

EFFICIENT ALGORITHMS FOR SPARSE SINGULAR VALUE DECOMPOSITION

By

SIVASANKARAN RAJAMANICKAM

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2009

© 2009 Sivasankaran Rajamanickam

To Aarthi, Akilan, and Sowmya

ACKNOWLEDGMENTS

I am fortunate to have the support of some truly wonderful people. First and foremost, I would like to thank my advisor Dr. Timothy Davis who has been an excellent mentor. I will be indebted to him forever for the trust he placed in me even before I started graduate studies and while doing this study. He has not only taught me technical nuances, but also helped me appreciate the beauty in everything from perfect software to poetry. I am grateful for his valuable advice, patience and for making me a better person.

I would like to acknowledge Dr. Gene Golub who was in my PhD committee for his kind and encouraging words. He gave the motivation to solve the sparse singular value decomposition problem. I would also like to thank Dr. Alin Dobra, Dr. William Hager, Dr. Jorg Peters, and Dr. Meera Sitharam for serving in my PhD committee.

On the personal side, I would like to thank my wife Sridevi whose sacrifices and constant support made this whole effort possible. She has always been with me whenever I needed her. I could have never completed this study without her beside me. My son Akilan and daughter Sowmya deserve a special mention for the time spent with them rejuvenated me while working on this dissertation.

I am grateful for the love and support of my parents G. Rajamanickam and R. Anusuya. They provided me with the best resources they knew and gave me the foundation that made graduate studies possible. My brother Sarangan deserves a special thanks for acting as a sounding board of ideas both in the personal and professional side. I would like to thank my friends and colleagues Srijith and Venkat for all their help during the graduate studies. I would also like to acknowledge my friends Balraj, Baranidharan, Dhamodaran, Karthikeya Arasan, Ramasubbu, Seenivasan, Senthilkumar and Simpson Fernando for cheering me on for the past fifteen years.

TABLE OF CONTENTS

	<u>page</u>
ACKNOWLEDGMENTS	4
LIST OF TABLES	7
LIST OF FIGURES	8
ABSTRACT	10
CHAPTER	
1 INTRODUCTION	12
1.1 Singular Value Decomposition	12
1.1.1 Computing the Singular Value Decomposition	13
1.1.2 Orthogonal Transformations	13
1.2 Singular Value Decomposition of Sparse and Band Matrices	14
2 LITERATURE REVIEW	17
2.1 Givens Rotations and Band Reduction	17
2.2 Methods to Compute the Singular Value Decomposition	20
3 BAND BIDIAGONALIZATION USING GIVENS ROTATIONS	22
3.1 Overview	22
3.2 Blocking for Band Reduction	23
3.3 Pipelining the Givens Rotations	25
3.3.1 Seed Block	26
3.3.2 Column Block	27
3.3.3 Diagonal Block	29
3.3.4 Row Block	30
3.3.5 Fill Block	31
3.3.6 Floating Point Operations for Band Reduction	32
3.4 Accumulating the Plane Rotations	34
3.4.1 Structure of U	35
3.4.2 Floating Point Operations for Update	42
3.5 Performance Results	43
3.6 Summary	47
4 PIRO_BAND: PIPELINED PLANE ROTATIONS FOR BAND REDUCTION	49
4.1 Overview	49
4.2 Features	49
4.2.1 PIRO_BAND Interface	50
4.2.2 LAPACK Style Interface	52
4.2.3 MATLAB Interface	52

4.3	Performance Results	53
4.3.1	Choosing a Block Size	53
4.3.2	PIRO_BAND svd and qr Performance	56
5	SPARSE R-BIDIAGONALIZATION USING GIVENS ROTATIONS	58
5.1	Introduction	58
5.2	Theory for Sparse Bidiagonalization	58
5.2.1	Profile and Corner Definitions	58
5.2.2	Mountain, Slope and Sink	62
5.2.3	Properties of Skyline and Mountainview Profiles	64
5.2.4	Properties for Blocking the Bidiagonal Reduction	70
5.3	Bidiagonalization Algorithms	73
5.3.1	Blocking in Mountainview Profile	74
5.3.2	Blocking in Skyline Profile	75
5.4	Symbolic Factorization	80
5.5	PIRO_SKY: Pipelined Plane Rotations for Skyline Reduction	84
5.6	Performance Results	87
6	CONCLUSION	90
	REFERENCES	92
	BIOGRAPHICAL SKETCH	97

LIST OF TABLES

<u>Table</u>	<u>page</u>
4-1 MATLAB interface to PIRO_BAND	52
4-2 PIRO_BAND svd vs MATLAB dense svd	56
4-3 PIRO_BAND qr vs MATLAB dense and sparse qr	57
5-1 MATLAB interface to PIRO_SKY	86
5-2 C interface to PIRO_SKY	86

LIST OF FIGURES

<u>Figure</u>	<u>page</u>
2-1 Schwarz's methods for band reduction	18
3-1 First two iterations of blocked band reduction	23
3-2 Finding and applying column rotations in the S-block	27
3-3 Applying column rotations to the C-block.	27
3-4 Applying column rotations and finding row rotations in the D-block	28
3-5 Applying row rotations to one column in the R-block.	29
3-6 Applying row rotations and finding column rotations in the F-block	31
3-7 Update of U for symmetric A with $r = 1$	34
3-8 Update of U for symmetric A with $r = 3$	35
3-9 Update of U for unsymmetric A with $r = 1$	36
3-10 Update of U for unsymmetric A with $r = 2$	39
3-11 Performance of the blocked reduction algorithm with default block sizes	44
3-12 Performance of <code>piro_band_reduce</code> vs SBR and LAPACK	44
3-13 Performance of <code>piro_band_reduce</code> vs SBR and LAPACK with update	45
3-14 Performance of <code>piro_band_reduce</code> vs DGBBRD of LAPACK	45
4-1 Performance of <code>piro_band_reduce</code> for different block sizes	54
4-2 Performance of <code>piro_band_reduce</code> for different block sizes	55
5-1 A sparse R , skyline profile of R and mountainview profile of R	59
5-2 Corner definition in the profile of R for corner columns j and n	60
5-3 Corner columns j and n after column rotations	60
5-4 Corners for the sparse R	61
5-5 Types of columns in the mountainview profile	63
5-6 End column lemma	64
5-7 Top row lemma	65
5-8 Rubble row lemma	66

5-9	Different conditions to establish progress in reduction	68
5-10	Potential columns for blocking in the mountainview profile	70
5-11	Problem with simple blocking in the skyline profile	76
5-12	First three blocks and ten corners of the skyline reduction algorithm	79
5-13	Classification of columns in the skyline profile	80
5-14	Symbolic factorization of the skyline profile	81
5-15	Performance of PIRO_SKY vs MATLAB svds	87
5-16	Performance of PIRO_SKY vs MATLAB svds	88

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

EFFICIENT ALGORITHMS FOR SPARSE SINGULAR VALUE DECOMPOSITION

By

Sivasankaran Rajamanickam

December 2009

Chair: Timothy A. Davis
Major: Computer Engineering

Singular value decomposition is a problem that is used in a wide variety of applications like latent semantic indexing, collaborative filtering and gene expression analysis. In this study, we consider the singular value decomposition problem for band and sparse matrices. Linear algebraic algorithms for modern computer architectures are designed to extract maximum performance by exploiting modern memory hierarchies, even though this can sometimes lead to algorithms with higher memory requirements and more floating point operations. We propose blocked algorithms for sparse and band bidiagonal reduction.

The blocked algorithms are designed to exploit the memory hierarchy, but they perform nearly the same number of floating point operations as the non-blocked algorithms. We introduce efficient blocked band reduction algorithms that utilize the cache correctly and perform better than competing methods in terms of the number of floating point operations and the amount of required workspace. Our band reduction methods are several times faster than existing methods.

The theory and algorithms for sparse singular value decomposition, especially algorithms for reducing a sparse upper triangular matrix to a bidiagonal matrix are proposed here. The bidiagonal reduction algorithms use a dynamic blocking method to reduce more than one entry at a time. They limit the sub-diagonal fill to one scalar by pipelining the blocked plane rotations. A symbolic factorization algorithm for computing

the time and memory requirements for the bidiagonal reduction of a sparse matrix helps the numerical reduction step.

Our sparse singular value decomposition algorithm computes all the singular values at the same amount of time it takes to compute a few singular values using existing methods. It performs much faster than existing methods when more singular values are required. The features of the software implementing the band and sparse bidiagonal reduction algorithms are also presented.

CHAPTER 1 INTRODUCTION

The Singular Value Decomposition (SVD) is called the “swiss army knife” of matrix decompositions. Applications of the SVD tend to be varied from Latent Semantic Indexing (LSI) to collaborative filtering. The SVD is also one of the most expensive decompositions, both in terms of computation and memory usage. We present efficient algorithms for finding the singular value decomposition for two special classes of matrices: sparse matrices and band matrices.

This chapter introduces the singular value decomposition in Section 1.1. Section 1.1.1 describes how to compute the SVD and the orthogonal transformations used in computing the SVD. Section 1.2 considers the special problems with band and sparse SVD.

1.1 Singular Value Decomposition

The Singular value decomposition of a matrix $A \in \mathcal{R}^{m \times n}$ is defined as

$$A = U\Sigma V^T$$

where $U \in \mathcal{R}^{m \times m}$ and $V \in \mathcal{R}^{n \times n}$ and U and V are orthogonal matrices. $\Sigma \in \mathcal{R}^{m \times n}$ is a diagonal matrix with real positive entries. This formulation of the SVD is generally referred to as the *full* SVD [Golub and van Loan 1996; Trefethen and Bau III 1997].

The more commonly used form of SVD is $A \in \mathcal{R}^{m \times n}$ and $m \geq n$ is defined as

$$A = \hat{U}\hat{\Sigma}V^T$$

where $\hat{U} \in \mathcal{R}^{m \times n}$ and $V \in \mathcal{R}^{n \times n}$ and \hat{U} and V have orthonormal columns. $\hat{\Sigma} \in \mathcal{R}^{n \times n}$ is a diagonal matrix with real positive entries. This is generally referred to as the *thin* SVD or the *economy* SVD [Trefethen and Bau III 1997]. The discussion in this dissertation applies to both the *thin* and *full* SVD. We will use the term SVD to refer to both of them and differentiate wherever required.

1.1.1 Computing the Singular Value Decomposition

The most general method to compute the SVD of A will be as follows [Golub and van Loan 1996; Trefethen and Bau III 1997]

1. Compute the QR factorization of the matrix A . ($A = QR$)
2. Reduce the upper triangular matrix R to a bidiagonal matrix B using orthogonal transformations. ($R = U_1BV_1$)
3. Reduce the bidiagonal matrix B to a diagonal matrix Σ using an iterative method. ($B = U_2\Sigma V_2$)

The computation of the SVD can then be represented as

$$\begin{aligned} A &= QR \\ &= Q(U_1BV_1) \\ &= Q(U_1(U_2\Sigma V_2)V_1) \\ &= (QU_1U_2)\Sigma(V_2V_1) \\ &= U\Sigma V \end{aligned}$$

There are several variations to these steps. The QR factorization can be skipped entirely if A is upper triangular. If A is sparse we can apply a fill reducing ordering [Davis et al. 2004a,b] before the QR factorization to minimize the number of non-zeros in R or use a profile reduction ordering of R [Cuthill and McKee 1969; Hager 2002; Gibbs et al. 1976; Reid and Scott 2002, 1998]. Some of the different choices for computing the SVD are discussed in Section 2.2.

1.1.2 Orthogonal Transformations

When reducing the upper triangular R to a bidiagonal matrix in step 2 of Section 1.1.1 we use orthogonal transformations such as the Givens rotations or Householder transformations. These transformations preserve the singular values of R . Householder-reflection based bidiagonalization was first suggested in [Golub and Kahan 1965]. They reduce the

sub-diagonal part of an entire column in a dense matrix to zero with one Householder transformation.

In the case of band matrices both Givens rotations [Schwarz 1968; Rutishauser 1963; Kaufman 2000] and Householder transformations [Bischof et al. 2000b; Murata and Horikoshi 1975] have been used in the past to reduce it to a bidiagonal matrix. While Householder transformations operate on entire columns at a time we would like to zero entries selectively in a sparse or band bidiagonalization step, so we will use Givens rotations in both the cases.

1.2 Singular Value Decomposition of Sparse and Band Matrices

The SVD of sparse and band matrices poses various challenges. Some of these problems are also faced by many other numerical algorithms and some problems are unique to this problem. We discuss these issues here.

Data structure. We assume that the band matrices are stored in a packed column band format i.e if $A \in \mathcal{R}^{m \times n}$ and the lower bandwidth is l and the upper bandwidth is u then we store A in a $(l + u + 1) \times n$ matrix B such that $A(i, j) = B(i - j + u + 1, j)$ [Golub and van Loan 1996]. Besides the advantage of reduced space, this column based data structure, enables the use of column based algorithms which can then be adapted to the sparse bidiagonal reduction. Sparse matrices are usually stored in a compressed column format with only the non-zeroes stored. However, we need more space than that for sparse bidiagonal reduction algorithm. We present two profile data structures for the storing a sparse matrix in order to compute the SVD efficiently.

Fill-in. Almost all sparse matrix algorithms have some ways to handle fill-in: trying to reduce the fill-in by reordering the matrix or doing a symbolic factorization to find the exact fill-in etc. The former ensures that the amount of work and memory usage is reduced. The later ensures a static data structure which leads to predictable memory accesses in turn improving the performance. We apply our Givens rotations in a pipelined scheme and avoid any fill-in in the band reduction algorithms. See Chapter 3 for more

details. We cannot avoid the fill-in in the sparse case, but we try to minimize the fill-in by choosing to reduce the columns that generate the least amount of fill first. See Chapter 5 for more details. It is better to avoid the Householder transformations in the sparse case because of the catastrophic fill they might generate.

FLOPS. Floating point operations or FLOPS measures the amount of work performed by a numerical algorithm. Generally, an increase in the fill-in leads to more floating point operations. Our band reduction algorithms do no more work than a simple non-blocked band reduction algorithm due to fill. However, they do more floating point operations than the non-blocked algorithms due to the order in we choose to do the reduction. The increase in the floating point operations is due to blocking, but the advantage of blocking overwhelms the small increase in FLOPS. We also show how to accumulate the left and right singular vectors by operating on just the non-zeroes thereby reducing the number of floating point operations required.

The sparse case, uses a dynamic blocking scheme for finding a block of plane rotations that can be applied together. The algorithm for sparse bidiagonal reduction tries to do the least amount of work, but there is no guarantee for doing the least amount of FLOPS for sparse matrices. However, given a band matrix the sparse algorithm will do the same amount of FLOPS as the blocked band reduction algorithm.

Memory access. Performance of any algorithm gets impacted by the number of times they access the memory and the order in which they access the memory. The development of the Basic Linear Algebra Subroutines (BLAS) [Blackford et al. 2002; Dongarra et al. 1990] and algorithms that use the BLAS mainly focus on this aspect. Unfortunately, generating Givens rotations and applying them are BLAS-1 style operations leading to poor performance. Blocked algorithms tend to work around this problem. We present blocked algorithms for both the sparse and band bidiagonalization cases. The symbolic factorization algorithm for bidiagonal reduction of sparse matrices helps us allocate a static data structure for the numerical reduction step.

Software. Availability of usable software for these linear algebraic algorithms play a crucial role in getting the algorithms adapted. We have developed software for band reduction and for computing the SVD of a sparse matrix.

In short, we have developed algorithms for blocked bidiagonal reduction of band matrices and sparse matrices that take into consideration memory usage, operation count, caching, symbolic factorization and data structure issues. We also present robust software implementing these algorithms.

Modified versions of Chapters 3 and 5, which discuss blocked algorithms for band reduction and sparse SVD respectively, will be submitted to ACM Transactions on Mathematical Software as two theory papers [Rajamanickam and Davis 2009a]. The software for band reduction (discussed in Chapter 4 and in [Rajamanickam and Davis 2009b]) and the software for sparse bidiagonal reduction will be submitted to ACM Transactions on Mathematical Software as Collected Algorithms of the ACM.

CHAPTER 2 LITERATURE REVIEW

We summarize past research related to singular value decomposition, band reduction, sparse singular value decomposition in this chapter. The problem of band reduction, especially with respect to the symmetric reduction to tridiagonal matrix case, has been studied for nearly 40 years. Section 2.1 presents the past work in band reduction. The singular value decomposition problem for dense matrices has been studied for several years and various iterative methods are known for finding the sparse singular value decomposition too. These are described in Section 2.2.

2.1 Givens Rotations and Band Reduction

Givens rotations (or plane rotations as it is called now) and Jacobi rotations [Schwarz 1968; Rutishauser 1963] are used in the symmetric eigenvalue problem of band matrices, especially to reduce the matrix to the bidiagonal form. The Givens rotations themselves, in their present unitary form, are defined in [Givens 1958]. It takes a square root and 5 floating point operations (flops) to compute the Givens rotations and two values to save them. Stewart [1976] showed how we can save just one value, the angle of the rotation, and save some space. The computation of the square root is one of the expensive operations in modern computers. Square root free Givens rotations are discussed in [Hammarling 1974; Gentleman 1973; Anda and Park 1994]. Bindel et al. [2002] showed that it was possible to accurately find the Givens rotation without overflow or underflow.

Rutishauser [1963] gave the first algorithm for band reduction using a pentadiagonal matrix. This algorithm removes the fill caused by a plane rotation [Givens 1958] immediately with another rotation. Schwarz generalized this method for tridiagonalizing a symmetric band matrix [Schwarz 1963, 1968]. Both algorithms use plane rotations (or Jacobi rotations) to zero one non-zero entry of the band resulting in a scalar fill. The algorithms then use new plane rotations to reduce the fill, creating new fill further down

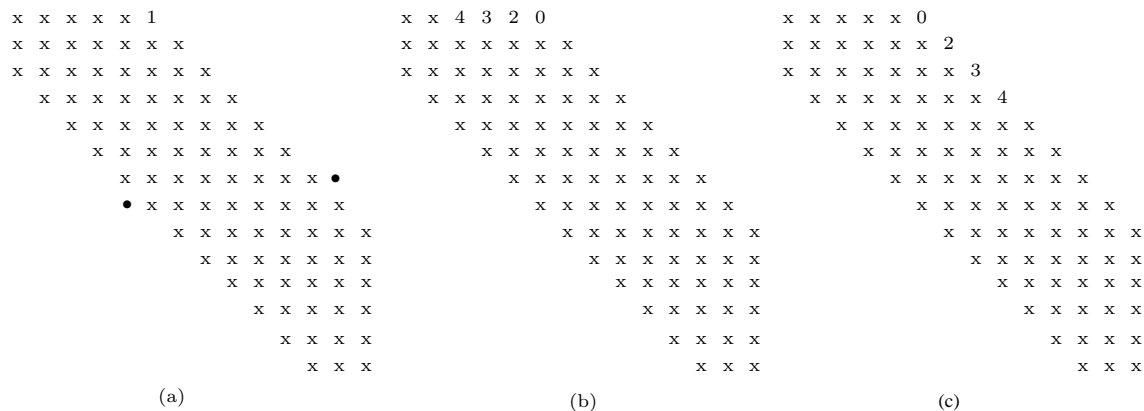


Figure 2-1. Schwarz's methods for band reduction

the matrix, until there is no fill. This process of moving the fill down the matrix is called chasing the fill.

We give a brief idea of Schwarz's generalizations [Schwarz 1963, 1968] of Rutishauser's method [Rutishauser 1963] as our blocked method for band reduction uses a mixed approach of these two methods. We refer to a band matrix with l subdiagonals and u super diagonals as an (l, u) -band matrix. A step in Schwarz's algorithms is reducing an entry in the band and chasing the resultant fill down the matrix. Figure 2-1(a) shows the first step in the reduction of a 14-by-14 $(2, 5)$ -band matrix when using Schwarz's methods. The first plane rotation tries to reduce the entry marked with 1. This causes a fill in the third subdiagonal (marked as ●). The methods rotate rows 7 and 8 and columns 12 and 13 to complete the chase. However, Schwarz's methods differ in whether it reduces an entire row to bidiagonal first ([Schwarz 1968]) or it reduces one diagonal at a time ([Schwarz 1963]) in the subsequent steps. We will call the former the row reduction method and the latter the diagonal reduction method. Figure 2-1(b)-(c) shows the difference between the two methods. In steps 2...4, the row reduction method will reduce the entries marked 2...4 in Figure 2-1(b). In steps 2...4, the diagonal reduction method will reduce the entries marked 2...4 in Figure 2-1(c).

