftarrow []$

`/* Remove the column */`

for $k = delete_{index} \rightarrow (n-2)$ **do**

  $r \leftarrow \sqrt{R(k,k)^2 + R(k+1,k)^2}$

  $cos \leftarrow \frac{R(k,k)}{r}$

  $sin \leftarrow \frac{-R(k+1,k)}{r}$

  $R(k,k) \leftarrow r$

  `/* Zero out element by using Givens rotation */`

  for $m = k+1 \rightarrow n-1$ **do**

    $R(k,m) \leftarrow R(k,m) * cos - R(k+1,m) * sin$

    $R(k+1,m) \leftarrow R(k,m) * sin + R(k+1,m) * cos$

  end

  `/* Update the rest elements affected by Givens rotation */`

end

$R(n,:) = []$ `/* Remove last row */`

$dirx \leftarrow R^{-1} * ((R^{-1})^T) * u(gamma_x))$

`/* Update of direction with triangular matrix inverse */`

Algorithm 10: Cholesky rank-1 update as one element deleted

non-zero off-diagonals in the lower triangular part of the matrix. Then, matrix inverse is performed on updated R, whose result is used to calculate the new direction vector $dirx$ (see Algo. 10).

B.6 Stop criterion evaluation process

Input: Step parameter ϵ, the direction vector $dirx$, and the solution vectors ps, xs

if $\epsilon > 1$ **then**

  $\delta \leftarrow 1 - \epsilon$ `/* Update final step size */`

  $xs \leftarrow xs + \delta * dirx$ `/* update solution vector */`

end

Algorithm 11: Stop criterion evaluation process

The minimization process will be terminated as ϵ reaches 1. Then, the final step size and solution xs are updated accordingly as Algo. 11.

BIBLIOGRAPHY

Ahmedsaid, A., Amira, A., and Bouridane, A. (2003). Improved SVD systolic array and implementation on FPGA. *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT)*, pages 35–42.

Anderson, M. J., Ballard, G., Demmel, J., and Keutzer, K. (2011). Communication-avoiding QR Decomposition for GPUs. *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 48–58.

Ang, Z. P. and Kumar, A. (2013). Real-time and low power embedded ℓ_1-optimization solver design. *Proceedings of International Conference on Field-Programmable Technology*, pages 168–175.

Asif, M. S. and Romberg, J. (2014). Sparse recovery of streaming signals using ℓ_1-*homotopy*. *IEEE Transactions on Signal Processing*, 62(16):4209–4223.

Asif, M. S. and Romberg, J. K. (2013). Sparse recovery of streaming signals using ℓ_1-homotopy. *CoRR*, abs/1306.3331.

Aslan, S., Niu, S., and Saniie, J. (2012). FPGA implementation of fast QR Decomposition based on Givens rotation. *Proceedings of the IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 470–473.

Bai, B., Weston, J., Grangier, D., Collobert, R., Sadamasa, K., Qi, Y., Chapelle, O., and Weinberger, K. (2009). Supervised semantic indexing. *Proceedings of the ACM Conference on Information and Knowledge Management*, pages 187–196.

Bai, L., Maechler, P., Muehlberghuber, M., and Kaeslin, H. (2012). High-speed compressed sensing reconstruction on FPGA using OMP and AMP. *Proceedings of IEEE International Conference on Electronics, Circuits and Systems*, pages 53–56.

Bečka, M., Okša, G., and Vajteršic, M. (2012). *Parallel Block-Jacobi SVD Methods*. Springer London, London.

Becker, S., Bobin, J., and Cands, E. J. (2011). Nesta: A fast and accurate first-order method for sparse recovery. *SIAM Journal on Imaging Sciences*, 4(1):1–39.

Bertrand, A. and Moonen, M. (2011). Consensus-based distributed total least squares estimation in ad hoc wireless sensor networks. *IEEE Transactions on Signal Processing*, 59(5):2320 –2330.