Given a symmetric (b, b) -band matrix A of the order n , the two algorithms of Schwarz are shown in Algorithms 2.1 and 2.2. Schwarz's band reduction algorithms are not

blocked algorithms. [Rutishauser \[1963\]](#) modified the Givens method [[Givens 1958](#)] for reducing a full symmetric matrix to tridiagonal form to use blocking. The structure of the block, a parallelogram, will be the same for the modified Givens method and our algorithm, but we will allow the number of columns and number of rows in the block to be different. Furthermore, [[Rutishauser 1963](#)] led to a bulge-like fill which is expensive to chase, whereas our method results in less intermediate fill which takes much less work to chase.

Algorithm 2.1 Schwarz’s row reduction algorithm for symmetric band reduction

```

for  $i$  such that  $1 \leq i \leq n - 2$  do
  for  $j = \text{MIN}(i + b, n) - 1 : -1 : i + 1$  do
    Rotate columns  $j$  and  $j + 1$  to reduce  $A[i, j + 1]$ .
    Chase the fill.
  end for
end for

```

Algorithm 2.2 Schwarz’s diagonal reduction algorithm for symmetric band reduction

```

for  $j = b - 1 : -1 : 1$  do
  for  $i = 1 : n - (j + 1)$  do
    Rotate columns  $i + j$  and  $i + j + 1$  to reduce  $A[i, i + j + 1]$ .
    Chase the fill.
  end for
end for

```

[Kaufman \[1984\]](#) suggested an approach to reduce and chase more than one entry at a time using vector operations. The algorithm uses plane rotations and reduces one diagonal at a time. This is the method in the LAPACK [[Anderson et al. 1999](#)] library for bidiagonal and tridiagonal reduction (xGBBRD and xSBTRD routines). The latest symmetric reduction routines in LAPACK use a revised algorithm by Kaufman [[Kaufman 2000](#)]. Though Kaufman’s algorithms reduce more than one entry at a time it is not a blocked algorithm as each of the entries has to be separated by a distance. [Rutishauser \[1963\]](#) also suggested using Householder transformations for band reduction and chasing the triangular fill. [Murata and Horikoshi \[1975\]](#) also used Householder transformations to introduce the zeroes in the band but chose to chase only part of the fill. This idea

reduced the flops required, but increased the workspace. Recently, [Lang \[1993\]](#) used this idea for parallel band reduction. The idea is also used in the SBR tool box [[Bischof et al. 2000b](#)]. However, they use QR factorization of the subdiagonals and store it as WY transformations [[Van Loan and Bischof 1987](#)] so that they can take advantage of level 3 style BLAS operations. The SBR framework can choose to optimize for floating point operations, available work space and using the BLAS.

In order to compute the singular vectors we can start with an identity matrix and apply the plane rotations to it. There are two types of methods to do this: we apply the plane rotations when they are applied to the original band matrix (forward accumulation) or we save all the rotations and apply it to the identity matrices later (backward accumulation)[[Smith et al. 1976](#)]. Even though simple forward accumulation requires more floating point operations than backward accumulation, [Kaufman \[1984\]](#) showed that we can do forward accumulation efficiently by exploiting the non zero pattern when applying the rotations to the identity. Kaufman’s accumulation algorithm will do the same number of floating point operations as backward accumulation. The non zero pattern of U or V will be dependent upon the order in which we reduce the entries in the original band matrix. The accumulation algorithm exploits the non zero pattern of U or V while using our blocked reduction.

2.2 Methods to Compute the Singular Value Decomposition

[Golub and Kahan \[1965\]](#) introduced the two step approach to compute the singular value decomposition. In order to compute the SVD of A they reduce A to a bidiagonal matrix and then reduce the bidiagonal matrix to a diagonal matrix (the singular values). [Golub and Reinsch \[1970\]](#) introduced the QR algorithm to compute the second step. [Demmel and Kahan \[1990\]](#) showed that the singular values of the bidiagonal matrix can be computed accurately. The two step method was modified to a three step method described in Section 1.1.1 by Chan [[Chan 1982](#)]. The bidiagonalization was preceded by a QR factorization step followed by the bidiagonalization of R . This method leads to less work

whenever $m \geq 5n/3$. [Trefethen and Bau III 1997] suggest a dynamic approach where one starts with bidiagonalization first and switches to QR factorization when appropriate. We will use the three step approach for computing the SVD.

Sparse SVD. Iterative methods like the Lanczos or subspace iteration based algorithms in SVDPACK [Berry 1992] and PROPACK [Larsen 1998] are available for finding the sparse SVD. The main advantage of these iterative methods is their speed when only a few singular values are required. However, we have to guess the required number of singular values when we use the iterative algorithms. There is no known direct method to compute all the singular values of a sparse matrix.

Other methods and applications. The Singular value decomposition is one of the factorizations that has its uses in a diverse group of applications. Latent Semantic Indexing based applications use the SVD of large sparse matrices [Berry et al. 1994; Deerwester et al. 1990]. The SVD is also used in collaborative filtering [Pryor 1998; Paterek 2007; Goldberg et al. 2001; Billsus and Pazzani 1998] and recommender systems [Sarwar et al. 2002; Koren 2008]. Sparse graph mining applications that need to find patterns can do so efficiently with the SVD. Compact Matrix Decomposition (CMD) [Sun et al. 2008] or the approximate versions [Drineas et al. 2004] do better than SVD now. Other applications include gene expression analysis [Wall et al. 2003], finding the pseudo inverse of a matrix [Golub and Kahan 1965] and solving the linear least squares problem [Golub and Reinsch 1970]. There are other alternative methods to compute the bidiagonal reduction. One sided bidiagonal reductions are discussed in [Ralha 2003; Barlow et al. 2005; Bosner and Barlow 2007] and the semidiscrete decomposition (SDD) [Kolda and O'leary 1998] was proposed as an alternative to SVD itself.

CHAPTER 3 BAND BIDIAGONALIZATION USING GIVENS ROTATIONS

3.1 Overview

Eigenvalue computations of symmetric band matrices depend on a reduction to tridiagonal form [Golub and van Loan 1996]. Given a symmetric band matrix A of size n -by- n with l subdiagonals and super diagonals, the tridiagonal reduction can be written as

$$U^T A U = T \tag{3-1}$$

where T is a tridiagonal matrix and U is orthogonal. Singular value decomposition of an m -by- n unsymmetric band matrix A with l subdiagonals and u super diagonals depends on a reduction to bidiagonal form. We can write the bidiagonal reduction of A as

$$U^T A V = B \tag{3-2}$$

where B is a bidiagonal matrix and U and V are orthogonal.

Some of the limitations of existing band reduction methods [Bischof et al. 2000a; Kaufman 1984] can be summarized as follows:

- they lead to poor cache access and they use expensive non-stride-one access row operations during the chase (or)
- they use more memory and/or more work to take advantage of cache friendly BLAS3 style algorithms.

In this chapter, we describe an algorithm that uses plane rotations, blocks the rotations for efficient use of cache, does not generate more fill than non blocking algorithms, performs only slightly more work than the scalar versions, and avoids non-stride-one access as much as possible when applying the blocked rotations.

Section 3.2 introduces the blocking idea. Blocking combined with pipelined rotations helps us achieve minimal fill, as discussed in Section 3.3. We introduce the method to accumulate the plane rotations efficiently in Section 3.4. The performance results for the new algorithms are in Section 3.5.

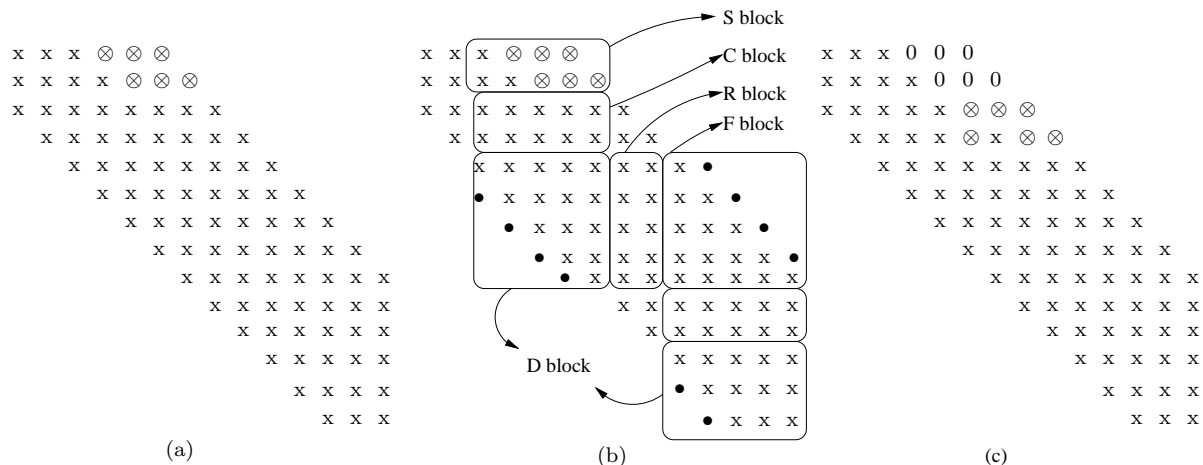


Figure 3-1. First two iterations of blocked band reduction.

We refer to a plane rotation between two columns (rows) as column (row) rotations. We use an unsymmetric band matrix as an example even though the original algorithms are for the symmetric band case as our algorithm applies to both symmetric and unsymmetric matrices. The term *k-lower Hessenberg matrix* refers to a matrix with k super diagonals and the lower triangular part.

3.2 Blocking for Band Reduction

Our blocking scheme combines Schwarz's row reduction method [Schwarz 1968] and Rutishauser's modified Givens method [Rutishauser 1963]. A *step* in our blocked reduction algorithm reduces an r -by- c block of entries and chases the resultant fill. At each step, we choose an r -by- c parallelogram as our block such that its reduction and the chase avoids the bulge-like fill when combined with our pipelining scheme (described in Section 3.3). The first c entries reduced by the row reduction method and the first r entries reduced by the diagonal reduction method are the outermost entries of the parallelogram for the first step in the blocked reduction. Figure 3-1(a) shows the selected block for the first step when the block is of size 2-by-3.

At each step, once we select the block, we partition the entries in the band that change in the current step into blocks. The blocks can be one of five types:

S-block The S (*seed*) block consists of the r -by- c parallelogram to be reduced and the entries in the same r rows as the parallelogram to which the rotations are applied.

The S-block in the upper (lower) triangular part is reduced with column (row) rotations.

C-block The C (*column rotation*) block has the entries that are modified only by column rotations in the current step.

D-block The D (*diagonal*) block consists of the entries that are modified by both column and row rotations (in that order) in the current step. Applying column rotations in this block causes fill which are chased by new row rotations. This is at most a $(c + r)$ -by- $(c + r)$ block.

R-block The R (*row rotation*) block has the entries that are modified only by row rotations in the current step.

F-block The F (*fill*) block consists of the entries that are modified by both row and column rotations (in that order) in the current step. Applying row rotations in this block causes fill which is chased by new column rotations. This is at most a $(c + r)$ -by- $(c + r)$ block.

Generally the partitioning results in one seed block and potentially more than one of the other blocks. The blocks other than the seed block repeat in the same order. The order of the blocks is C, D, R and F-block when the seed block is in the upper triangular part. Figure 3-1(b) shows all the different blocks for the first step. The figure shows more than one fill, but no more than one fill is present at the same time. The fill is limited to a single scalar. The pipelining (explained in Section 3.3) helps us restrict the fill. As a result, our method uses very little workspace other than the matrix being reduced and an r -by- c data structure to hold the coefficients for the pending rotations being applied in a single sweep.

Once we complete the chase in all the blocks, subsequent steps reduce the next r -by- c entries in the same set of r rows as long as there are more than r diagonals left in them.

This results in the two pass algorithm where we reduce the bandwidth to r first and then reduce the thin band to a bidiagonal matrix. Figure 3-1(c) highlights the entries for the second step with \otimes . The two pass band reduction is given in Algorithm 3.1. The while loop in Line 1 reduces the bandwidth to r and is taken at most twice.

Algorithm 3.1 Two pass algorithm for band reduction

```

1: while lowerbandwidth > 0 or upperbandwidth > 1 do
2:   for each set of  $r$  rows/columns do
3:     reduce the band in lower triangular part
4:     reduce the band in upper triangular part
5:   end for
6:   set  $c$  to  $r - 1$ 
7:   set  $r$  to 1.
8: end while

```

There are two exceptions with respect to the the number of diagonals left. The number of diagonals left in the lower triangular part is $r - 1$ instead of r , when r is one (to avoid the second iteration of the while loop in the above algorithm) and when the input matrix is an unsymmetric $(l, 1)$ -band matrix (for more predictable accumulation of the plane rotations). See Section 3.4 for more details on the later.

The algorithm so far assumes the number of rows in the block for the upper triangular part and the number of columns in the block for the lower triangular part are the same. We reduce the upper and lower triangular part separately if that is not the case. Algorithm 3.2 reduces the band in the upper triangular part. The algorithm for reducing the lower triangular part is similar to the one for the upper triangular part except that the order of the blocks in the chasing is different. In the inner while loop the rotations are applied to the R, F, C and D-blocks, in that order.

3.3 Pipelining the Givens Rotations

The algorithm for the reduction of the upper triangular part in Section 3.2 relies on finding and applying the plane rotations on the blocks. Our pipelining scheme does not cause triangular fill-in and avoids non-stride-one access of the matrix. The general idea is to find and store r -by- c rotations in the workspace and apply them as efficiently

Algorithm 3.2 Algorithm for band reduction in the upper triangular part

```
1: for each set of  $c$  columns in current set of rows do
2:   Find the trapezoidal set of entries to zero in this iteration
3:   Divide the matrix into blocks
4:   Find column rotations to zero  $r$ -by- $c$  entries in the S-block and apply them in the
   S-block
5:   while column rotations do
6:     Apply column rotations to the C-block
7:     Apply column rotations to the D-block, Find row rotations to chase fill
8:     Apply row rotations to the R-block
9:     Apply row rotations to the F-block, Find column rotations to chase fill
10:    Readjust the four blocks to continue the chase.
11:   end while
12: end for
```

as possible in each of the blocks to extract maximum performance. Figures 3-2 - 3-6 in this section show the first five blocks from Figure 3-1(b) and the operations in all of them are sequentially numbered from 1..96. A solid arc between two entries represents the generation of a row (or column) rotation between adjacent rows (or columns) of the matrix to annihilate one of the two entries in all the figures in this section. A dotted arc represents the application of these rotations to other pairs of entries in the same pair of rows (or columns) of the matrix.

3.3.1 Seed Block

The r -by- c entries to be reduced in the current step are in the S-block. The number of rows in this block is r and the number of columns is at most $c + r$. We find new column rotations to reduce the entries one row at a time. The rotations generated from the entries in the same row (column) in the upper (lower) triangular part are called a wave. Given a row in the seed block, we apply rotations from previous rows before finding the new rotations to reduce the entries in the current row. This is illustrated in the Figure 3-2. We reduce the three entries (marked as \otimes) in the first row with column rotations $G_1 \dots G_3$ and save the rotations in work space. This wave of rotations is also applied to the second row as operations 4...6. We then reduce the three entries in the second row with new set of rotations $G_4 \dots G_6$ and save the rotations. The steps to generate and apply the

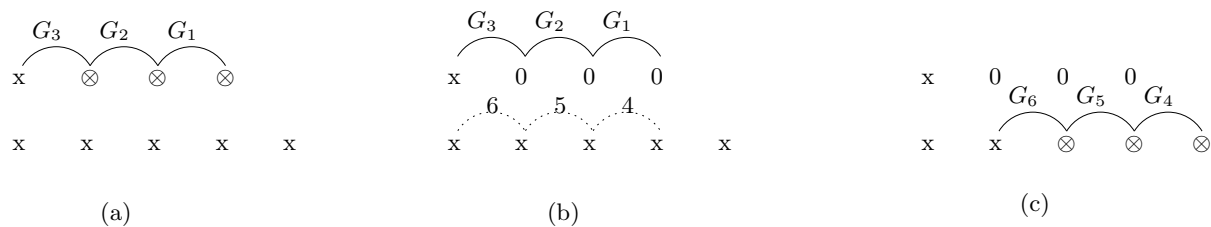


Figure 3-2. Finding and applying column rotations in the S-block



Figure 3-3. Applying column rotations to the C-block.

rotations in the S-block are shown in Algorithm 3.3. Finding and applying rotations in the S-block is the only place in the algorithm where we use explicit non-stride-one access to access the entries across the rows.

Algorithm 3.3 Generating and applying rotations in the S-block

- 1: **for** all rows in the S-block **do**
 - 2: **if** not the first row **then**
 - 3: apply column rotations from the previous rows.
 - 4: **end if**
 - 5: Find the column rotations to zero entries in current row (if there are any)
 - 6: **end for**
-

3.3.2 Column Block

We apply r waves of rotations to the C-block. Each wave consists of c column rotations. We apply each column rotation to the entire C-block one at a time (and one wave at a time), as this leads to efficient stride-one access of memory, assuming the matrix is stored in column-major order.

A column rotation operates on pairs of entries in adjacent columns (columns $i - 1$ and i , say). The next column rotation in the same wave operates on columns $i - 2$ and $i - 1$, reusing the $i - 1$ th column which presumably would remain in cache. Finally, when this

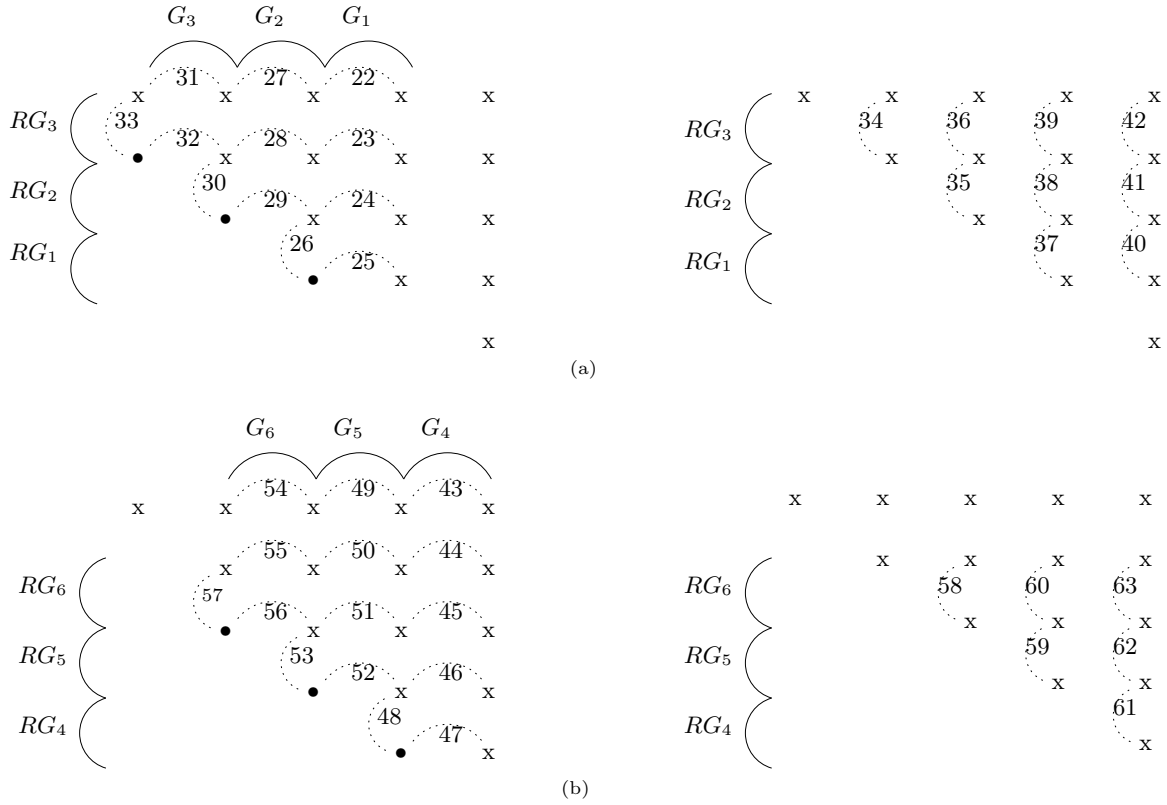


Figure 3-4. Applying column rotations and finding row rotations in the D-block.

first wave is done we repeat and apply all the remaining $r - 1$ waves to the corresponding columns of the C-block. The column rotations in any wave $j + 1$ use all but one column used by the wave j leading to possible cache reuse. Figure 3-3(a)-(b) illustrates the order in which we apply the two waves of rotations ($G_1 \dots G_3$ and $G_4 \dots G_6$) to the 2 rows in the C-block. Algorithm 3.4 describes these steps to apply the rotations in the C-block.