Betkaoui, B., Thomas, D. B., and Luk, W. (2010). Comparing performance and energy efficiency of FPGAs and GPUs for high productivity computing. *Proceedings of International Conference on Field-Programmable Technology (FPT)*, pages 94–101.

Bindel, D., Demmel, J., Kahan, W., and Marques, O. (2002). On computing givens rotations reliably and efficiently. *ACM Transaction on Mathematical Software*, 28(2):206–238.

Blache, P., Rabah, H., and Amira, A. (2012). High level prototyping and FPGA implementation of the orthogonal matching pursuit algorithm. *Proceedings of International Conference on Information Science, Signal Processing and their Applications*, pages 1336–1340.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022.

Bouwmeester, H., Jacquelin, M., Langou, J., and Robert, Y. (2011). Tiled QR factorization algorithms. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.

Brent, R. P. and Luk, F. T. (1982). A systolic architecture for the Singular Value Decomposition. Technical report, Ithaca, NY, USA.

Brent, R. P., Luk, F. T., and Loan, C. V. (1985). Computation of the Singular Value Decomposition using mesh-connected processors. *Journal of VLSI Computer Systems*, pages 243–270.

Brewer, T. M. (2010). Instruction set innovations for the Convey HC-1 computer. *IEEE Micro*, 30(2):70–79.

Bruckstein, A. M., Donoho, D. L., and Elad, M. (2009). From sparse solutions of systems of equations to sparse modeling of signals and images. *Journal of SIAM Review*, 51(1):34–81.

Buttari, A., Langou, J., Kurzak, J., and Dongarra, J. (2009). A class of parallel tiled linear algebra algorithms for multicore architectures. *Journal of Parallel Computing*, 35(1):38–53.

Byna, S., Meng, J., Raghunathan, A., Chakradhar, S., and Cadambi, S. (2010). Best-effort semantic document search on GPUs. pages 86–93.

Cadambi, S., Majumdar, A., Becchi, M., Chakradhar, S., and Graf, H. P. (2010). A programmable parallel accelerator for learning and classification. pages 273–284.

Candès, E. J., Li, X., Ma, Y., and Wright, J. (2011). Robust principal component analysis? *Journal of the ACM*, 58(3):11:1–11:37.

Cavanagh, J. M., Potok, T. E., and Cui, X. (2009). Parallel Latent Semantic Analysis using a Graphics Processing Unit. *Proceedings of the Annual Conference Companion on Genetic and Evolutionary Computation Conference*, pages 2505–2510.

Chan, T. F. (1982). An improved algorithm for computing the Singular Value Decomposition. *Journal of ACM Transactions on Mathematical Software*, 8(1):72–83.

Che, S., Li, J., Sheaffer, J. W., Skadron, K., and Lach, J. (2008). Accelerating compute-intensive applications with GPUs and FPGAs. *Proceedings of the Symposium on Application Specific Processors*, pages 101–107.

Chen, S. S., Donoho, D. L., and Saunders, M. A. (2001). Atomic decomposition by basis pursuit. *Journal of SIAM Review*, 43(1):129–159.

Chen, X., Ren, L., Wang, Y., and Yang, H. (2015). GPU-accelerated sparse LU factorization for circuit simulation with performance modeling. *Journal of IEEE Transactions on Parallel and Distributed Systems*, 26(3):786–795.

Chen, X., Wang, Y., and Yang, H. (2013). Nicslu: An adaptive sparse matrix solver for parallel circuit simulation. *Journal of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(2):261–274.

Chen, Y., Wang, L., and Dong, M. (2010). Non-negative matrix factorization for semisupervised heterogeneous data coclustering. *Journal of IEEE Transactions on Knowledge and Data Engineering*, 22(10):1459–1474.

Cleveland Ashcraft, C., Grimes, R. G., Lewis, J. G., Peyton, B. W., Simon, H. D., and Bjørstad, P. E. (1987). Progress in sparse matrix methods for large linear systems on vector supercomputers. *International Journal of High Performance Computing Applications*, 1(4):10–30.

Constantine, P. G., Gleich, D. F., Hou, Y., and Templeton, J. (2014). Model reduction with mapreduce-enabled tall and skinny Singular Value Decomposition. *SIAM Journal on Scientific Computing*, 36(5).

Convey Computer (2012). Convey: Better computing for better analytics. Technical report.

Convey Computer HC-2 (2012). The Convey HC-2 computer architecture overview. White paper.

Convey Computer PDK (2012). *Convey Personality Development Kit Reference Manual.*

Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. (2001). *Introduction to Algorithms.* McGraw-Hill Higher Education, 2nd edition.

Cosoroaba, A. and Rivoallon, F. (2006). Achieving higher system performance with the Virtex-5 family of FPGAs. Technical report.

Cunningham, K., Nagvajara, P., and Johnson, J. (2011). Reconfigurable multicore architecture for power flow calculation. *Proceedings of North American Power Symposium (NAPS)*, pages 1–7.

Dai, J., Xu, W., Zhang, J., and Chang, C. (2015). Homotopy algorithm for ℓ_1-norm minimization problems. *IET Signal Processing*, 9(1):1–9.

Davis, T. A. (2004). Algorithm 832: UMFPACK V4.3—an Unsymmetric-pattern Multifrontal Method. *Journal of ACM Transaction on Mathematical Software*, 30(2):196–199.

Davis, T. A. and Hu, Y. (2011). The University of Florida sparse matrix collection. *Journal of ACM Transaction on Mathematical Software*, 38(1):1:1–1:25.

Deerwester, S., Dumais, S. T., Furnas, G. W., Landauer, T. K., and Harshman, R. (1990). Indexing by Latent Semantic Analysis. *Journal of the American society for information science*, 41(6):391–407.

Demmel, J., Gilbert, J., and Li, X. (1999). An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4):915–952.

Demmel, J. and Kahan, W. (1990). Accurate singular values of bidiagonal matrices. *SIAM Journal on Science and Statistical Computing*, 11(5):873–912.

Dongarra, J., Faverge, M., Herault, T., Langou, J., and Robert, Y. (2012). Hierarchical QR factorization algorithms for multi-core cluster systems. *Proceedings of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*.

Donoho, D. and Tsaig, Y. (2008). Fast solution of ℓ_1-norm minimization problems when the solution may be sparse. *Journal of IEEE Transactions on Information Theory*, 54(11):4789–4812.

Drmac, Z. (1997). Implementation of Jacobi Rotations for accurate singular value computation in floating point arithmetic. *SIAM Journal on Scientific Computing*, 18(4):1200–1222.

Duff, I. S. and Reid, J. K. (1983). The multifrontal solution of indefinite sparse symmetric linear. *Journal of ACM Transaction on Mathematical Software*, 9(3):302–325.

Echman, F. and Owall, V. (2005). A scalable pipelined complex valued matrix inversion architecture. *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, 5:4489–4492.

Efron, B., Hastie, T., Johnstone, I., and Tibshirani, R. (2004). Least angle regression. *Journal of Annals of Statistics*, 32:407–499.

Eick, S., Lockwood, J., Loui, R., Levine, A., Mauger, J., Weishar, D., Ratner, A., and Byrnes, J. (2006). Hardware accelerated algorithms for semantic processing of document streams. *Proceedings of IEEE Aerospace Conference*.

El-Amawy, A. and Dharmarajan, K. R. (1989). Parallel VLSI algorithm for stable inversion of dense matrices. *Journal of Computers and Digital Techniques*, 136(6):575–580.

Fang, Y., Chen, L., Wu, J., and Huang, B. (2011). Gpu implementation of orthogonal matching pursuit for compressive sensing. *Proceedings of International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1044–1047.

Figueiredo, M., Nowak, R., and Wright, S. (2007). Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems. *IEEE Journal of Selected Topics in Signal Processing*, 1(4):586–597.

Fowers, J., Brown, G., Cooke, P., and Stitt, G. (2012). A performance and energy comparison of FPGAs, GPUs, and Multicores for sliding-window applications. *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, pages 47–56.