Algorithm 3.4 Applying rotations in the C-block

- 1: **for** each wave of column rotations **do**
 - 2: **for** each column in this wave of rotations **do**
 - 3: Apply the rotation for the current column from current wave to all the rows in the C-block.
 - 4: **end for**
 - 5: Shift column indices for next wave.
 - 6: **end for**
-

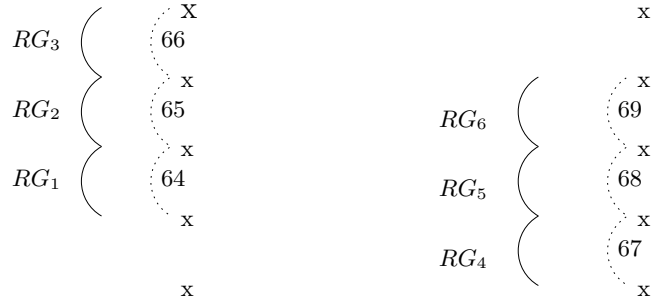


Figure 3-5. Applying row rotations to one column in the R-block.

3.3.3 Diagonal Block

We apply r waves each with c column rotations to the D-block. Each column rotation G_i (say between columns k and $k - 1$) can generate a fill in this block. In order to restrict the fill to a scalar, we reduce the fill immediately with a row rotation (RG_i) before we apply the next column rotation G_{i+1} (to columns $k - 1$ and $k - 2$). We save the row rotation after reducing the fill and continue with the next column rotation G_{i+1} , instead of applying the row rotation to the rest of the row as that would lead to non-stride-one access. When all the column rotations in the current wave are done, we apply the new wave of row rotations to each column in the D-block.

This algorithm leads to two sweeps on the entries of the D-block for each wave of column rotations: one from right to left when we apply column rotations that generate fill, find row rotations, reduce the fill and save the row rotations, and the next from left to right when we apply the pipelined row rotations to all the columns in the D-block. We repeat the two sweeps for each of the subsequent $r - 1$ waves of column rotations. Figure 3-4(a) shows the order in which we apply the column rotations $G_1 \dots G_3$, find the corresponding row rotations and apply them to our example. The operations 22...42 show these steps. Operations 22...33 show the right to left sweep and 34...42 show the left to right sweep. Figure 3-4(b) shows the order in which we apply a second wave of rotations $G_4 \dots G_6$ to our example and handle the fill generated by them (numbered 43...63). Algorithm 3.5 shows both applying and finding rotations in the D-block.

Algorithm 3.5 Applying and finding rotations in the D-block

```
1: for each wave of column rotations do
2:   for each column in this wave of rotations (right-left) do
3:     Apply the rotation for the current column from current wave to all the
       rows in the D-block, generate fill.
4:     Find row rotation to remove fill.
5:     Apply row rotation to remove fill. (not to entire row)
6:   end for
7:   for each column in this wave of rotations (left-right) and additional columns in the
       D-block to the right of these columns do
8:     Apply all pending row rotations to current column.
9:   end for
10:  Shift column indices for next wave.
11: end for
```

3.3.4 Row Block

We apply r waves of rotations to the R-block. Each wave consists of c row rotations. We could apply each row rotation to the entire R-block one at a time (and one wave at a time), but this would require an inefficient non-stride-one access of memory, assuming the matrix is stored in column-major order.

Instead, we apply all the c row rotations to just the first column of the R-block, for the first wave. A row rotation operates on adjacent entries in the column (rows $i - 1$ and i , say). The next row rotation in this wave operates on rows $i - 2$ and $i - 1$, reusing the cache for the entry from row $i - 1$. Finally, when this first wave is done we repeat and apply all the remaining $r - 1$ waves to the first column of the R-block, which presumably would remain in cache for all the r waves. This entire process is then repeated for the second and subsequent columns of the R-block. This order of operations leads to efficient cache reuse and purely stride-one access of the memory in the R-block.

Figure 3-5 shows two waves of row rotations $RG_1 \dots RG_3$ and $RG_4 \dots RG_6$ applied to one column in the R-block. The order in which we apply the rotations to the column is specified by numbers 64 \dots 69. Algorithm 3.6 gives the steps to apply the rotations in the R-block. There is no fill in the R-block. We do not show the operations 70 \dots 75 in

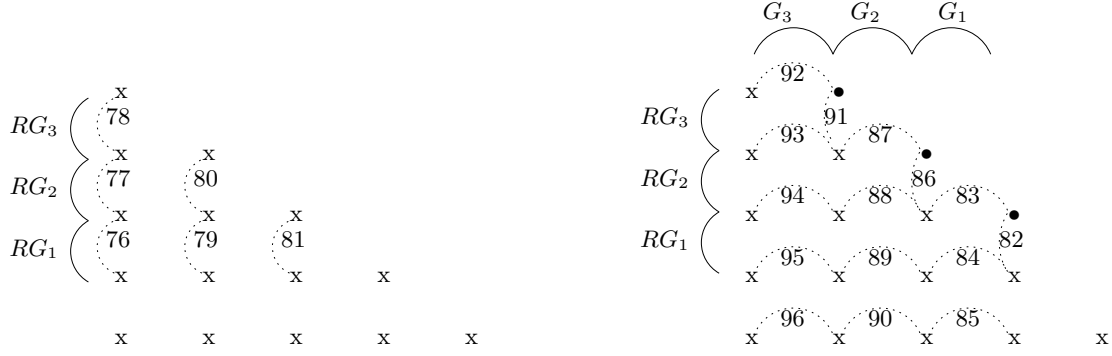


Figure 3-6. Applying row rotations and finding column rotations in the F-block

the second column of the R-block (from Figure 3-1(b)) in the Figure 3-5. But the row rotations are applied in the same pattern as in the first column.

Algorithm 3.6 Applying rotations in the R-block

- 1: **for** each column in the R-block **do**
 - 2: **for** each wave of row rotations **do**
 - 3: Apply all the rotations in current wave to appropriate rows in current column.
 - 4: Shift row indices for next wave.
 - 5: **end for**
 - 6: **end for**
-

3.3.5 Fill Block

We apply r waves each with c row rotations to the F-block. Each row rotation RG_p (between rows i and $i - 1$ say) can create a fill in this block. We could adopt the same approach as in the D-block, completing the row rotation RG_p in this block to generate the fill and using a column rotation to reduce the fill immediately before we do row rotation RG_{p+1} , to restrict the fill to a scalar, but that would require us to access all the entries in a row.

Instead, for each column in the F-block we apply all the relevant row rotations from the current wave, but just stop short of creating any fill. Note that we need not apply all c row rotations to all the columns of the F-block. When applying the row rotations to a column, successive row rotations share entries between them for maximum cache reuse as in the R-block. This constitutes one left-to-right sweep of all the entries in the F-block.

We then access the columns from right to left by applying the row rotation say RG_p (between rows i and $i - 1$ say) that generates the fill in that column of the F-block (say k), remove the fill immediately with column rotation G_p and apply the column rotation to rest of the entries in columns k and $k - 1$ in the F-block.

Figure 3-6 shows the order in which we apply the row rotations (operations 76...81), generate fill (operations 82, 86, 91), and apply the column rotations. Operations 76...81 constitute the left-to-right sweep. Operations 82...96 constitute the right-to-left sweep. We repeat this for all the $r - 1$ pending waves of rotations. Algorithm 3.7 shows the two phases in applying the rotations in the F-block.

Algorithm 3.7 Applying row rotations and finding column rotations in the F-block

```

1: for each wave of row rotations do
2:   for each column in the F-block (left-right) do
3:     Apply all rotations from current wave, except those that generate fill-in in the
       current column, to current column.
4:   end for
5:   for each column in the F-block (right-left) do
6:     Apply the pending rotation that creates fill.
7:     Find column rotation to remove fill.
8:     Apply column rotation to entire column.
9:   end for
10:  Shift row indices for next wave.
11: end for

```

To summarize, the only non-stride-one access we do in the entire algorithm is in the seed block when we try to find the rotations. We generate no more fill than the any of the scalar algorithms by pipelining our plane rotations. The amount of work we do varies based on the block size which is explained in the next subsection.

3.3.6 Floating Point Operations for Band Reduction

For the purposes of computing the exact number of floating point operations, we assume that finding a plane rotation takes six operations, as in the real case with no scaling of the input parameters (two multiplications, two divides, one addition and a

square root). We also assume that applying the plane rotations to two entries requires six floating point operations as in the real case (four multiplications and two additions).

Let us consider the case of reducing a (u, u) -symmetric band matrix of order n to tridiagonal form by operating only on the upper triangular part of the symmetric matrix. The number of floating point operations required to reduce the symmetric matrix using Schwarz's diagonal reduction method is

$$f_d = 6 \sum_{d=3}^b \sum_{i=d}^n d + 2 \left\lfloor \frac{\max(i-d, 0)}{d-1} \right\rfloor (d+1) + \text{rem} \left(\frac{\max(i-d, 0)}{d-1} \right) + 2 \quad (3-3)$$

where rem is the remainder function and $b = u + 1$. The number of floating point operations required to reduce the symmetric matrix using Schwarz's row reduction method is

$$f_r = 6 \sum_{i=1}^{n-2} \sum_{d=3}^{\min(b, n-i+1)} d + 2 \left\lfloor \frac{\max(i-d, 0)}{b-1} \right\rfloor (b+1) + \text{rem} \left(\frac{\max(i-d, 0)}{b-1} \right) + 2 \quad (3-4)$$

In order to find the number of floating point operations for our blocked reduction, when block size is r -by- c we note that we perform the same number of operations as using a two step Schwarz's row reduction method where we first reduce the matrix to r diagonals using Schwarz's row reduction method and then reduce the r diagonals using Schwarz's row reduction again. Then we can obtain the floating point operations required by the blocked reduction to reduce the matrix by using the equation 3-4 for f_r .

$$f_{b1} = \sum_{i=1}^{n-r-1} \sum_{d=r+2}^{\min(b, n-i+1)} d + 2 \left\lfloor \frac{\max(i-d, 0)}{b-1} \right\rfloor (b+1) + \text{rem} \left(\frac{\max(i-d, 0)}{b-1} \right) + 2 \quad (3-5)$$

$$f_{b2} = \sum_{i=1}^{n-2} \sum_{d=3}^{\min(r+1, n-i+1)} d + 2 \left\lfloor \frac{\max(i-d, 0)}{r} \right\rfloor (r+2) + \text{rem} \left(\frac{\max(i-d, 0)}{r} \right) + 2 \quad (3-6)$$

$$f_b = 6 (f_{b1} + f_{b2}) \quad (3-7)$$

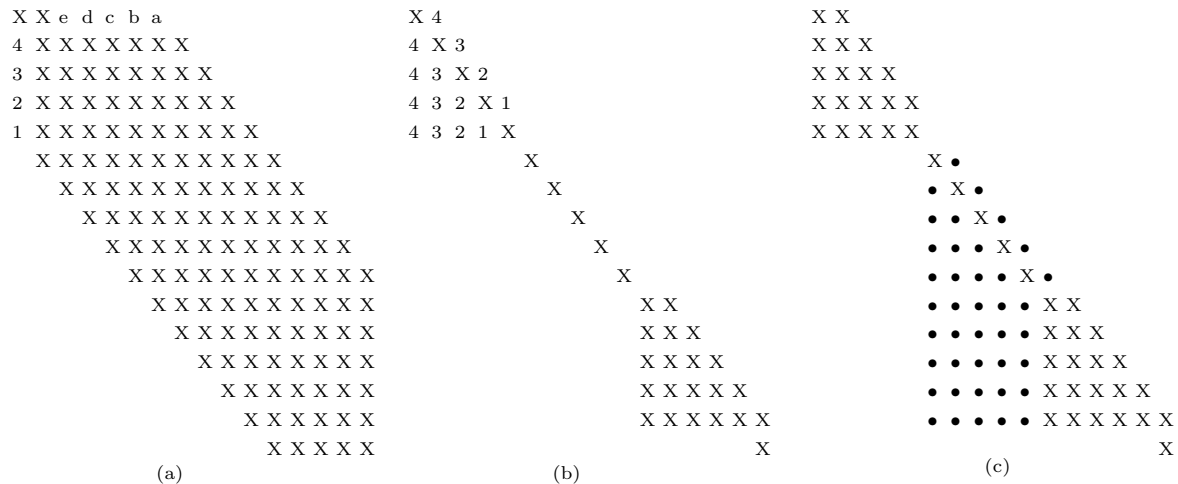


Figure 3-9. Update of U for unsymmetric A with $r = 1$

Given a band matrix A and a r -by- c block, the algorithm to find the non zero pattern in U differs based on four possible cases

- A is a symmetric or A is an unsymmetric $(0, u)$ -band matrix.
- A is an unsymmetric $(l, 1)$ -band matrix.
- A is an unsymmetric (l, u) -band matrix where $l > 0$, $u > 1$ and $r = 1$.
- A is an unsymmetric (l, u) -band matrix where $l > 0$, $u > 1$ and $r > 1$.

The structure of the algorithm is the same in all four cases. Applying the plane rotations from the first major step, i.e, reduction of the first r columns and/or rows of A , to the identity matrix U leads to a specific non zero pattern in the upper and lower triangular part of U . This non zero pattern in U then dictates the pattern of the fill in U while applying the plane rotations from the subsequent major steps.

Symmetric A : We consider a symmetric (u, u) -band matrix and an unsymmetric $(0, u)$ -band matrix as symbolically equivalent in this section as we operate only on the upper triangular part of the symmetric matrix. When A is a symmetric (u, u) -band matrix and $r = 1$, our reduction Algorithm 3.1 reduces each row to bidiagonal form before reducing any entries in the subsequent rows. When reducing the first row of A to bidiagonal form, the $u - 1$ plane rotations from the first minor step applied to columns

$2 \dots u + 1$ of U , results in a block fill in U where the block is a lower Hessenberg matrix of the the order u . The fill in U , from the next $u - 1$ plane rotations (from the next minor step of the chase in A), is in the subsequent u columns of U and has the same lower Hessenberg structure. When we apply all the plane rotations from the first major step in the reduction of A we get a block diagonal U , where each block is lower Hessenberg of order u . For example, consider reducing the first row of entries marked $1 \dots 5$ in the Figure 3-7(a). Figure 3-7(b) shows the fill pattern after applying to U the plane rotations that were generated to reduce these entries and chase the resultant fill. The fill in U , generated from the first minor step of the reduction of A is marked with the corresponding number of the entry from Figure 3-7(a) that caused the fill. For example, fill from entry marked 3 in Figure 3-7(a) is marked 3 in Figure 3-7(b). The fill in U from the subsequent minor steps are marked x .

Applying the plane rotations from reducing every other row of A (and the corresponding chase) results in a u -by- $(u - 1)$ block fill for each minor step in the lower triangular part of U (until it is full) and an increase in one super diagonal. Figure 3-7(c) shows the fill in U caused by the plane rotations from the second major step. The fill from the first minor step is marked with \bullet and the fill from subsequent minor steps are marked with \circ . Note that keeping track of the fill in the upper triangular part is relatively easy as all plane rotations that reduce entries from row k and the corresponding chase generate fill in the k -th super diagonal of U . This fill pattern in U is exactly same as the fill pattern in U if we reduced A with Schwarz's row reduction method (Algorithm 2.1), as blocked reduction follows the same order of rotations when r is one.

When $r > 1$ each block in the block diagonal U , after applying the rotations from the first major step to U , is a $(u - r, r)$ -band matrix of the order u . Each subsequent minor step generates a block fill in U that consists of a $(u - r + 1)$ -by- $(u - r)$ block and below that a block of $(0, u - r)$ -band matrix with a zero diagonal. The example given below illustrates this case clearly. As discussed in the $r = 1$ case, there is also an addition of r

super diagonals in U , for every r rows that are reduced in A . Figure 3-8 shows the fill in U for the first two major steps when $r = 3$. The conventions are same as before. In the first major step (Figure 3-8(b)) the fill from first minor step is numbered for the entries in A (Figure 3-8(a)) that caused the fill, and further fill from the other minor steps are shown as x . In the second major step (Figure 3-8(c)), the fill from two different minor steps are differentiated with \bullet and \circ . Furthermore, the two blocks of fill in the lower triangular part of U for one minor step are bounded with rectangles in Figure 3-8(c).

$(l, 1)$ -band A : Given a $(l, 1)$ -band matrix the band reduction algorithm reduces the lower triangular part to $r - 1$ diagonals to keep the pattern of the fill in U simple. When $r = 1$, applying the rotations from the first major step of the reduction leads to a block diagonal U , with each block lower Hessenberg of the order $l + 1$. Applying rotations from each subsequent major step, results in rectangular blocks of fill of the order of $(l + 1)$ -by- l in the lower triangular part of U , and one additional super diagonal in the upper triangular part of U . The basic structure of the fill is similar to the one shown in Figure 3-7(b)-(c) except the size of the blocks is different.

When $r > 1$, applying rotations from the first major step results in a block diagonal U , where each block is $(l - r + 2, r)$ -band of the order $(l + 1)$. Applying rotations from each subsequent major step results in a block of fill that consists of a $(l - r + 2)$ -by- $(l - r + 1)$ block and below that a block in the shape of $(0, l - r + 1)$ -band of the order $(r - 1)$ -by- l with a zero diagonal in the lower triangular part of U , and r additional super diagonals in the upper triangular part of U . The basic structure of the fill is similar to the one shown in Figure 3-8(b)-(c) except the size of the blocks is different.

Unsymmetric A , $r = 1$: When we consider the case of the unsymmetric (l, u) -band matrix A , to find the non zero pattern of U , we need to only look at the interaction between the plane rotations to reduce the entries in lower triangular part and the plane rotations to chase the fill from lower triangular part (even though the original entries were from the upper triangular part), as these are the only rotations applied to U . We can

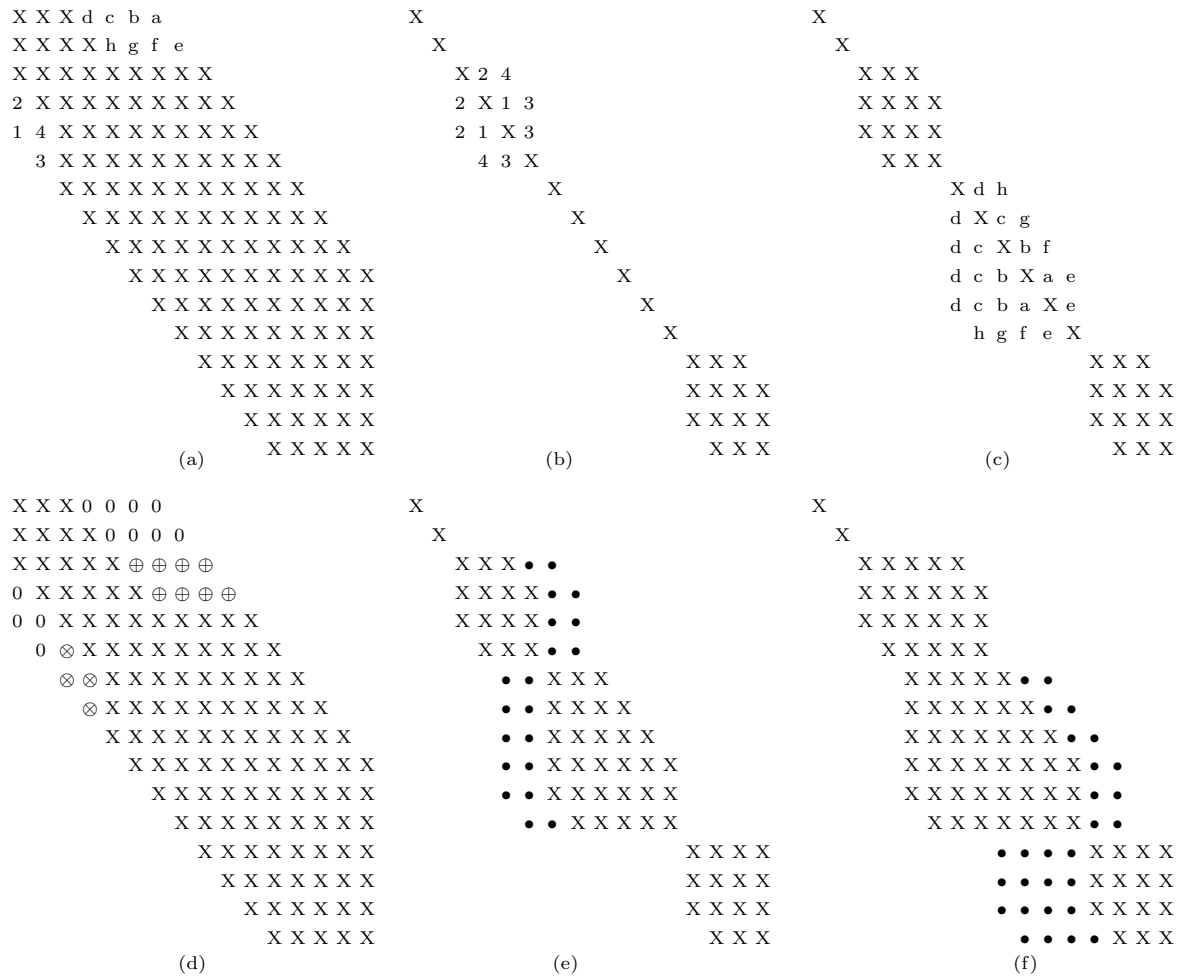


Figure 3-10. Update of U for unsymmetric A with $r = 2$

simply say that we need to look at all the row rotations to matrix A . Let us consider the case when the the number of rows in the block r is one. The reduction algorithm from Section 3.2 reduces the first column and row pair to the bidiagonal form before reducing subsequent column and rows.

After applying the rotations from the first major step of the reduction of A , U will be a block diagonal matrix where the first block is $(l + 1)$ -by- $(l + 1)$ and the rest of the blocks $(l + u)$ -by- $(l + u)$ all lower Hessenberg. Figure 3-9(a) uses an example $(6, 4)$ -band matrix A of the size 16-by-16. Figure 3-9(b) shows the fill pattern in U after reducing the first column of A and the resultant chase. The fill from the first minor step is numbered

for the corresponding entries that caused it. Figure 3-9(c) shows the fill pattern in U after the first major step. All the new fill is marked with \bullet .

Each subsequent minor step results in a block fill of size $(l + u)$ -by- $(l + u - 1)$ in U in the lower triangular part, except the first block which is of size $(l + u)$ -by- l . The fill in the upper triangular part of U is still one super diagonal for a every column and row pair. The structure of the fill in U is similar to Figure 3-7(c) except the size of the blocks is larger.

The reason for the block Hessenberg structure when applying the plane rotations in the first major step is not straight forward as in the symmetric case, but we describe it here for completeness. The plane rotations to reduce the entries in the first column is between rows $l + 1 \dots 1$ resulting in a lower Hessenberg block of the size $(l + 1)$ -by- $(l + 1)$ in U as we saw in the symmetric case. The plane rotations from the corresponding chase results in the same lower Hessenberg block structure fill in U , but each of these blocks is separated by $u - 1$ columns (not u). When we reduce the first row later, the plane rotations are between columns $(u + 1) \dots 2$ of A . Note that column $u + 1$ of A should have been involved in the chase to reduce the fill generated from reducing the entry $A(1, 2)$ (as $l > 0$). When we reduce the entry in $A(1, u + 1)$ and chase the resultant subdiagonal fill, applying the plane rotation to U results in fill of size $l + 2$ in the lower triangular part of U . This fill is caused by the existing Hessenberg blocks in U of order $l + 1$. In other words, as column $u + 1$ in A is part of the chase when reducing the first column of A and part of the actual reduction when reducing the entries in first row of A the interaction of the fill leads to a bigger Hessenberg block of order $(l + u)$.

Unsymmetric A , $r > 1$: When $r > 1$ the reduction Algorithm 3.1 leaves r diagonals in the upper and lower triangular part, whereas when $r = 1$ above we reduce the matrix to bidiagonal in one iteration, which is equivalent to leaving $r - 1$ diagonals in the lower triangular part. We need to leave r diagonals in the lower triangular part instead of $r - 1$ diagonals consistent with the $r = 1$ case to avoid the one column interaction between the plane rotations of the lower and upper triangular part as explained above in $r = 1$ case, so

that the fill pattern in U remains simple in $r > 1$ case. Without this small change, the fill pattern in U is still predictable, but is vastly different from the previous three cases. We do not discuss that fill pattern here.

Applying the plane rotations from reducing and chasing an fill of the first r columns of A , to U , results in diagonal blocks in U , where each block is $(l - r, r)$ -band matrix of the order l . Each of these blocks is separated by a distance of u . The next set of plane rotations, to reduce the entries in r rows in the upper triangular part of A , causes fill in U in the u columns between the existing Hessenberg blocks in U . The block fill is a $(u - r, r)$ -band matrix of the order u . Thus after applying the rotations in the first major step U is block diagonal with two different type of blocks (as described above) alternating in the diagonal. Figure 3-10(b)-(c) shows the fill pattern after reducing the entries in the first r columns and rows respectively. The fill from the first minor step is numbered for the corresponding entries that created the fill in both the figures.

Reducing r columns in A after the first major step creates a block fill in the lower triangular part of U that consist of a rectangular block of the size $(u - r + 1)$ -by- $(l - r)$ and below that a block of $(0, l - r)$ -band matrix of size $(r - 1)$ -by- $(l - 1)$ with a zero diagonal. Figure 3-10(e) shows the fill while reducing the entries from the third and fourth columns marked \oplus in Figure 3-10(d). Reducing the second set of r rows creates a block fill in the lower triangular part of U that consist of a rectangular block of the size $(l - r + 1)$ -by- $(u - r)$ and below that a block of $(0, u - r)$ -band matrix of size $(r - 1)$ -by- $(u - 1)$ with a zero diagonal. Figure 3-10(f) shows the fill while reducing the entries from the third and fourth rows (marked \oplus) of Figure 3-10(d).

From the above description of the fill for the four cases, it is straight forward to keep track of the first and last row of any column of U . For any given column k , the first row in U to which a plane rotation that reduced an entry in row i and the plane rotations to chase its fill is applied is $k - i$. This holds true in all four cases above. For keeping track of the last row for any column in U , we only need to keep track of the last row of

the first block of U and V and update it after each major step based on the above four cases. While reducing a particular set of r column and rows and chasing the fill we can obtain the last row for a given column from the last row of the first block and the number of minor steps completed and the fill size based on the above cases. We leave the indexing calculations out of the chapter for a simpler presentation.

3.4.2 Floating Point Operations for Update

Let us consider the case of reducing (l, u) -symmetric band matrix of order n to tridiagonal form again. The number of plane rotations required to reduce the symmetric matrix using Schwarz's diagonal reduction method is

$$g_d = \sum_{d=3}^b \sum_{i=d}^n 2 + 2 \left\lfloor \frac{\max(i-d, 0)}{d-1} \right\rfloor \quad (3-8)$$

where $b = u + 1$. The number of plane rotations required to reduce the symmetric matrix using Schwarz's row reduction method is

$$g_r = \sum_{i=1}^{n-2} \sum_{d=3}^{\min(b, n-i+1)} 2 + 2 \left\lfloor \frac{\max(i-d, 0)}{b-1} \right\rfloor \quad (3-9)$$

In order to find the number of floating point operations for our blocked reduction, when block size is r -by- c we use the two step approach as before to get

$$g_{b_1} = \sum_{i=1}^{n-r-1} \sum_{d=r+2}^{\min(b, n-i+1)} 2 + 2 \left\lfloor \frac{\max(i-d, 0)}{b-1} \right\rfloor \quad (3-10)$$

$$g_{b_2} = \sum_{i=1}^{n-2} \sum_{d=3}^{\min(r+1, n-i+1)} 2 + 2 \left\lfloor \frac{\max(i-d, 0)}{r} \right\rfloor \quad (3-11)$$

$$g_b = g_{b_1} + g_{b_2} \quad (3-12)$$

Asymptotically, number of rotations in all the three methods are the same, but there is a significant difference in the constant factor. Let us assume that we do not exploit the structure of U . All of the plane rotations are applied to one column of U (each of size n)

then. Reducing the number of plane rotations plays a significant role in reducing the flops for accumulation of the plane rotations.

Given a 1000-by-1000 matrix, and $l = 150$, Schwarz's row reduction method computes the bidiagonal reduction using 78% fewer plane rotations than the diagonal reduction method. It performs 43% fewer plane rotations than the blocked reduction method (with the default block size). When $l = 999$, the the row reduction method computes 83% and 48% fewer plane rotations than the diagonal reduction and blocked reduction methods respectively. The increase in the number of plane rotations for the blocked method is due to the rotations to reduce the r diagonals as a second iteration (g_{b2} in Equation 3-11). These rotations are especially costly they are applied to a full U .

We can exactly mimic the Schwarz's row reduction and still do blocking by choosing $r = 1$, leading to a 50% reduction in the flop count. Note that the cost of small increase in the flops when we reduce A without the accumulation was offset by the blocking, but in this case, when the flops almost doubles it is important to use $r = 1$. The default block size will use $r = 1$ when the plane rotations are accumulated. Unless there are some compelling reasons, say improvements like blocking the accumulation in U itself which offsets the cost in flops $r > 1$ case will be practically slower when U and V are required.

3.5 Performance Results

We compare our band reduction routine `piro_band_reduce` against that of SBR's driver routine `DGBRDD` [Bischof et al. 2000a] and LAPACK's symmetric and unsymmetric routines `DSBTRD` and `DGBBRD` [Anderson et al. 1999] in this section. All the tests were done with double precision arithmetic with real matrices. The performance measurements were done on a machine with 8 dual core 2.2 GHz AMD Opteron processors, with 64GB of main memory. `PIRO_BAND` was compiled with `gcc 4.1.1` with the options `-O3`. `LAPACK` and `SBR` are compiled with `g77` with the options `-O3 -funroll-all-loops`.

Figure 3-11 shows the raw performance of our algorithms with the default block size used by our routines. We estimate the default block size based on the problem size.

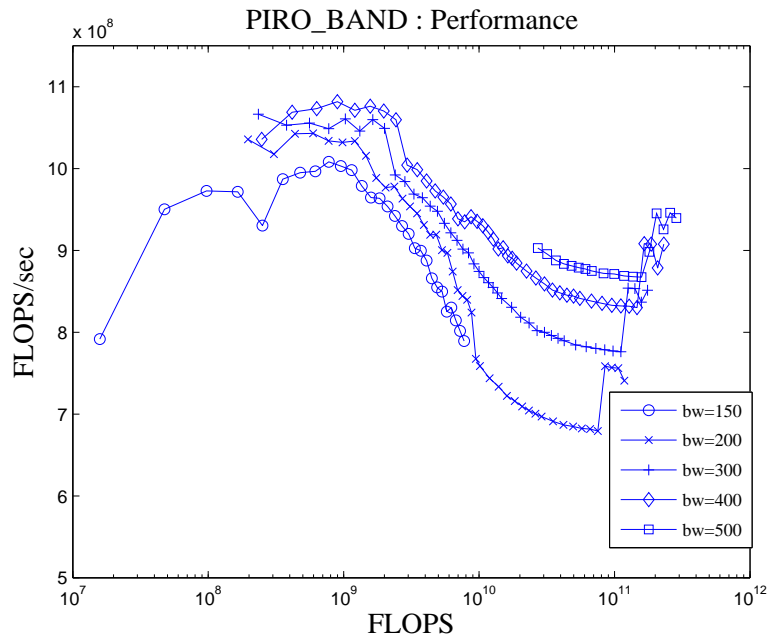


Figure 3-11. Performance of the blocked reduction algorithm with default block sizes

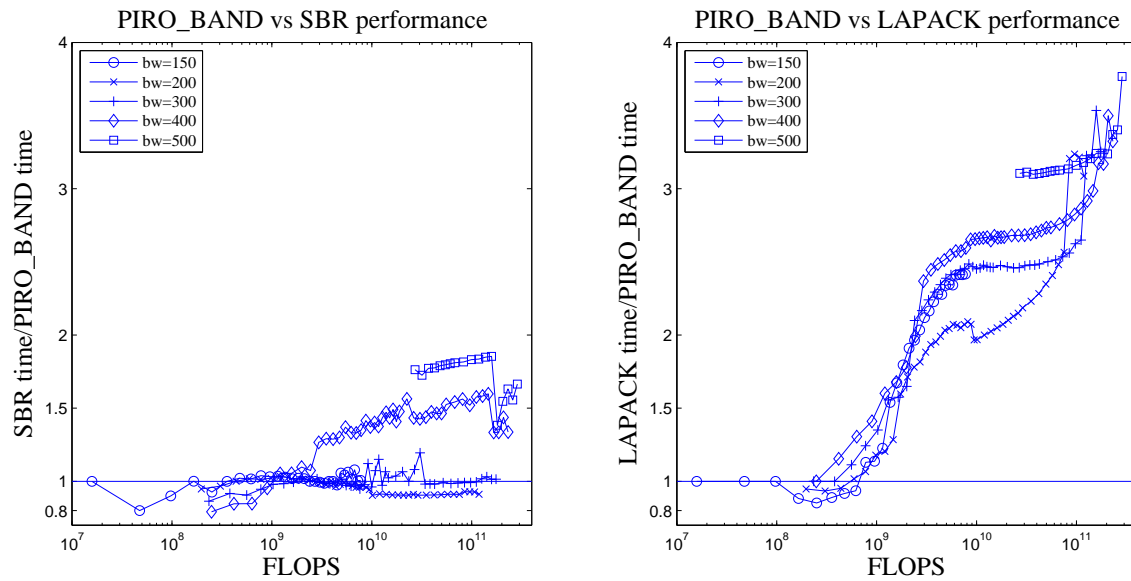


Figure 3-12. Performance of `piro_band_reduce` vs SBR and LAPACK

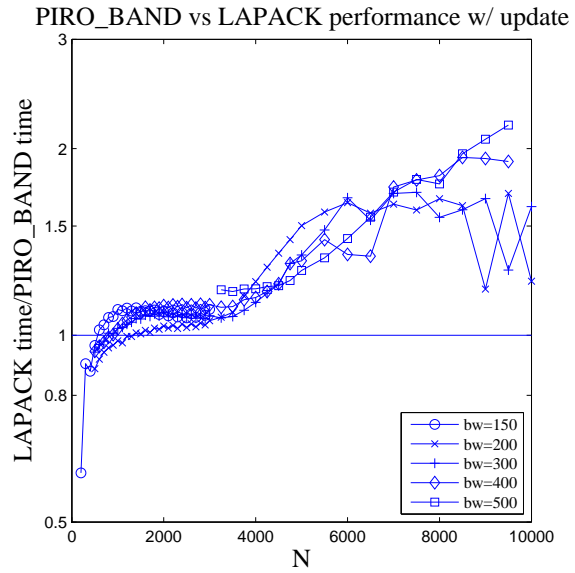
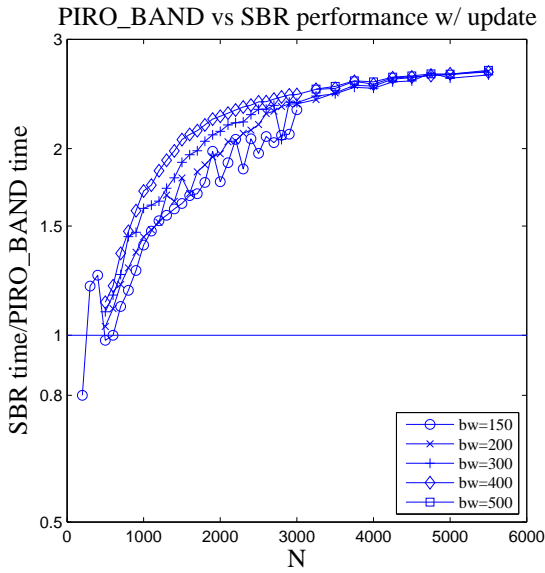


Figure 3-13. Performance of `piro_band_reduce` vs SBR and LAPACK with update

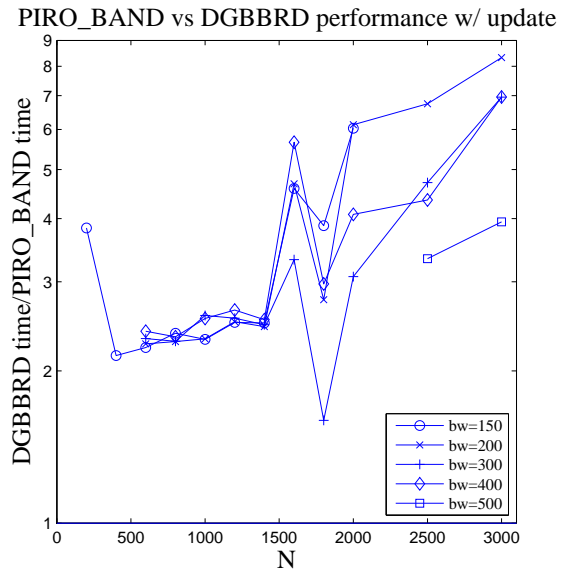
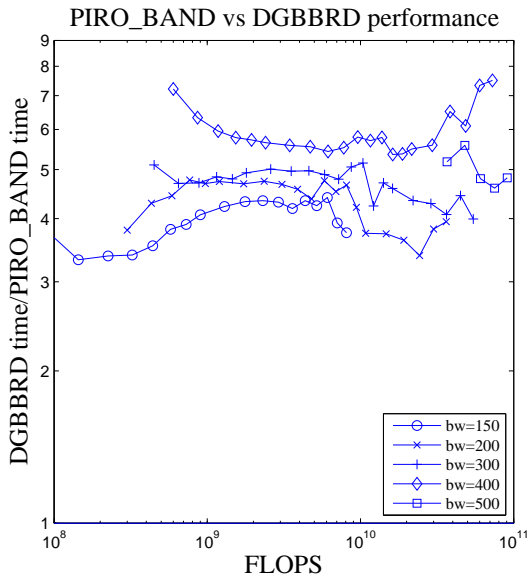


Figure 3-14. Performance of `piro_band_reduce` vs DGBBRD of LAPACK

The block size estimate is not based on hardware or compiler parameters. Performance of our algorithms depend to some extent on the block size. The optimal block sizes can vary depending on the architecture, compiler optimizations and problem size. The default block size is not the optimal block size for all the problems, even in this particular machine. Nevertheless, the results from default block size is used for all the performance comparisons in this section. The effects of the different block sizes on the performance are discussed in [Rajamanickam and Davis 2009b] and in Section 4.3.1

Figure 3-11 shows the effect of blocking for different bandwidths. The tests were run on symmetric matrices of the order 200 to 10000 with bandwidths 150 to 500. The band reduction algorithm performs slightly better for problems with non-narrow bandwidths than for the problems with narrow bandwidths. The figure shows the effect of cache: helping the performance for smaller matrices and then negatively affecting the performance as the problem size increases, but blocking stabilizes the performance as the problem size increases.

Given a problem size, we use the floating point operations from the Schwarz’s row reduction method as the flops required for that problem. The different algorithms we compare may do more floating point operations than Schwarz’s row reduction method (no algorithm will do less). We use the smallest possible flops for a given problem (the flops from Schwarz’s row reduction method) as a baseline to compare all the algorithms. The graphs in Figures 3-11, 3-12, 3-14 use the FLOPS from Schwarz’s row reduction method.