Gabbay, J. E. and Scott, W. R. (2012). A simple method for computing discrete spectrum relaxations of body of revolution targets using Eigenvalue Decomposition. *Proceedings of IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 582 –585.

Gårtner, K. (2004). Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20:475–487.

Gee, K. R. (2003). Using Latent Semantic Indexing to filter spam. *Proceedings of the ACM Symposium on Applied Computing*, pages 460–464.

Gentleman, W. M. (1975). Error analysis of QR Decompositions by Givens transformations. *Journal of Linear Algebra and its Applications*, 10:189–197.

Georghiades, A., Belhumeur, P., and Kriegman, D. (2001). From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 23(6):643–660.

Glaskowsky, P. N. (2009). NVidia fermi: The first complete GPU computing architecture. Technical report.

Golub, G. and Kahan, W. (1965). Calculating the Singular Values and Pseudo-Inverse of a Matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224.

Golub, G. and Reinsch, C. (1970). Singular Value Decomposition and least squares solutions. *Numerische Mathematik*, 14(5):403–420.

Golub, G. and Van Loan, C. (1996). *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA.

Graf, H. P., Cadambi, S., Jakkula, V., Sankaradass, M., Cosatto, E., Chakradhar, S., and Dourdanovic, I. (2009). A massively parallel digital learning processor. *Advances in Neural Information Processing Systems 21*, pages 529–536.

Gu, M. and Eisenstat, S. C. (1995). A divide-and-conquer algorithm for the bidiagonal SVD. *SIAM Journal on Matrix Analysis and Applications*, 16(1):79–92.

Günther, F., Dudschig, C., and Kaup, B. (2014). LSAfun - an R package for computations based on latent semantic analysis. *Journal of Behavior Research Methods*, pages 1–15.

Haidar, A., Kurzak, J., and Luszczek, P. (2013). An improved parallel singular value algorithm and its implementation for multicore hardware. *Proceedings of SC13: International Confer-*

ence for High Performance Computing, Networking, Storage and Analysis (SC '13), pages 90:1–90:12.

Hamada, T., Benkrid, K., Nitadori, K., and Taiji, M. (2009). A comparative study on ASIC, FPGAs, GPUs and General Purpose Processors in the $O(N^2)$ gravitational N-body simulation. *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems*, pages 447–452.

Harteneck, M. and Stewart, R. W. (1998). Adaptive iir filtering using QR matrix Decomposition. *IEEE Transactions on Signal Processing*, 46(9):2562–2565.

Hauck, S. and DeHon, A. (2007). *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Heinecke, A., Klemm, M., and Bungartz, H. J. (2012). From GPGPU to many-core: Nvidia Fermi and Intel many integrated core architecture. *Computing in Science Engineering*, 14(2):78–83.

Herbordt, M., Gu, Y., VanCourt, T., Model, J., Sukhwani, B., and Chiu, M. (2008). Computing models for FPGA-based accelerators. *Computing in Science Engineering*, 10(6):35–45.

Hestenes, M. (1958). Inversion of matrices by biorthogonalization and related results. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):51–90.

Hofmann, T. (1999). Probabilistic Latent Semantic Indexing. *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–57.

Jeffers, J. and Reinders, J. (2013). *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Joao, Z., Mzyece, M., and Kurien, A. (2009). Matrix decomposition methods for the improvement of data mining in telecommunications. *Proceedings of IEEE Vehicular Technology Conference Fall*, pages 1–5.

Kapre, N. and DeHon, A. (2009). Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs. *Proceedings of International Conference on Field-Programmable Technology*, pages 190–198.

Keckler, S. W., Dally, W. J., Khailany, B., Garland, M., and Glasco, D. (2011). GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17.

Kerr, A., Campbell, D., and Richards, M. (2009). QR Decomposition on GPUs. *Proceedings of Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*.

Kestur, S., Davis, J. D., and Williams, O. (2010). BLAS comparison on FPGA, CPU and GPU. *Proceedings of the IEEE Annual Symposium on VLSI (ISVLSI)*, pages 288–293.