Figure 3-12 compares the performance of our blocked algorithm and SBR’s band reduction routines, without the accumulation of plane rotations. We provide SBR the maximum workspace it requires $n \times u$ for a matrix of size (u, u) -band matrix of order n and let it choose the best possible algorithm. For smaller matrices, SBR is about 20% faster than our algorithm. When the problems get larger, for smaller bandwidths, the performance of both the algorithms are comparable even though the work space required by our algorithm is considerably smaller (32-by-32). As the bandwidth increases our

blocked algorithm is about 25% to 2x faster than SBR algorithms while using only fraction of the work space.

The results of the performance comparison against LAPACK's DSBTRD is shown in Figure 3-12. LAPACK's routines does not have any tuning parameters except the size n workspace required by the routines. DSBTRD performs slightly better than our algorithm for smaller problem sizes. As the problem sizes increase, our algorithm is faster than DSBTRD even for smaller bandwidths. The performance improvement is in the order of 1.5x to 3.5x. DSBTRD uses the revised algorithm as described in [Kaufman 1984].

When we accumulate the plane rotations we use one entire row of the symmetric matrix as our block as this cuts the floating point operations required by nearly half when compared to a block size of 32-by-32. The performance of our algorithm against SBR and LAPACK algorithms are shown in the Figure 3-13. Our algorithm is faster than SBR by a factor 2.5 for larger matrices. It is 20% to 2 times faster than LAPACK's DSBTRD implementation. As in the no accumulation case, the larger the bandwidth the better the performance improvement for our algorithm.

We compare our algorithm against LAPACK's DGBBRD for the unsymmetric case. DGBBRD does not use the revised algorithm of Kaufman yet, so it can do better in the update case. The results when we compare our algorithm against the existing version of DGBBRD are in Figures 3-14 without and with the accumulation of plane rotations. Our algorithm is about 4 to 7 times faster in the former case and about 2 to 8 times faster in the later case.

3.6 Summary

The problem of band reduction had two possible style of solutions: a vectorized solution using plane rotations implemented in LAPACK libraries and a BLAS3 based solution using QR factorization and Householder transformations implemented in SBR toolbox. We presented a blocked algorithm in this chapter that is better than the competitive methods in terms of the required workspace, the number of floating point

operations and real performance. The software package that implements this algorithm, PIRO_BAND, is described in detail in Chapter 4.

CHAPTER 4 PIRO_BAND: PIPELINED PLANE ROTATIONS FOR BAND REDUCTION

4.1 Overview

Band reduction methods are an important part of algorithms for eigen value computations of symmetric band matrices and the singular value decompositions of unsymmetric band matrices. We introduced new blocked algorithms for band reduction in Chapter 3. The software implementing the algorithms PIRO_BAND (pronounced “pyro band”) is described in this chapter.

PIRO_BAND is a library for tridiagonal reduction of symmetric matrices and bidiagonal reduction of unsymmetric band matrices. We also provide functions to compute the band singular value decomposition using the band reduction functions and a simplicial left-looking band QR factorization. We call this QR factorization simplicial factorization as it does not use any supernodes or fronts to take advantage of BLAS3 style operations. See [Davis 2009b] for a description of a multifrontal QR factorization.

Section 4.2 introduces the features in the PIRO_BAND library. The comparison of the band reduction algorithms in PIRO_BAND against other band reduction algorithms is given in Section 3.5. We present some additional performance results of PIRO_BAND in Section 4.3.

4.2 Features

PIRO_BAND is a library with both MATLAB and C interfaces for band reduction. The C interfaces are described in Sections 4.2.1 and 4.2.2. The MATLAB interfaces are described in Section 4.2.3.

PIRO_BAND accepts general sparse and dense matrices in the MATLAB interface. The C library requires the input matrices to be in the packed band format. Given an m -by- n matrix A with bl diagonals in the upper triangular side and bu diagonals in the lower triangular side, the packed band data structure is of the size $(bl + bu + 1)$ -by- n where entry $A(i, j)$ is stored in entry $(i - j + bu, j)$. The MATLAB interface will accept

both sparse and dense matrices and convert them to band form using the structure of the matrix when the input is sparse and the numerical values when the input is dense.

PIRO_BAND's C interface and the MATLAB interface support real and complex matrices. Complex matrices are expected to be stored in the C99 style complex format where the real and imaginary part of every entry is stored in consecutive memory locations. We do not require C99 for compiling and using the libraries, but the test coverage for the library requires C99 support. PIRO_BAND C libraries do not support complex matrices where the real and imaginary part are stored in separate arrays.

4.2.1 PIRO_BAND Interface

The primary function for band reduction is `piro_band_reduce`. There are eight different versions of this function for all the combinations of double precision and single precision arithmetic, real and complex matrices and 32-bit and 64-bit integers. The eight versions of the functions can be defined using the following template.

```
piro_band_reduce_<x><y><z>
x := 'd' | 's' (for double or single precision)
y := 'r' | 'c' (for real of complex matrices)
z := 'i' | 'l' (for 32-bit or 64-bit integers)
```

For example, `piro_band_reduce_dri`, is the function name for using double precision arithmetic for reducing a real band matrix with 32-bit integers. The prototype for this function is

```
int piro_band_reduce_dri
(
    int blks[], int m, int n, int nrc, int bl, int bu, double *A,
    int ldab, double *B1, double *B2, double *U, int ldu, double *VT,
    int ldv, double *C, int ldc, double *dws, int sym
)
```

The `piro_band_reduce` functions destroy their input matrix A as they reduce it to bidiagonal/tridiagonal form. The two diagonals are returned on success. The plane rotations are accumulated in the right and left only if they are required. We do not provide a separate interface for symmetric and unsymmetric matrices. Instead, we use an

input flag to differentiate between the two: the last parameter to the function `sym` is true for a symmetric matrix.

All the eight versions of the `piro_band_reduce` functions require an optional block size (`blks`) and a workspace (`dws`) as a parameter. `blks` is an array of size four where the first two entries specify the number of columns (c_u) and rows (r_u) in the block for the reducing the super diagonals. The next two entries in `blks` specify the number of rows (r_l) and columns (c_l) in the block for reducing the subdiagonals. The number of rows and columns in the blocks are limited by

$$c_u + r_u < u \tag{4-1}$$

$$c_l + r_l \leq l \tag{4-2}$$

The block size can be different for the reductions in the upper triangular and lower triangular part. The work space should be two times the maximum of the two block sizes. The recommended block size is provided by the function `piro_band_get_blocksize` in architectures with 32-bit integers. The 64-bit version of the function has a suffix `_l`. `PIRO_BAND` gets the recommended block size and allocates the required work space if they are not passed to the library. See the `PIRO_BAND` user guide [[Rajamanickam and Davis 2009c](#)] for the exact prototype and the description of the parameters. The selection of the default block size is explained in Section [4.3.1](#).

The `PIRO_BAND` interface has some differences from the LAPACK style interface we provide (described in Section [4.2.2](#)), but it is faster as it will avoid a few transposes. The major differences between the two interfaces are :

- `PIRO_BAND` requires the upper bandwidth to be at least one.
- For symmetric matrices `PIRO_BAND` requires the upper triangular part to be stored.

Table 4-1. MATLAB interface to PIRO_BAND

MATLAB function	Description of the function
<code>piro_band</code>	Bidiagonal reduction routine for band matrices
<code>piro_band_qr</code>	QR factorization of a band matrix
<code>piro_band_svd</code>	SVD of a band matrix
<code>piro_band_lapack</code>	MATLAB interface to the LAPACK style interfaces
<code>storeband</code>	Store a sparse/full matrix in packed band format

- PIRO_BAND finds $C^T U$ instead of $U^T C$, and V instead of V^T , as in LAPACK. These are more efficient to compute.

The LAPACK style interfaces given below do not have any of these restrictions and mimic the functionality of LAPACK subroutines.

4.2.2 LAPACK Style Interface

The LAPACK style interface supports all eight functions for band reduction in the LAPACK library. The corresponding function names in PIRO_BAND have a prefix `piro_band_` added to it. For example, LAPACK's band reduction function `DSBTRD` is `piro_band_dsbtrd` in our interface. For the version that supports 64-bit integers we add a suffix `_l` to the function name. The interface has the exact functionality as the LAPACK libraries except for one difference : the return values of the functions are different in certain cases. Like LAPACK's functions, a return value of zero is success and a negative value means failure, but the exact error codes are different from LAPACK's functions as we use one common function for symmetric and unsymmetric matrices. A return value of `-i` may not indicate the `i`th argument is invalid in the LAPACK style interfaces.

4.2.3 MATLAB Interface

The MATLAB interface in PIRO_BAND provides the functions listed in Table 4-1. The MATLAB functions `piro_band` and `piro_band_lapack` are provide a simple interfaces for the band reduction algorithms both in PIRO_BAND style and LAPACK style. `storeband` finds a band structure, if one exists, and stores the matrix in band format. It uses the numerical values when the input is a dense matrix. It uses the structure of the matrix if the input is sparse to find the band matrix.

The function `piro_band_svd` computes the singular value decomposition of a band matrix. It can compute both the full and economy singular value decomposition of the band matrix. The usage also supports finding only the singular values. The singular value decomposition function uses our band reduction algorithm to reduce the matrix to bidiagonal/tridiagonal form. It then uses LAPACK (bundled within MATLAB) to diagonalize bidiagonal/tridiagonal matrix to compute the singular values. The `piro_band_svd` function computes the economy singular value decomposition in three steps. It uses the simplicial left-looking QR factorization to compute an upper triangular banded matrix R . It then reduces R using the bidiagonal reduction algorithm and the LAPACK diagonal reduction algorithm.

`piro_band_qr` provides a MATLAB interface to the left-looking band QR factorization. It provides an option to find both Q and R and an option to find the Householder vectors instead of Q . See PIRO_BAND user guide [Rajamanickam and Davis 2009c] for the exact usage of all these functions.

4.3 Performance Results

All the performance results here were taken on a machine with eight dual core 2.2 GHz Opteron processors with 64GB of main memory. We used MATLAB 7.6.0.324 (R2008a). PIRO_BAND was compiled with gcc version 4.1.1 with the option `-O3`.

4.3.1 Choosing a Block Size

Choosing the right block size for different machines is a difficult problem. We provide a function in C that estimates the recommended block size based on the number of floating point operations. The function will not optimize the block size for any specific hardware or compiler. The simple rule is to use a block size of 8-by-8 when the required flops is less than 10^{10} and the block size of 32-by-32 otherwise. This is our default block size. The function `piro_band_get_blocksize` uses $6kn^2$ as an estimate for the number of FLOPS for the band reduction algorithm. Figure 4-1 uses the estimated FLOPS for the x-axis.

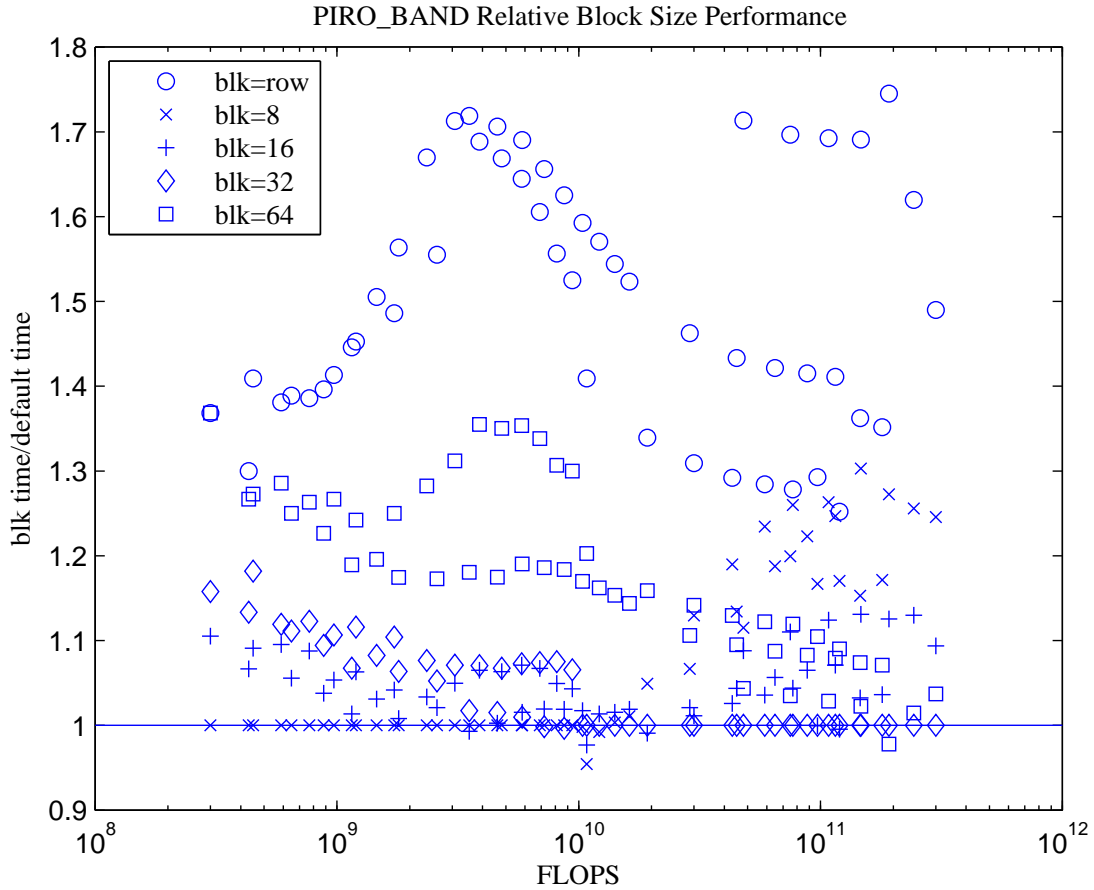


Figure 4-1. Performance of `piro_band_reduce` for different block sizes

To verify this simple block size selection we compare five different options : an entire row as the block, block sizes of 8-by-8, 16-by-16, 32-by-32 and 64-by-64. When the entire row is used as the block the blocked reduction algorithm does exactly the same amount of flops as Schwarz' row reduction method. It does more flops (by a constant factor) for the other four blocking sizes. Figure 4-1 shows the performance comparison of various block sizes in relation to the default block size. When the flop count is smaller 8-by-8 is the ideal block size. As the flops reach 10^{10} there are a few cases where 16-by-16 is better, but 32-by-32 is ideal when the flop count is higher. Note that except when choosing the entire row as the block, all the other block sizes perform within 30% of the default size. We can also see that 64-by-64 is almost as good as 32-by-32 when we do lot of floating

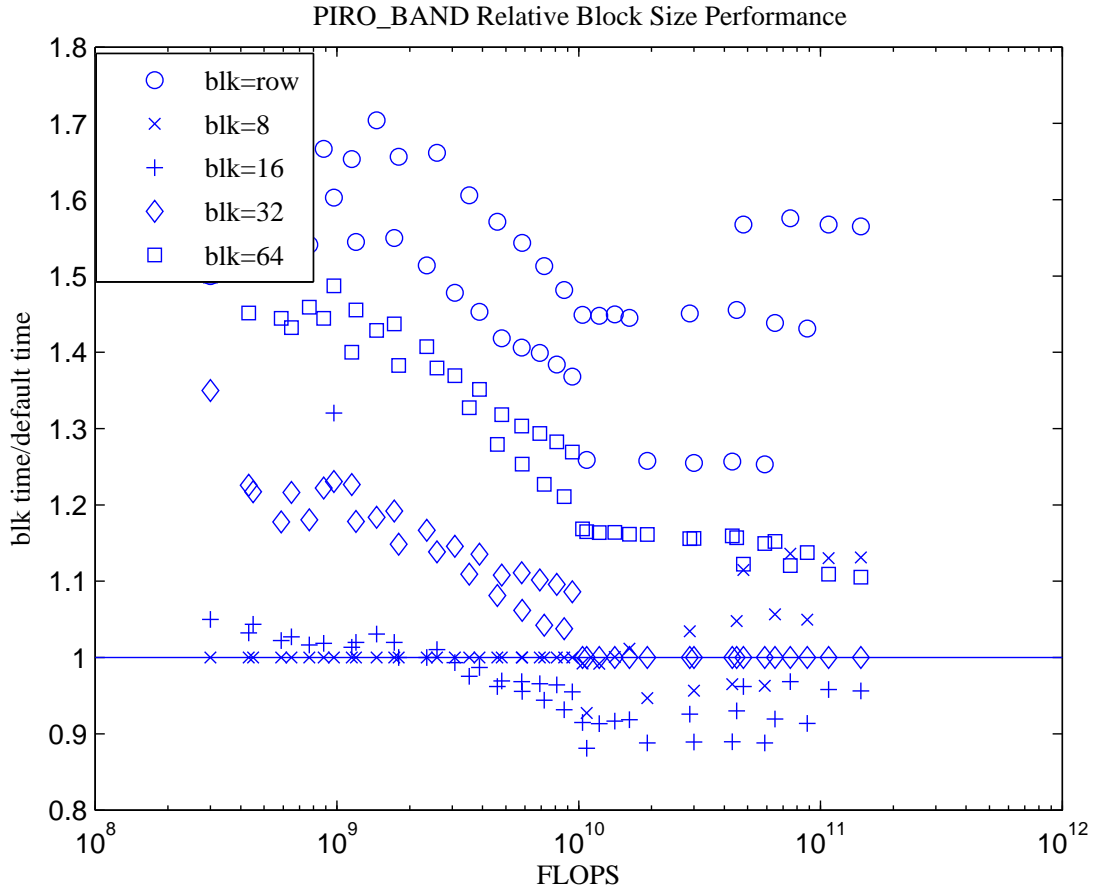


Figure 4-2. Performance of `piro_band_reduce` for different block sizes

point operations. There can be a cut-off point where 64-by-64 may start performing better than 32-by-32. We chose to leave the default at 32-by-32 and let the user experiment for their common problem sizes.

As the block sizes are platform dependent we repeated this experiment on a machine with a 3.2 GHz Pentium 4 and 4 GB of memory. The results are shown in Figure 4-2. Though the performance of various block sizes are somewhat different from Figure 4-1 the default block size is never more than 15% worse than the best block size which in this case is 16-by-16 for some problems.

Table 4-2. PIRO_BAND svd vs MATLAB dense svd

N	Bandwidth	PIRO_BAND SVD (seconds)	MATLAB DENSE SVD (seconds)
1000	100	1.7	47.1
2000	200	11.2	149.3
3000	300	34.9	306.0
4000	300	64.2	540.5

4.3.2 PIRO_BAND svd and qr Performance

We compare the performance results of PIRO_BAND's singular value decomposition and PIRO_BAND's QR factorization with their MATLAB's equivalents in this section. Detailed performance results of PIRO_BAND package against the LAPACK and SBR band reduction routines can be found in Section 3.5.

MATLAB does not have a band singular value decomposition. The sparse singular value decomposition is too slow when compared with the dense singular value decomposition, especially when all the singular values are required. We compare our band singular value decomposition against the dense algorithm which is better when one needs to compute all the singular values. As we operate only on the band, our algorithm is asymptotically faster than the dense SVD. The timing results for finding only the singular values are summarized in Table 4-2.

We compare PIRO_BAND's simplicial left-looking QR factorization against MATLAB's dense QR factorization and MATLAB's sparse QR factorization. The performance results are provided in Table 4-3. We should note that MATLAB's sparse QR is not a very fast algorithm. As our QR factorization is a support routine used only for economy singular value decomposition we use a simplicial algorithm. An efficient QR factorization algorithm like the multifrontal algorithm SPQR [Davis 2009a] will be able to do better than the simplicial QR factorization algorithm with a small additional cost in the integer workspace. The latest release of MATLAB R2009b includes SPQR as its QR factorization and should have better results than the ones presented here for the sparse

Table 4-3. PIRO_BAND qr vs MATLAB dense and sparse qr

N	Bandwidth	PIRO_BAND QR (seconds)	DENSE QR (seconds)	SPARSE QR (seconds)
1000	100	1.5	5.3	11.9
2000	200	14.7	34.0	159.9
3000	300	48.4	106.0	1907.3
4000	300	87.0	239.0	4145.9

case. QR factorization is used by PIRO_BAND only when the economy singular value decomposition is required.

CHAPTER 5
SPARSE R-BIDIAGONALIZATION USING GIVENS ROTATIONS

5.1 Introduction

Given a sparse matrix A we follow the three step process described in Section 1.1.1 to compute the SVD of A . There are various sparse QR factorization algorithms we can choose from to do the first step efficiently. Reducing a bidiagonal matrix to a diagonal matrix is a problem where numerical considerations are more important than sparsity considerations. The subject of this chapter covers the only other piece left in the puzzle: bidiagonalization of the upper triangular sparse matrix R from the QR factorization.

While trying to solve the sparse problem, we use the lessons from the band reduction algorithm. We do not generate more than one fill for any rotation. We block the rotations for better cache access and pipeline the rotations to avoid more than one fill. This chapter is organized as follows. Section 5.2 covers the theory for sparse bidiagonalization. We discuss various algorithms for sparse bidiagonalization in Section 5.3. The symbolic factorization for sparse bidiagonal reduction is presented in Section 5.4. Section 5.5 details the software implementing the reduction algorithms and its performance is covered in Section 5.6.

5.2 Theory for Sparse Bidiagonalization

5.2.1 Profile and Corner Definitions

Let $R \in \mathcal{R}^{n \times n}$ be the sparse upper triangular matrix from the QR factorization. We use the following definitions throughout this chapter. Some of the definitions and lemmas in this section are derived from [Davis and Golub 2003].

Definition 5.1 (Skyline Profile). *Skyline profile of R is defined as follows.*

$$\mathcal{S}(R) = \{(i, j) : 1 \leq j \leq n, (f_j(R) \leq i \leq j) \vee (j - i = 0) \vee (j - i = 1)\}$$

where

$$f_j(R) = \min\{i : 1 \leq i \leq j, r_{ij} \neq 0\}$$

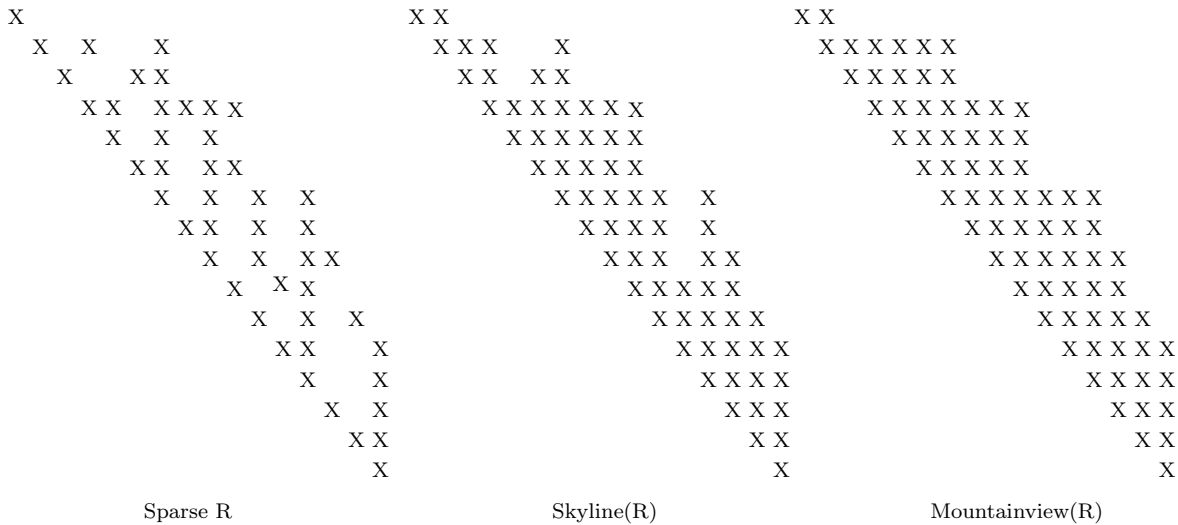


Figure 5-1. A sparse R , skyline profile of R and mountainview profile of R

or simply the skyline profile of R is the set of index pairs (i, j) that satisfy the following criteria:

- All entries in the upper bidiagonal are in the skyline profile.
- if $r_{ij} \neq 0$ then (i, j) is in the skyline profile.
- if (i, j) is in the profile and $i \neq j$ then $(i + 1, j)$ is in the skyline profile.

Definition 5.2 (Mountainview Profile). *The Mountainview profile of R is defined as follows.*

$$\mathcal{M}(R) = \{(i, j) : 1 \leq i \leq n, i \leq j \leq g_i(R)\}$$

where

$$g_i(R) = \max\{k : i \leq k \leq n, (i, k) \in \mathcal{S}(R)\}$$

or simply the mountainview profile of R is the set of index pairs that satisfy the following criteria:

- All entries in the skyline profile are in the mountainview profile.
- if (i, j) is in the mountainview profile and $i \neq j$ then $(i, j - 1)$ is also in the mountainview profile.

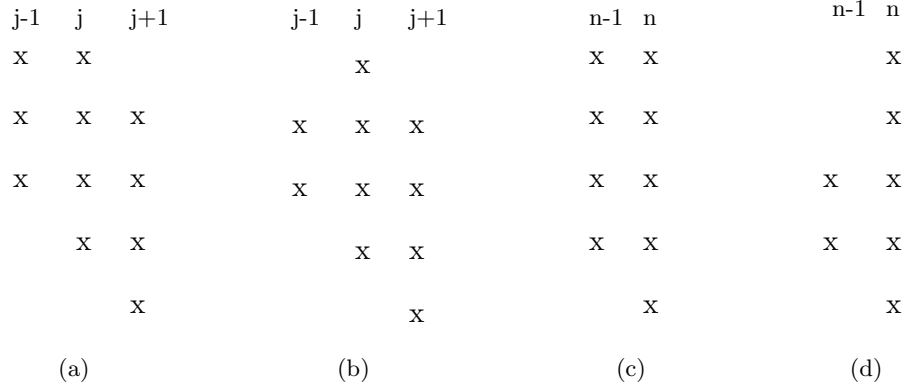


Figure 5-2. Corner definition in the profile of R for corner columns j and n

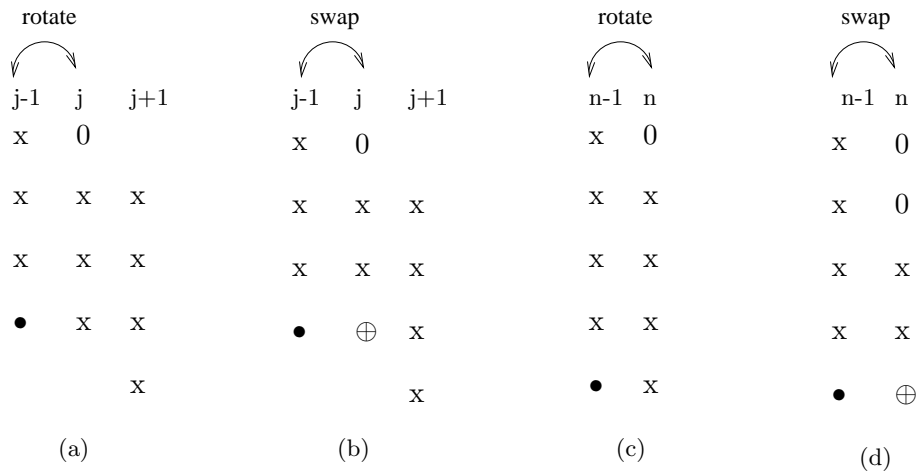


Figure 5-3. Corner columns j and n after column rotations

This style of defining the profile is similar to the envelope definition in [Hager 2002].

Figure 5-1 shows a small sparse R its skyline profile and mountainview profile.

Definition 5.3 (Top row). *Top row of a column j , $trow(j)$, is the smallest row index in the profile of column j .*

Definition 5.4 (End column). *End column of a row i , $ecol(i)$, is the largest column index in the profile of row i .*

Definition 5.5 (Corner). *A column j is a corner if*

$$\begin{aligned}
 & trow(j-1) \geq trow(j) && \text{when } j = n, \\
 & trow(j-1) \geq trow(j) < trow(j+1) && \text{when } 3 < j < n.
 \end{aligned}$$

corners in an upper triangular matrix (and chase the fill) to get a bidiagonal matrix with no corners. The fact that if column j is a corner then columns $j - 1$ and $j + 1$ are not corners helps us so we do not rotate two corners with one another.

Definition 5.6 (Rightmost / Leftmost Corner). *Column k is called the rightmost (leftmost) corner if k is largest (smallest) column index for which k is corner.*

Definition 5.7 (Rightmost Corner Entry). *If k is the rightmost corner we call the entry $(\text{trou}(k), k)$ as the rightmost column entry.*

Definition 5.8 (Rubble). *If k is the rightmost corner then the set of columns $k + 1 \dots n$ is called the rubble.*

Definition 5.9 (Height of a column). *Height of a column j is $j - \text{trou}(j)$.*

The corners, *trou* and *ecol* values are shown in Figure 5-4. The corners for the skyline matrix R from Figure 5-1 are $\{4, 7, 10, 13\}$. The *trou* and *ecol* values for the skyline profile of R from Figure 5-1 are

$$\text{trou}(\mathcal{S}(R)) = \{1, 1, 2, 2, 4, 3, 2, 4, 4, 4, 7, 10, 7, 9, 11, 12\}.$$

$$\text{ecol}(\mathcal{S}(R)) = \{2, 7, 7, 10, 10, 10, 13, 13, 14, 14, 15, 16, 16, 16, 16, 16\}.$$

The rightmost corner is column 13 and the leftmost corner is column 4. The rightmost column entry is $(7, 13)$. The rightmost corner entry is marked with a R and the leftmost corner is marked with a L. Columns $14 \dots 16$ are called the rubble.

5.2.2 Mountain, Slope and Sink

The mountainview profile has special properties that we can exploit while doing the reduction. The following definitions apply *both* to the mountainview and the skyline profiles, but are defined using the *mountainview profile* of R . We classify every column of the mountainview profile in the range $3 \dots n$ as one of the following 3 cases: mcolumn, sink or slope.

Definition 5.10 (Mcolumn). *A column j is an mcolumn (a column in a mountain) if $\text{trou}(j - 1) = \text{trou}(j)$ in the mountainview profile of R .*

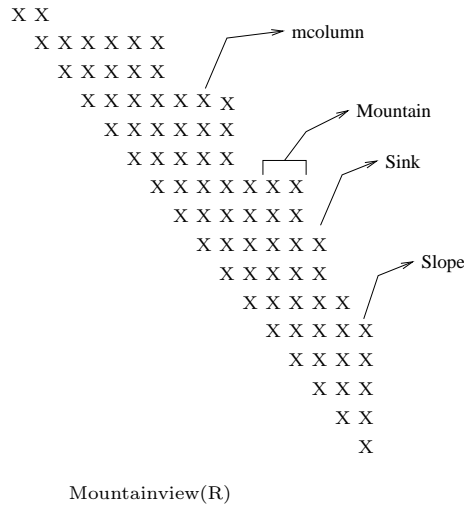


Figure 5-5. Types of columns in the mountainview profile

Definition 5.11 (Mountain). *A set of m columns all with the same top row value in the mountainview profile of R .*

Definition 5.12 (Slope). *A column j is a slope if $trow(j - 1) + 1 = trow(j)$ in the mountainview profile of R .*

Definition 5.13 (Sink). *A column j is a sink if $trow(j - 1) + 1 < trow(j)$ in the mountainview profile of R .*

Definition 5.14 (Base height of a mountain). *Base height of a mountain is the height of the m column with the smallest index in that mountain in the mountainview profile of R .*

Figure 5-5 illustrates the definitions with the mountainview profile of R matrix we used in Section 5.2.1. A sample m column, slope and sink are marked in the figure along with one of the mountains. There are three mountains in the mountainview profile of R including columns $\{4, 5, 6, 7\}$, $\{9, 10\}$ and $\{12, 13\}$. The base height of each of the mountain is 2, 5 and 5 respectively. Note that we do not define column 3 to be part of the first mountain even when it shares the $trow$ value with columns $4 \dots 7$. Column 3 is defined as a slope instead.

There are a few properties that follow directly from the definition. In the mountainview profile:

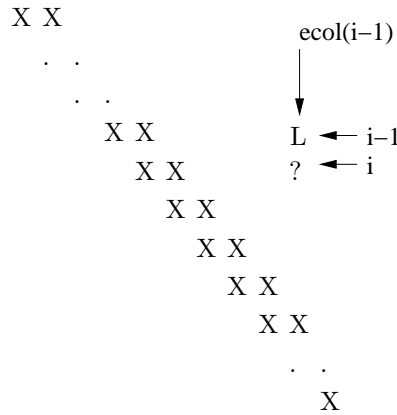


Figure 5-6. End column lemma

- A corner will always be an mcolumn.
- The rightmost corner is in the rightmost mountain.
- For any column j and its top row $i = \text{trow}(j)$, rotating rows i and $i - 1$ generates more than one fill if j is an mcolumn, at least one fill that cannot be chased immediately if j is a sink and one fill that can be chased if j is a slope.

Mcolumn height property. In a mountainview profile the height of the columns in any mountain monotonously increases from the lowest numbered column to the highest numbered column.

Proof. Every column j in the mountain has the same $\text{trow}(j)$ value from the definition. The smallest numbered column sets the base height. As j increases for every column in the mountain the height, $j - \text{trow}(j)$ increases too. \square

5.2.3 Properties of Skyline and Mountainview Profiles

There are potentially more than one corner in the profile of a sparse matrix R that we can reduce to get a fill in the subdiagonal. Furthermore, the chase to reduce the subdiagonal fill has to be handled as well. We need to characterize the fill and the profiles better to arrive at an algorithm for reducing the upper triangular matrix. The following properties help us do it.

Lemma 1 (End-column lemma). *Given the profile of R , $\text{ecol}(i - 1) \leq \text{ecol}(i)$ for all i in the range $2 \dots n$.*

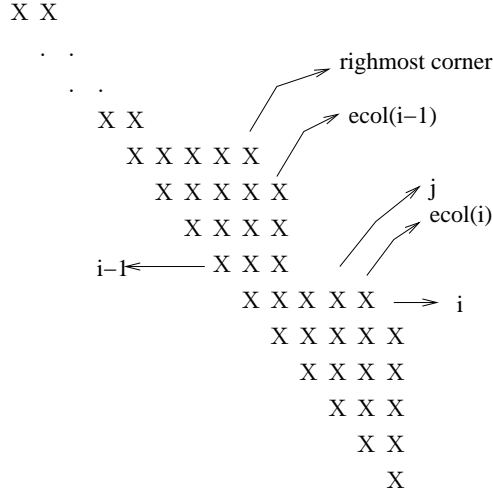


Figure 5-8. Rubble row lemma

We still have the case of j in the range $k + 2 \dots n - 1$. As we assumed $trow(j - 1) \geq trow(j)$, to avoid j being a corner (and result in a contradiction immediately) $trow(j) \geq trow(j + 1)$ must be true. As we proceed from columns $j - 1 \dots n$ we have the inequality

$$trow(j - 1) \geq trow(j) \geq trow(j + 1) \geq \dots \geq trow(n - 1) \geq trow(n)$$

This results in column n being the rightmost corner. A contradiction. This case is shown in Figure 5-7. \square

Lemma 4 (Rubble column lemma). *If k is the rightmost corner then for all j in the range $k \dots n$ the skyline profile is the same as the mountainview profile.*

Proof. Assume that there exists an entry $(i, j), k < j \leq n$ in the skyline profile of R such that $(i, j - 1)$ is not in the skyline profile of R . It is in the mountainview profile of R by definition. From the definition of the skyline profile, $(p, j - 1), 1 \leq p < i$ will not be in the skyline profile. So $trow(j - 1) > i$. From our assumption $trow(j) \leq i$. Combining the two we get $trow(j - 1) > trow(j)$ contradicting the Top-row lemma. \square

Lemma 5 (Rubble row lemma). *If k is the rightmost corner for all i in the range $trow(k) + 1 \dots n$, $ecol(i) - ecol(i - 1) = 0$ or 1 .*

Proof. From the End-column lemma, $ecol(i) - ecol(i - 1) \geq 0$. We just need to prove that $ecol(i) - ecol(i - 1) < 2$. Assume there exists two rows $i - 1$ and i in the range $trow(k) + 1 \dots n$ such that $ecol(i) - ecol(i - 1) \geq 2$. Figure 5-8 shows the rightmost corner and the rows $i - 1$ and i . Let j be a column in the range $ecol(i - 1) + 1 \dots ecol(i) - 1$. There is at least one such column. Column j satisfies $trow(j) = trow(j + 1) = i$ contradicting the Top-row lemma. \square .

The columns to the right of the rightmost column will not have a mountain as shown in the Top-row lemma. They satisfy the properties of a mountainview profile. Hence the name rubble. Lemma 4 and Lemma 5 fully define the characteristics of the rubble.

Theorem 5.1 (Rightmost Corner Theorem). *Reducing the rightmost column entry, $(trow(k), k)$, where k is the rightmost corner, using a pair of Givens rotations causes zero or one fill in the profile.*

Proof. From the property of the corner we get one fill in the entry $(k, k - 1)$ when we apply a Givens rotations to columns k and $k - 1$.

We apply a row rotation between rows k and $k - 1$ to remove the subdiagonal fill. As the row rotations are applied to columns in the range $k \dots n$ the rubble column lemma and mountainview lemma ensure that the total number of fill-in will be $ecol(i) - ecol(i - 1)$ (like the rotation between any two rows in the mountain view profile). Though column $k - 1$ is involved in this rotation, to which the rubble row lemma does not apply, we are chasing the fill in column $k - 1$. We cannot get a fill there by that rotation. As rows $k - 1$ and k are in the range $trow(k) + 1 \dots n$, the rubble row lemma ensures that the $ecol(i) - ecol(i - 1)$ will be either zero or one. \square

Lemma 6 (Rubble Height Lemma). *Let k be the rightmost corner in the profile, for all columns j in the range $k + 1 \dots n$, the height of the columns is monotonously non increasing.*

Proof. As the column number increases from k to n , from the top-row lemma (Lemma 3), the $trow$ values have to increase by at least 1 for each new column which

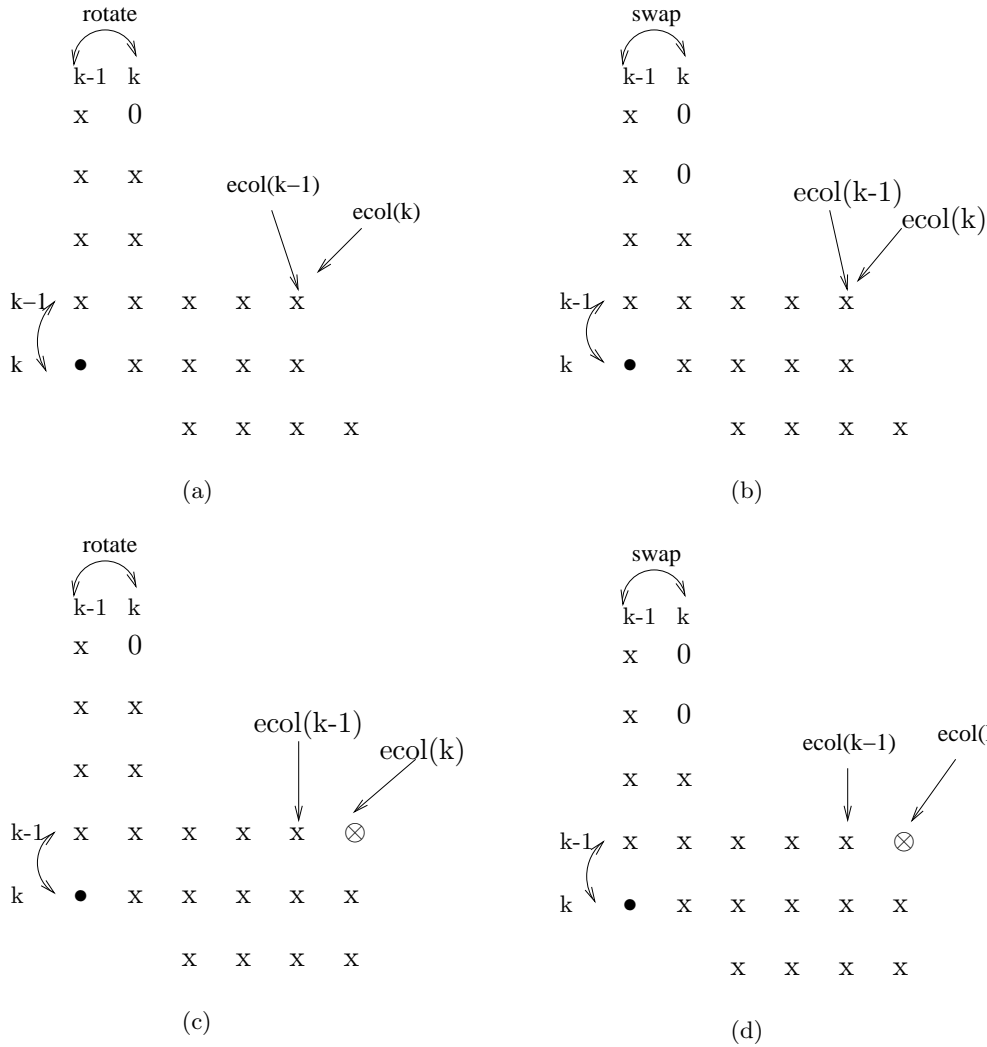


Figure 5-9. Different conditions to establish progress in reduction

ensures that the height of the columns remain at least the same. When $trow$ values increase by more than one the height of the corresponding columns decreases too. \square

Progress. There are four cases to consider when we try to establish progress when using the Theorem 5.1. We use either a swap or a normal rotation to reduce the rightmost corner and create a subdiagonal fill. We then reduce the subdiagonal fill with a row rotation. The row rotation may create zero or one fill as established by the rightmost corner theorem (Theorem 5.1). The four cases from the combinations of these two conditions are

1. rotate the rightmost corner, row rotation creates no fill.
2. swap the rightmost corner, row rotation creates no fill.
3. rotate the rightmost corner, row rotation creates one fill.
4. swap the rightmost corner, row rotation creates one fill.