Kirk, D. B. and Hwu, W.-m. W. (2010). *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Kojima, M., Mizuno, S., and Megiddo, N. (1990). Theoretical convergence of large-step primal-dual interior point algorithms for linear programming. Technical Report RJ 7872, IBM US Research Centers (Yorktown,San Jose,Almaden, US).

Kotas, C. and Barhen, J. (2011). Singular Value Decomposition utilizing parallel algorithms on graphical processors. *Proceedings of OCEANS 2011*, pages 1–7.

Kuhn, H. W. and Tucker, A. W. (1951). Nonlinear programming. *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492.

Kuon, I., Tessier, R., and Rose, J. (2008). Fpga architecture: Survey and challenges. *Journal of Foundations and Trends in Electronic Design Automation*, 2(2):135–253.

Lahabar, S. and Narayanan, P. (2009). Singular Value Decomposition on GPU using CUDA. *Proceedings of IEEE International Symposium on Parallel Distributed Processing*, pages 1–10.

Landauer, T. K., Foltz, P. W., and Laham, D. (1998). An introduction to latent semantic analysis. *Journal of Discourse Processes*, (25):259–284.

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

Lee, K., Ho, J., and Kriegman, D. (2005). Acquiring linear subspaces for face recognition under variable lighting. *IEEE Transaction on Pattern Analysis and Machine Intelligence*, 27(5):684–698.

Leoncini, M., Manzini, G., and Margara, L. (1996). Parallel complexity of Householder QR factorization. *Proceedings of the Fourth Annual European Symposium on Algorithms (ESA)*.

Liao, R., Fernandess, Y., Tavernier, K., Taylor, G., and Irving, M. (2012). Recognition of partial discharge patterns. *Proceedings of IEEE Power and Energy Society General Meeting*, pages 1–8.

Lindholm, E. and Oberman, S. (2007). Nvidia geforce 8800 GPU. *Proceedings of Hot Chips*, 19.

Liu, Z. and McCanny, J. V. (2003). Implementation of adaptive beamforming based on QR Decomposition for CDMA. *Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2:II–609–12 vol.2.

Luebke, D. and Humphreys, G. (2007). How GPUs work. *Computer*, 40(2):96–100.

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

Majumdar, A., Cadambi, S., Chakradhar, S., and Graf, H. (2011). A parallel accelerator for semantic search. pages 122–128.

Maletic, J. and Marcus, A. (2000). Using Latent Semantic Analysis to identify similarities in source code to support program understanding. *Proceedings of IEEE International Conference on Tools with Artificial Intelligence*, pages 46–53.

Malioutov, D., Cetin, M., and Willsky, A. (2005). Homotopy continuation for sparse signal representation. 5:v/733–v/736 Vol. 5.

Martin, C. D. and Porter, M. A. (2012). The extraordinary SVD. *The American Mathimatical Monthly*, 119(10):838–852.

Meher, P., Valls, J., Juang, T.-B., Sridharan, K., and Maharatna, K. (2009). 50 years of CORDIC: algorithms, architectures, and applications. *IEEE Transactions on Circuits and Systems I*, 56(9):1893–1907.

Meng, J., Chakradhar, S., and Raghunathan, A. (2009). Best-effort parallel execution framework for recognition and mining applications. *Proceedings of IEEE International Symposium on Parallel Distributed Processing*, pages 1–12.

Microprocessor Standards Committee of the IEEE Computer Society (2008). Ieee standard for floating-point arithmetic. Technical report, Microprocessor Standards Committee of the IEEE Computer Society, New York, USA.

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

Nurvitadhi, E., Weisz, G., Wang, Y., Hurkat, S., Nguyen, M., Hoe, J. C., Martnez, J. F., and Guestrin, C. (2014). Graphgen: An fpga framework for vertex-centric graph computation. pages 25–28.

Okša, G. and Vajteršic, M. (2009). Parallel SVD computing in the latent semantic indexing applications for data retrieval. *Parallel Computing*, pages 359–395.