Note that the fill generated cannot be chased all the time. We need to establish progress irrespective of the chasability of the fill. We need to remember the fact that the column rotation reduces the number of entries in the profile by one and the swap leaves the number of entries in the profile the same (ignoring the subdiagonal fill). In case 1, the column rotation reduces the profile by one entry and the row rotation leaves the number of entries in the profile unchanged. We have made progress by reducing one entry without any fill. In case 2, the number of entries in the profile remain unchanged, but the rightmost corner entry moved closer to the diagonal (from column k to column $k - 1$) making progress. Figure 5-9(a)-(b) shows these first two cases. The 0's show the entries that were just made zero either by a rotate or a swap. The subdiagonal fill is marked with a \bullet . There could have been more than one entry that got swapped reducing the profile even faster. For example, Figure 5-9(b) shows two 0's in column k for the two entries that got swapped. Both the entries got closer to the diagonal making progress.

In case 3, the column rotation reduced one entry in the profile and the row rotation created a fill. The rubble height lemma (Lemma 6) ensures that either all the columns from k to $ecol(k)$ are of the same height (as that of k) or the column $ecol(k)$ is shorter than k itself. In the former case when the all the columns from $k \dots ecol(k)$ are of the same height the fill created by the row rotation is away from the main diagonal by one more than the rightmost column entry and will be chasable as $trow(ecol(k)) = trow(ecol(k - 1))$ when the fill is at entry $(k - 1, ecol(k))$. Then the progress has to be established by one of the other cases. The Figure 5-9(c) shows the fill from row rotation as \oplus .

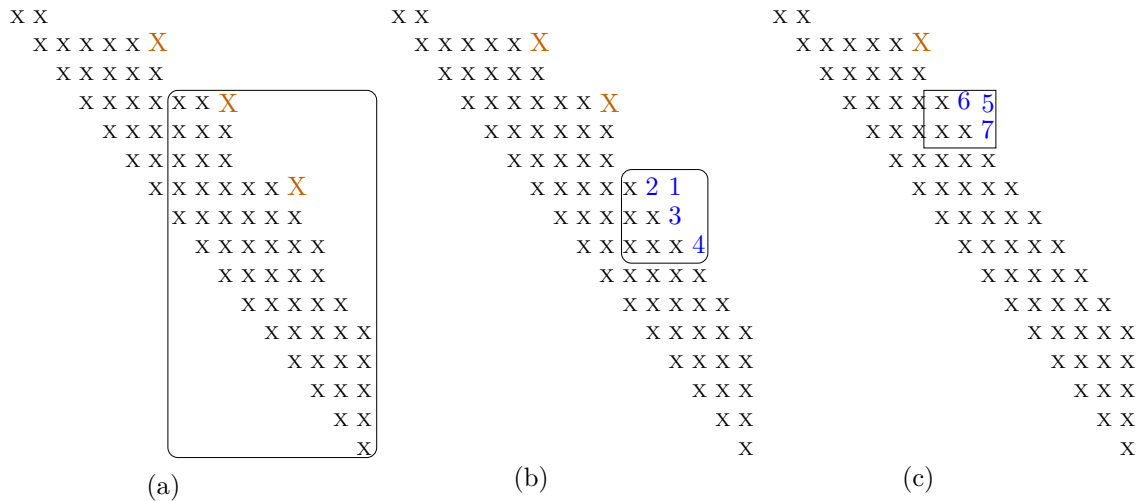


Figure 5-10. Potential columns for blocking in the mountainview profile

In the later case when column $ecol(k)$ is shorter than column k the fill will be at the same distance as the rightmost column entry or closer to the diagonal than the rightmost column entry. Progress is established in the later case as the fill is closer to the diagonal and the former case because the fill is closer to be chased out.

Figure 5-9 shows the swap case with fill. The swap leaves the number of entries unchanged and the row rotation creates one fill. The fill temporarily increases the number of entries in the skyline profile. But the new entry is closer to column n and will eventually be chased out of the matrix. Thus we make clear progress in three cases and in one case we temporarily increase the profile with an entry which is easier to reduce and chase out of the matrix.

5.2.4 Properties for Blocking the Bidiagonal Reduction

All the properties we have discussed in Section 5.2.3 leads us to a correct algorithm for sparse R -bidiagonalization. However, the algorithm will not be a blocked algorithm. We show that blocking the plane rotations is possible when reducing the rightmost corner in this section.

Theorem 5.2. *Reducing any corner in the range of columns $trow(k) + 1 \dots k$, where k is the rightmost corner, using a Givens rotation and chasing the fill causes zero or one fill in the mountainview profile.*

Proof. In the mountainview profile we do not need the Rubble-column lemma as the mountainview lemma guarantees that the Givens rotations are applied between two rows, say $i - 1$ and i , to eliminate a sub-diagonal fill, the new fill can only be in entries $(i - 1, ecol(i - 1) + 1) \dots (i - 1, ecol(i))$. Any corner, by definition, generates only one sub-diagonal fill when its corner entry is reduced to zero with a Givens rotation. The rubble-row lemma still applies to rows $trow(k) + 1 \dots k$. So any corner in the range $trow(k) + 1 \dots k$ will cause zero or one fill in the mountainview profile. \square

The progress statement for the rightmost corner applies even when we use Theorem 5.2. However, Theorem 5.2 does not apply to the skyline profile as the mountainview Lemma 2 does not apply to skyline profile and the fill need not just be in $(i - 1, ecol(i - 1) + 1) \dots (i - 1, ecol(i))$ when a Givens rotations is applied to rows $i - 1$ and i . This theorem shows us that there are multiple corners that can be reduced with zero or one fill. When they are within a few columns $trow(k) + 1 \dots k$ apart this theorem shows that blocking might be possible, at least in the mountainview profile. Figure 5-10(a) shows the columns from where we can choose our corner so that we can effectively block the plane rotations. We can see from the figure that there are two corners that can be reduced with zero or one fill. We can choose these columns as the block and reduce the corners as they appear in one of these columns after reducing the initial corners. But we can characterize the algorithm better using the rightmost mountain theorem given below.

Theorem 5.3 (Rightmost Mountain Theorem). *Let j be the least column in the rightmost mountain and k be the rightmost corner, if we always reduce the rightmost corner and chase the fill, then when a column m in the range $3 \dots j - 1$ gets chosen as the rightmost corner for the first time, the height of all the columns in the range $j \dots n$ will be less than the original height of j .*

Proof. Let m be the first corner chosen as rightmost corner in the range $3 \dots j - 1$. Column $j - 1$ will still be at its original height, one less than the height of j , as m is the first corner chosen in the range $1 \dots j - 1$. Let us assume there exists a column p in the range $j \dots n$ such that $\text{height}(p) \geq (\text{original}) \text{height}(j)$.

As we are considering the columns in the rubble (to the right of m , the current rightmost corner) we can use the rubble height lemma (Lemma 6) to see that height of $p - 1$ should remain the same or increase from the height of p . This leads to

$$\text{original height}(j) \leq \text{height}(p) \leq \text{height}(p - 1) \leq \dots \leq \text{height}(j)$$

All the inequalities in the above equation should be equalities as $\text{height}(j)$ cannot increase as m is the first corner in the range $3 \dots j - 1$. If the heights of all the columns in the range $p - 1 \dots j$ remain the same, i.e the inequalities above are all equalities, then $\text{height}(j) = \text{height}(j - 1)$ resulting in a corner in j which is to the right of the rightmost corner m . A contradiction. \square

Theorem 5.3 shows that if we reduce the rightmost corner always and chase the fill then we reduce the rightmost mountain and the rubble to a height less than the base height of the rightmost mountain before we reduce any corner from the left of the mountain. In short, always reducing the rightmost corner actually is a method that always reduces the rightmost mountain. We did not use any of the special properties of the mountainview profile here except the definitions. Theorem 5.3 holds for the skyline profile with a small change : All the columns in the range $j \dots n$ will have height less than or equal to the original height of $j - 1$. The proof remains the same.

This shows one can effectively block the reduction in the mountainview profile. The size of the block depends on the size of the mountain. The size of the block is dynamic since the size of the mountain changes. Figure 5-10(b) shows the block and first four corners that will be reduced together as one block. Note that though the first three corners are in the mountain, one column in the rubble contributes to the block. Figure

5-10(c) shows the second mountain and the three corners that will be reduced as one block. There is no rubble column that can be added to this block for the reduction. The corner entries 1...7 in these figures are chosen in that order because the rightmost corner method would have chosen them too.

The blocking in the skyline profile is a little more trickier. It is presented in detail in Section 5.3.2. We see all the algorithms to reduce the sparse R in Section 5.3.

5.3 Bidiagonalization Algorithms

The simplest algorithm to reduce R uses the result of Theorem 5.1 that reducing the rightmost corner results in zero or one fill. Algorithm 5.1 reduces the rightmost corner and uses a plane rotation to reduce the subdiagonal fill. This algorithm does not use blocking or pipelining.

Algorithm 5.1 Rightmost corner algorithm for bidiagonalization

- 1: Find all the corners from $1 \dots n$ and push them into a stack.
 - 2: **while** there are more corners **do**
 - 3: Pop the rightmost corner k
 - 4: Reduce the rightmost corner entry by applying a rotation between columns k and $k - 1$.
 - 5: Reduce the fill by applying a rotation between rows k and $k - 1$.
 - 6: Inspect columns $k - 1, k + 1$, and $ecol(k)$ for new corners and push the new corners to stack.
 - 7: **end while**
-

We do not deal with the fill explicitly in this algorithm. If the fill in column $ecol(k)$ can be chased it will be the new rightmost corner and chased. If the fill can not be chased then we reduce the next rightmost corner. Given a band matrix in sparse form the rightmost corner algorithm reduces it exactly like Schwarz's diagonal reduction method (Algorithm 2.2). Just like the diagonal reduction method the rightmost corner algorithm is not blocked, does more work and results in poor performance.

When the rightmost corner method is similar to the diagonal reduction method [Schwarz 1963] it is natural to look for a method to reduce R like the Schwarz's row reduction method [Schwarz 1968]. We need to choose the leftmost corner and reduce it to

mimic Schwarz’s row reduction method. However, the leftmost corner may generate more than one fill in the profile. We can alter the definition to always rotate a ‘valid corner’, one that generates only one fill, and give preference to the leftmost valid corner. Though we can prove progress in this case, and that there exists at least one valid corner always - the rightmost corner, the leftmost corner algorithm does not work very well in practice. The fill generated and accessing columns in a complex fashion lead to poor performance. It does not have the nice properties of the rightmost corner algorithm, so blocking the leftmost corner method turns out to be difficult. We do not describe the leftmost corner algorithm in detail here. The idea of the leftmost corner algorithm was fully given shape in [Davis and Golub 2003].

5.3.1 Blocking in Mountainview Profile

Algorithm 5.2 Blocked rightmost mountain algorithm for mountainview bidiagonalization

- 1: Find all the corners from $1 \dots n$ and push them into a stack.
 - 2: **while** there are more corners **do**
 - 3: Pop the rightmost corner k and find the rightmost mountain
 - 4: **while** there are more corners in the rightmost mountain/rubble **do**
 - 5: Use the $(\text{throw}(k), k)$ as an anchor point and find a block in the mountain
 - 6: Divide the matrix into blocks based on the block.
 - 7: Find column rotations to reduce the corners in the block and apply them to the columns in this block
 - 8: **while** there are column rotations **do**
 - 9: Apply column rotations to C-block.
 - 10: Apply column rotations to D-block in a pipelined order and generate row rotations to chase the fill.
 - 11: Apply row rotations to R-block.
 - 12: Apply row rotations to F-block in a pipelined order and generate column rotations to chase the fill wherever possible.
 - 13: Readjust the four blocks to continue the chase
 - 14: **end while**
 - 15: **end while**
 - 16: **end while**
-

A blocked algorithm can be derived from the rightmost mountain theorem (5.3) and Theorem 5.2 to reduce the mountainview profile of R . The basic idea is to reduce the

mountainview profile one mountain at a time starting from the rightmost mountain. When the number of columns in the mountain becomes too large we can limit the block size and reduce one mountain in multiple steps.

We choose the rightmost corner entry as our anchor point for our blocked rightmost mountain algorithm and find a block around that anchor point. This block serves as our seed block to find all the entries that can be reduced in one step. Finding a block around the anchor point in the mountainview profile is similar to finding the block in the band reduction algorithm in Chapter 3. Once we have the seed block defined we can reduce the corners within the seed block, divide the rest of the matrix into blocks and pipeline the plane rotations similar to the algorithms in Sections 3.2 and 3.3, but there are some differences as we operate on a sparse R . When we chase the fill some fill is chasable and some fill is left behind, so the indexing is not as smooth as it was in the band case. Instead, we need to store the indices of columns and rows along with the plane rotations and apply it to the corresponding columns/rows.

Algorithm 5.2 reduces the mountainview profile in a blocked fashion by reducing the rightmost mountain in one or more steps depending on the block size and the size of the mountain. The blocked rightmost mountain algorithm establishes that blocking and pipelining is possible to reduce a sparse upper triangular R . However, it operates on mountainview profile of R and the size of the mountainview profile is quite large. The blocked algorithm must work on the skyline profile to be of practical use.

5.3.2 Blocking in Skyline Profile

Among the theorems we used to arrive at a blocked algorithm for the mountainview profile Theorem 5.3 and Lemma 6 apply to the skyline profile. But the mcolumn height property and Theorem 5.2 do not apply to the skyline profile. This means we still reduce one mountain (the rightmost one) to rubble in the skyline profile when we use the rightmost corner algorithm, but we cannot consider the entire mountain as one block

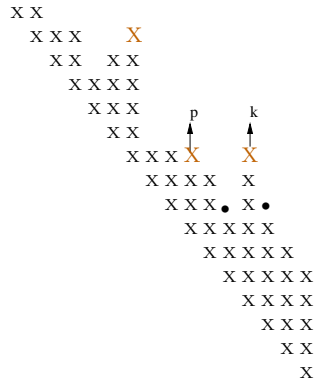


Figure 5-11. Problem with simple blocking in the skyline profile

as the fill need not be just in the rubble. The fill could be in one of the columns of the skyline that correspond to the mcolumns of the rightmost mountain.

Say the mcolumns of the rightmost mountain are $j \dots k$ where k is the rightmost corner. Given the skyline profile, we can find a column p such that $j < p < k$ and a row rotation of p and $p - 1$ generates fill in some column less than k , so we avoid columns that could generate such fill when we find the block. This is shown in Figure 5-11. We use a different skyline matrix in this figure than the one we have been using so far in this chapter to illustrate the trouble. There are two corners in the rightmost mountain. When we reduce the left corner in column p it results in more than one fill in columns $k - 1$ and $k + 1$ (marked as •). This leads us to the problem of finding the right seed block after finding the anchor point which is the rightmost corner. In the best case it could be the rightmost mountain and in the worst case it could be just the rightmost corner entry when no blocking is possible.

The block definitions are the same as in Section 3.2. If k is the rightmost corner then the first (or least) row in the S-block is $trow(k)$. This is also the $trow$ value for all the columns in the rightmost mountain. Let us say $fcoll$ and $lcoll$ define the first and last columns of the S-block and $lrow$ defines the last row in the S-block. The algorithm to find the S-block finds the values for $fcoll$, $lcoll$ and $lrow$. The basic idea is that once we select the rightmost corner entry $(trow(k), k)$ as our anchor point and our S-block with two entries

we inspect the columns to the left of it, rows below it and columns to the right of it and expand the S-block to get a good number of corners to be blocked and reduced together. However, a few restrictions must be placed on the S-block so that we always limit the fill to the rubble and we are able to partition the rest of the matrix correctly into blocks.

Algorithm 5.3 Finding the S-block for blocked skyline reduction

- 1: Set the initial values of $lrow$ to $trow(k - 1)$, $fcol$ to $k - 1$ and $lcol$ to k .
 - 2: Set the initial values for the first row of D-blk to $k - 1$ and first column of F-blk to $ecol(k - 1)$.
 - 3: Inspect columns to the left of S-block and include columns satisfying the conditions for S-block.
 - 4: Inspect row below the S-block and include row satisfying the conditions for S-block
 - 5: Inspect columns to the right of S-block and include columns satisfying the conditions for S-block.
-

1. The $trow$ of all the columns in the S-block should be in the S-block. In other words $lrow$ should be at least equal to the maximum of $trow$ values of all the columns in S-block.
2. The $fcol$ is limited by the least column in the mountain.
3. The S-block and the D-block cannot share a row.
4. The S-block and the F-block cannot share a column.

Condition one avoids the problem shown in Figure 5-11 as we do not do any row rotations in the S-block we can not create any fill in the mcolumns. Condition two is not strictly necessary. We can relax it to say $fcol$ is limited by the column $trow(k)$, but that greatly complicates the block structure, so we restrict the block to the mountain. Condition three and four ensure we can partition the matrix into blocks correctly. The algorithm to find the S-block checks for these conditions and increases the block size to utilize pipelining and caching for performance. The algorithm to find the S-block is listed in 5.3. Once we find the S-block the algorithm to reduce the skyline is similar to the blocked rightmost mountain algorithm for mountainview reduction algorithm (Algorithm 5.2). The algorithm to reduce the skyline is shown in Algorithm 5.4.