Pati, Y., Rezaiifar, R., and Krishnaprasad, P. (1993). Orthogonal matching pursuit: recursive function approximation with applications to wavelet decomposition. *Asilomar Conference on Signals, Systems and Computers*, 1:40–44.

Plumbley, M. D. (2006). Recovery of sparse representations by polytope faces pursuit. *Proceedings of International Conference on Independent Component Analysis and Blind Source Separation*, pages 206–213.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (2007). *Numerical Recipes 3rd ed.: The Art of Scientific Computing*. Cambridge Univ. Press, New York, NY.

Qiu, C. and Vaswani, N. (2011). ReProCS: A missing link between recursive robust PCA and recursive sparse recovery in large but correlated noise. *The Computing Research Repository*, abs/1106.3286.

Rabah, H., Amira, A., Mohanty, B., Almaadeed, S., and Meher, P. (2015). FPGA implementation of orthogonal matching pursuit for compressive sensing reconstruction. *Journal of IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 23(10):2209–2220.

Rafique, A., Kapre, N., and Constantinides, G. (2012). Enhancing performance of tall-skinny QR factorization using FPGAs. *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 443–450.

Rahman, R. (2013). *Intel Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, Berkely, CA, USA, 1st edition.

Redif, S., McWhirter, J. G., and Weiss, S. (2011). Design of FIR paraunitary filter banks for subband coding using a polynomial Eigenvalue Decomposition. *IEEE Transactions on Signal Processing*, 59(11):5253 –5264.

Řehůřek, R. and Sojka, P. (2011). Gensim - statistical semantics in python. *Proceedings of European Conference on Python in Science*.

Ren, L., Chen, X., Wang, Y., Zhang, C., and Yang, H. (2012). Sparse LU factorization for parallel circuit simulation on GPU. *Proceedings of Design Automation Conference*, pages 1125–1130.

Ritala, M., Kukli, K., Rahtu, A., Räisänen, P. I., Leskelä, M., Sajavaara, T., and Keinonen, J. (2000). Atomic layer deposition of oxide thin films with metal alkoxides as oxygen sources. *Science*, 288(5464):319–321.

Rus, V., Lintean, M. C., Banjade, R., Niraula, N. B., and Stefanescu, D. (2013). SEMILAR: the semantic similarity toolkit. *Proceedings of Annual Meeting of the Association for Computational Linguistics*, pages 163–168.

Shia, V., Yang, A., Sastry, S., Wagner, A., and Ma, Y. (2011). Fast ℓ_1-minimization and parallelization for face recognition. *Proceedings of Forty Fifth Asilomar Conference on Signals, Systems and Computers (ASILOMAR)*, pages 1199–1203.

Siddhartha and Kapre, N. (2014a). Breaking sequential dependencies in FPGA-based sparse LU factorization. *Proceedings of IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 60–63.

Siddhartha and Kapre, N. (2014b). Heterogeneous dataflow architectures for FPGA-based sparse LU factorization. *Proceedings of International Conference on Field Programmable Logic and Applications*, pages 1–4.

Sjöstrand, K. (2005). Matlab implementation of LASSO, LARS, the elastic net and SPCA. Version 2.0.

Soliman, M. (2011). Efficient implementation of QR Decomposition on Intel multi-core processors. *Prceedings of the Seventh International Computer Engineering Conference (ICENCO)*, pages 25–30.

Strumpen, V., Hoffmann, H., and Agarwal, A. (2003). A stream algorithm for the SVD. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA.

Studer, C., Blosch, P., Friedli, P., and Burg, A. (2007). Matrix decomposition architecture for mimo systems: Design and implementation trade-offs. *Proceedings of Asilomar Conference on Signals, Systems and Computers*, pages 1986–1990.

Tai, Y.-G., Psarris, K., and Lo, C.-T. D. (2011). Synthesizing tiled matrix decomposition on FPGAs. *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, pages 464–469.

Tomov, S., Nath, R., Ltaief, H., and Dongarra, J. (2010). Dense linear algebra solvers for multicore with GPU accelerators. *Proceedings of the IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8.

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

Vachranukunkiet, P. (2007). *Power Flow Computation Using Field Programmable Gate Arrays*. Drexel University.

Vanteru, B., Shaik, J., and Yeasin, M. (2008). Semantically linking and browsing PubMed abstracts with gene ontology. *Journal of BMC Genomics*, 9(Suppl 1):S10.

Villarreal, J., Park, A., Atadero, R., Najjar, W., and Edwards, G. (2010). Programming the Convey HC-1 with ROCCC 2.0. *Proceedings of Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*.

Walker, H. F. (1988). Implementation of the GMRES method using Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 9(1):152–163.

Wang, X., Jones, P., and Zambreno, J. (2014). A reconfigurable architecture for QR Decomposition using a hybrid approach. *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, pages 541–546.

Wang, X., Jones, P. H., and Zambreno, J. (2016). A configurable architecture for sparse LU Decomposition on matrices with arbitrary patterns. *SIGARCH Computer Architecture News*, 43(4):76–81.

Wang, X. and Leeser, M. (2009). A truly two-dimensional systolic array FPGA implementation of QR Decomposition. *ACM Transaction on Embedded Computing System*, 9(1):1–17.

Wang, X. and Zambreno, J. (2014a). An efficient architecture for floating-point Eigenvalue Decomposition. *Proceedings of IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 64–67.

Wang, X. and Zambreno, J. (2014b). An FPGA implementation of the Hestenes-Jacobi algorithm for Singular Value Decomposition. pages 220–227.

Wen, Z., Zhongpan, Q., and Zhijun, S. (2010). FPGA implementation of efficient fft algorithm based on complex sequence. *IEEE International Conference on Intelligent Computing and Intelligent Systems*, 2:614–617.

Wright, S., Nowak, R., and Figueiredo, M. (2009). Sparse reconstruction by separable approximation. *Journal of IEEE Transactions on Signal Processing*, 57(7):2479–2493.

Wu, G., Xie, X., Dou, Y., Sun, J., Wu, D., and Li, Y. (2012). Parallelizing sparse LU Decomposition on FPGAs. *Proceedings of International Conference on Field-Programmable Technology (FPT)*, pages 352–359.

Xilinx Inc. (2012). Logicore IP Floating-point operator data sheet. Technical report.

Xu, H., Caramanis, C., and Sanghavi, S. (2012). Robust PCA via outlier pursuit. *IEEE Transactions on Information Theory*, 58(5):3047–3064.

Xue, P., Bae, K., Kim, K., and Yang, H. (2013). Progressive equalizer matrix calculation using QR Decomposition in MIMO-OFDM systems. *Proceedings of the IEEE Consumer Communications and Networking Conference (CCNC)*, pages 593–596.

Yang, A., Ganesh, A., Sastry, S., and Ma, Y. (2010). Fast ℓ_1-minimization algorithms and an application in robust face recognition: A review. Technical report, EECS Department, University of California, Berkeley.

Yang, A. Y., Zhou, Z., Balasubramanian, A. G., Sastry, S. S., and Ma, Y. (2013). Fast ℓ_1 -minimization algorithms for robust face recognition. *IEEE Transactions on Image Processing*, 22(8):3234–3246.

Yang, J. and Zhang, Y. (2011). Alternating direction algorithms for ℓ_1-problems in compressive sensing. *SIAM Journal on Scientific Computing*, 33(1):250–278.

Yang, Y.-H. E. and Prasanna, V. K. (2010). High throughput and large capacity pipelined dynamic search tree on FPGA. pages 83–92.

Yu, D. and Wang, H. (1990). A new parallel LU Decomposition method. *Journal of IEEE Transactions on Power Systems*, 5(1):303–310.

Zhang, Y., Shalabi, Y. H., Jain, R., Nagar, K. K., and Bakos, J. D. (2009). FPGA vs. GPU for sparse matrix vector multiply. *Proceedings of International Conference on Field-Programmable Technology (FPT)*, pages 255–262.