Algorithm 5.4 Blocked rightmost mountain algorithm for skyline bidiagonalization

```
1: Find all the corners from  $1 \dots n$  and push them into a stack.
2: while there are more corners do
3:   Pop the rightmost corner  $k$  and find the rightmost mountain columns
4:   while there are more corners in the rightmost mountain/rubble do
5:     Use the  $(\text{throw}(k), k)$  as an anchor point and find a block using Algorithm 5.3.
6:     Divide the matrix into blocks based on the block.
7:     Find column rotations to reduce the corners in the block and apply them to the
       columns in this block
8:     if there are new corners that got generated which are not within the boundaries
       of the seed block then
9:       push it in the stack for next iteration.
10:    end if
11:    while there are column rotations do
12:      Apply column rotations to C-block.
13:      Apply column rotations to D-block in a pipelined order and generate row
       rotations to chase the fill.
14:      Apply row rotations to R-block.
15:      Apply row rotations to F-block in a pipelined order and generate column
       rotations to chase the fill wherever possible.
16:      Readjust the four blocks to continue the chase
17:    end while
18:  end while
19: end while
```

We illustrate the blocked skyline algorithm with an example skyline R in Figure 5-12. The first S-block as determined by the algorithm to find the S-block is shown in Figure 5-12(a). All the corners are highlighted and the corner that is going to be reduced is numbered. Corner 1 results in a swap. Figure 5-12(b) shows the result of the swap with 0s in the swapped corner and the two new corners that were created as a result of the swap. The three corners 1, 2, 3 in Figure 5-12(a)-(b) are reduced in one step by the skyline algorithm. The corner numbered 3 is from the rubble but was included in the S-block to help blocking. Notice that the corresponding block in the mountainview profile was reduced with four rotations by the mountainview algorithm in Figure 5-10.

Figure 5-12(c) shows the newly created zeros and next S-block for the reduction. The third S-block is shown in Figure 5-12(d). Column 4 was not included as part of this S-block even though it is part of the same mountain because it would have created a

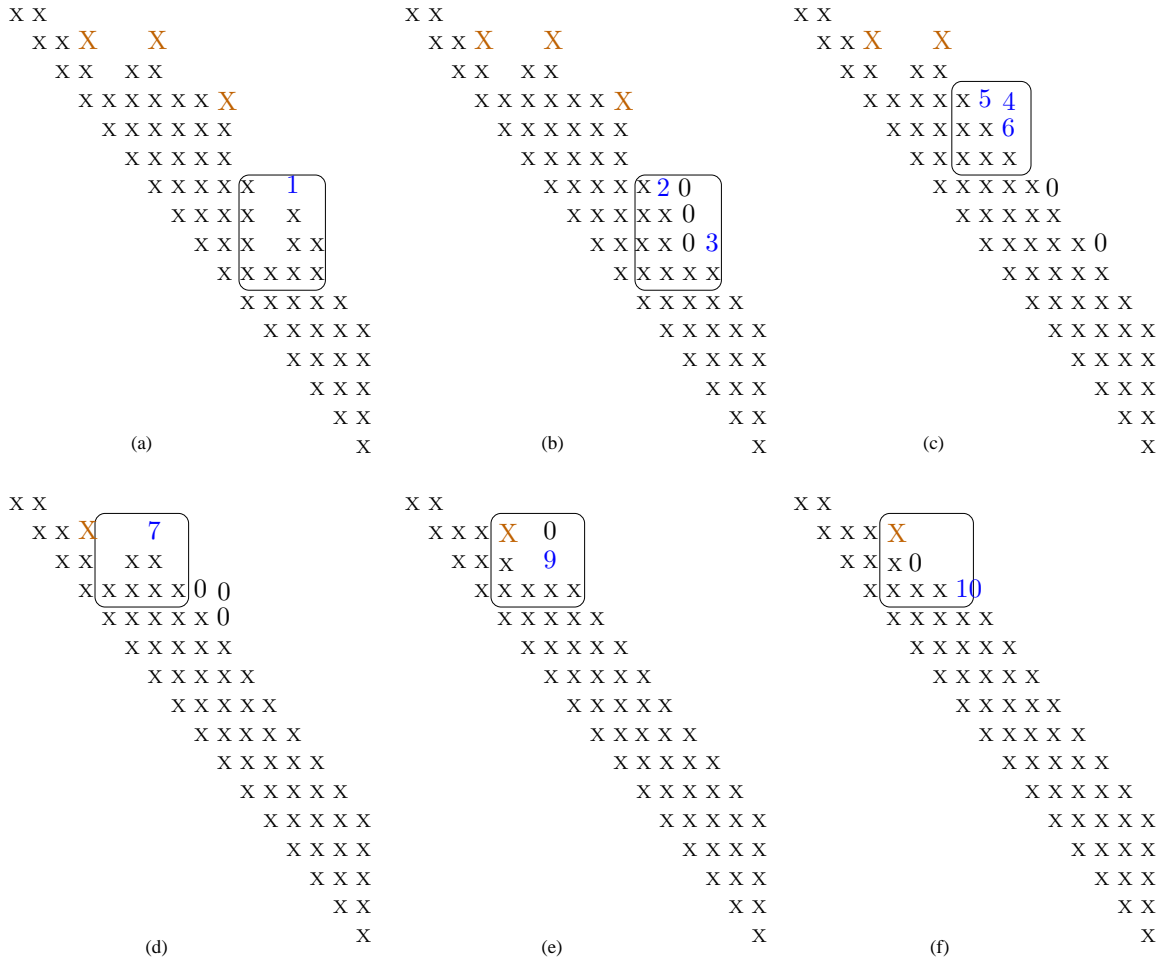


Figure 5-12. First three blocks and ten corners of the skyline reduction algorithm

fill in column 5. As corner 7 results in two swaps (corner 8 for the second swap is not shown) we get two new corners (shown in Figure 5-12(e)), but the corner in column 5 is not reduced in the current S-block as column 4 is not in the current S-block. This step is handled in Line 8 of Algorithm 5.4. We reduce corner 9 and then corner 10 as shown in Figure 5-12(e)-(f). Corners 7...10 are part of one step in the blocked reduction.

Note that even though we use the mcolumns to find the possible size of the S-block, we narrow the S-block in Algorithm 5.3, so the blocks are smaller than the blocks in the mountainview reduction algorithm. However, we operate on the skyline profile which is much more economical than the mountainview profile and as a result the number of operations is reduced.

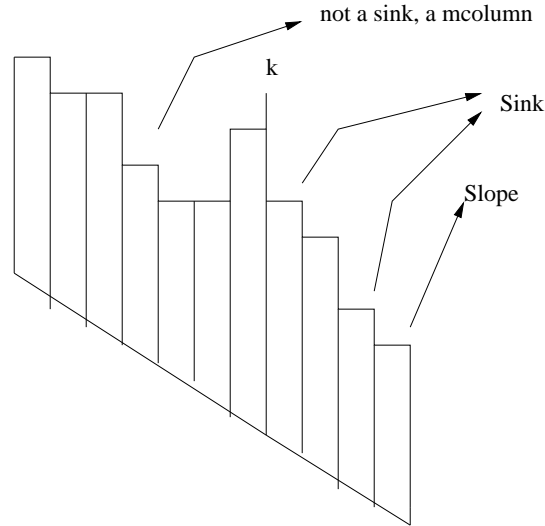


Figure 5-13. Classification of columns in the skyline profile

In the skyline algorithm the S-block can have more than one corner in it as in the Figure 5-12(b). This cannot happen in the mountainview profile. Each corner in the S-block causes at most one fill based on the way we found the S-block, so we can choose to reduce them in any order. Currently we eliminate the leftmost corner within the block so that we can reduce the operation count. We call this the *rightmost mountain left corner* algorithm. Given a band matrix this algorithm mimics the ‘row mode’ described in Section 3.5. Instead of choosing the leftmost corner all the time if we choose one rightmost corner after every c (say 32) leftmost corners we mimic the ‘block mode’ from Section 3.5.

There is an implementation issue we have not addressed in this section so far. The band reduction Algorithm 3.1 chases all the fill to end of the matrix. On the other hand the skyline reduction algorithm has to leave some of the fill that is not chasable in the profile which requires a dynamic data structure. A symbolic factorization for the skyline reduction that estimates the space requirements for each column is presented in Section 5.4

5.4 Symbolic Factorization

We use the definitions from Section 5.2.2 to arrive at the symbolic factorization algorithm. Notice that the definition are using the mountainview profile though they apply

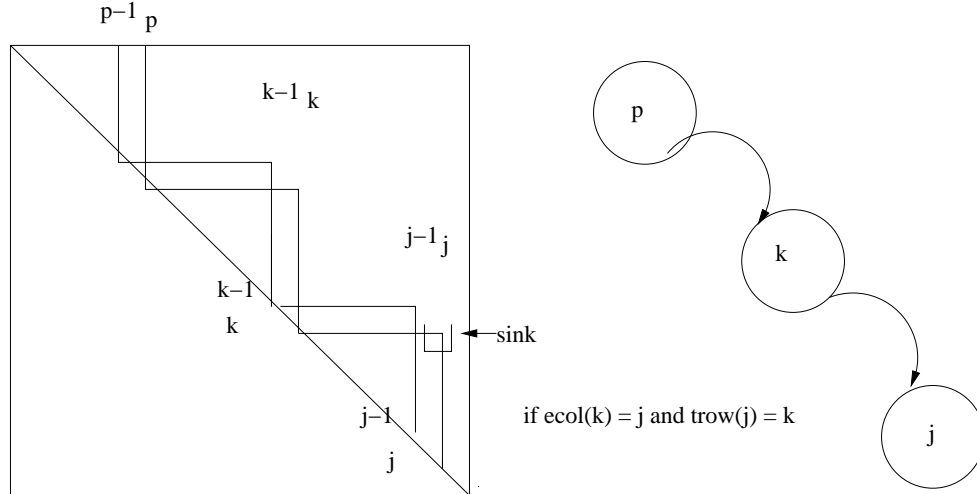


Figure 5-14. Symbolic factorization of the skyline profile

to both the profiles. Figure 5-13 illustrates the need for using the mountainview profile to define the mcolumn and sinks in the skyline profile. Column $k - 3$ in the figure is not a sink but an mcolumn. We classify it as such because column $k - 3$ cannot get filled when we use Algorithm 5.4 for reducing the skyline. It cannot be a sink. We can also write this as, column j in skyline profile is a sink if and only if $trow(j - 1) + 1 < trow(j)$ and $trow(k + 1 : n) \geq trow(k)$.

When we use this classification for the columns in skyline, from the definition of the corner, any fill in the slope is chasable, any fill in the sink is not chasable and a mcolumn cannot get a fill when we use the rightmost mountain algorithm for the skyline. We can then define a fill graph $\mathcal{F} = \{V, E\}$ where $V = \{1 \dots n\}$ are the vertices and edges E . An edge (i, j) is in E if and only if $ecol(i) = j$ and $trow(j) = i$. Figure 5-14 illustrates this graph with a simple example.

Theorem 5.4. *When column k is the rightmost corner, reducing the rightmost corner entry results in one fill in column j if and only if edge (k, j) exists in \mathcal{F} .*

Proof. We know that from the rightmost corner theorem (Theorem 5.1 there can be at most one fill in the rubble when you reduce the rightmost corner. We also know from the rubble column lemma (Lemma 4) and rubble row lemma (Lemma 5 that the fill, if it is

there, is in $ecol(k)$ which implies if the fill is in column j then $ecol(k) = j$. We also know that the fill is in row $k - 1$ (as the row rotation is between columns k and $k - 1$) which implies that $trow(j) = k$, so we can say from the definition of the edge (k, j) is present in \mathcal{F} .

On the other hand, if the edge $(k, trow(k))$ is present then we know $ecol(k) = j$ and $trow(j) = k$. We know $trow(j - 1) < trow(j)$ from the top row lemma which ensures $ecol(k - 1) = j - 1$. Then a row rotation between rows k and $k - 1$ has to lead to a fill in the entry $(k - 1, j)$. \square

Theorem 5.5. *If column k is the rightmost corner then reducing the corner and the resultant chase results in a fill that is not chasable in column q if and only if there is a sink-free path from k to q in \mathcal{F} and column q is a sink.*

Proof. If the fill in column q is not chasable then it is a sink by definition. If there was a fill in column q from the chase, from the fact that rightmost corner algorithm (Algorithm 5.1) does not handle the fill directly, but the fill itself will be the rightmost corner at every minor step in the chase (if it is chasable) we can apply Theorem 5.4 to every minor step so that a path exists from k to q in \mathcal{F} . If the path had any other sink column r then the fill in column r cannot be chased to q , a contradiction, so the path has to be sink-free.

On the other hand, if there is a sink free path from k to q in \mathcal{F} then for all vertices $k, p_1, p_2, \dots, p_n, q$ in the path it is true that $ecol(k) = p_1$, $ecol(p_1) = p_2$ until $ecol(p_n) = q$ and $trow(q) = p_n$, $trow(p_n) = p_{n-1}$ until $trow(p_1) = k$. Combining the $trow$ and $ecol$ values and the fact columns p_1, p_2, \dots, p_n have to be slopes (there are no sinks in between), the fill follows the path and will be the rightmost corner at each minor step and chasable to q creating a fill there. \square

Theorems 5.4 and 5.5 help us characterize the fill pattern. We define the *rightmost sink* as the largest column index for which a column is a sink. The sink helps us in two ways:

1. If we reduce the rightmost corner and follow the path using *trow* and *ecol* values and reach a sink we have an a fill that is not chasable and the column height has to grow.
2. If the rightmost sink is s , then the columns from $s + 1 \dots n$ cannot grow as long as the sink remains the same.

A simple symbolic factorization algorithm is to simulate the entire factorization but without doing any numerical work. As it operates only on *trow* and *ecol* values this algorithm takes only as many as steps as the number of plane rotations which is considerably less than the actual floating point operations. However, given a band matrix, which does not need any additional space at all, the symbolic factorization would reduce the entire matrix symbolically to find that it does not require any space.

The rightmost sink condition gives us a good stopping criterion for the symbolic factorization. Note that the band matrix has no sinks and if we use it as a stopping criterion the symbolic factorization stops immediately after finding that there no sinks. Algorithm 5.5 modifies the simple symbolic factorization to use this stopping criterion.

Algorithm 5.5 Symbolic factorization for the rightmost corner method

- 1: Find all the corners from $1 \dots n$ and push them into a stack.
 - 2: Inspect all the columns from $1 \dots n$ and find the rightmost sink. (s_r)
 - 3: **while** there are more corners and there exists at least one sink **do**
 - 4: Pop the rightmost corner k
 - 5: Follow the path from column k in graph \mathcal{F} until a sink (s) or until no fill.
 - 6: Inspect columns $k - 1, k + 1$, for new corners and push the new corners to stack.
 - 7: Inspect columns $k, k + 1$, and $s, s + 1$ for sink status and update the rightmost sink if necessary.
 - 8: **if** rightmost sink (s_r) is no longer a sink **then**
 - 9: scan columns $s_r - 1 \dots 1$ until you find the next rightmost sink.
 - 10: **end if**
 - 11: **end while**
-

Note that Line 12 in Algorithm 5.5 to maintain the rightmost sink may appear costly, but from the way the sinks get updated we can show that the cost to find the rightmost sink is $O(n)$ and amortized cost to update the rightmost sink is $O(1)$. It is easy to see that the initial cost to find the rightmost sink is $O(n)$ as we check the *trow* values for all the

columns. Note that if the rightmost sink s_r is no longer a sink (say if it became a slope) we scan the columns right-to-left to find the new rightmost sink. If this is the only type of change to the sinks then we will move back to column 1 with another $O(n)$ cost.

However, as a result of the fill created by a rotation and the swaps new sinks get created. A sink created by a swap can not be the rightmost sink. If a new rightmost sink gets created as result of fill then it has to be in $s_r + 1$. We can then update the rightmost sink in $O(1)$ time. The move towards n will pay for the move back to 1 again when we do the right-to-left scan to find the next rightmost sink. We do not maintain a list of all the sinks. Instead, we test it using the *throw* data structure every time we need it as we access the data structure to traverse the fill graph.

Algorithm 5.5 modified the non-blocked rightmost corner method for the symbolic factorization. We can apply the same changes to the blocked algorithms as well to get a simple symbolic factorization algorithm for the blocked skyline reduction. Algorithm 5.6 computes a symbolic factorization for the blocked skyline reduction. Note that we have removed the chase from Algorithms 5.5 and 5.6 as following the path in the fill graph is equivalent to the chase. The fill from these two algorithms can be different as the blocked skyline reduction algorithm will reduce the leftmost corner within the seed block.

5.5 PIRO_SKY: Pipelined Plane Rotations for Skyline Reduction

PIRO_SKY is the library that implements the sparse bidiagonal reduction algorithms discussed in this chapter. PIRO_SKY stands for PIpelined plane ROtations of sparse SKYline matrices. PIRO_SKY has both a MATLAB and C interface. PIRO_SKY works for real sparse matrices, with native integer (32-bit and 64-bit) support, and double precision arithmetic. Unlike PIRO_BAND, PIRO_SKY expects C99 compatibility.

The MATLAB interface of PIRO_SKY finds the singular value decomposition of sparse matrices using our blocked algorithms. To compute the singular value decomposition we depend on `symrcm` [Cuthill and McKee 1969] or the implementations of the GPS algorithm [Gibbs et al. 1976] for finding a profile reduction ordering for the matrix $A^T A$

Algorithm 5.6 Symbolic factorization for the blocked rightmost mountain algorithm for skyline bidiagonalization

```
1: Find all the corners from  $1 \dots n$  and push them into a stack.
2: Inspect all the columns from  $1 \dots n$  and find the rightmost sink. ( $s_r$ )
3: while there are more corners and there exists at least one sink do
4:   Pop the rightmost corner  $k$  and find the rightmost mountain columns
5:   while there are more corners in the rightmost mountain/rubble do
6:     Use the  $(\text{throw}(k), k)$  as an anchor point and find a block using Algorithm 5.3.
7:     For each corner  $k_c$  in the S-block follow the path from  $k_c$  in graph  $\mathcal{F}$  values until a
       sink ( $s$ ) or until no fill.
8:     if there are new corners that got generated which are not within the boundaries
       of the seed block then
9:       push it in the stack for next iteration.
10:    end if
11:    Inspect columns  $k, k+1$ , and  $s, s+1$  for sink status and update the rightmost sink
       if necessary.
12:    if rightmost sink ( $s_r$ ) is no longer a sink then
13:      scan columns  $s_r - 1 \dots 1$  until you find the next rightmost sink.
14:    end if
15:  end while
16: end while
```

and then use that for the column ordering of A . We depend on SPQR [Davis 2009a,b] for computing the QR factorization. PIRO_SKY then reduces the sparse upper triangular R to bidiagonal form. We can then use LAPACK to reduce the bidiagonal matrix to diagonal form [Golub and Reinsch 1970; Demmel and Kahan 1990; Deift et al. 1991]. We provide one interface `piro_sky_svd` which does all the above steps to compute the singular value decomposition. We also provide two custom interfaces for computing the symbolic factorization of the sparse upper triangular R (`piro_sky_symbolic`) and for computing the bidiagonal reduction itself (`piro_sky_reduce`).

`piro_sky_reduce` can operate in two modes. In the static mode, it can compute the symbolic factorization, allocate the required amount of memory and do the reduction in place. In the dynamic mode, it can allocate the space for the skyline matrix dynamically and compute the reduction. The dynamic mode is useful when the space required by the factorization (as computed by the symbolic factorization) is unavailable. PIRO_SKY manages the memory dynamically in the later case and reallocates memory as desired.

Table 5-1. MATLAB interface to PIRO_SKY

MATLAB function	Description of the function
<code>piro_sky_reduce</code>	Bidiagonal reduction routine for sparse matrices
<code>piro_sky_svd</code>	Computes the SVD of a sparse matrix
<code>piro_sky_symbolic</code>	Symbolic factorization for the bidiagonal reduction

Table 5-2. C interface to PIRO_SKY

C function	Description of the function
<code>piro_sky_set_common:</code>	Initializes PIRO_SKY with default values.
<code>piro_sky_allocate_symbolic:</code>	Allocates the data structure for the symbolic factorization.
<code>piro_sky_symbolic:</code>	Computes the symbolic factorization for a given sparse matrix.
<code>piro_sky_allocate_skyline:</code>	Allocates the skyline matrix based on the counts from symbolic factorization.
<code>piro_sky_sparse_to_skyline:</code>	Converts a compressed column format sparse matrix to a skyline matrix.
<code>piro_sky_reduce:</code>	Computes the bidiagonal reduction of the skyline matrix.
<code>piro_sky_free_skyline:</code>	Frees space used by the skyline data structure.
<code>piro_sky_free_symbolic:</code>	Frees space used by the symbolic data structure.

The reduction also accepts an approximate bandwidth and can reduce the skyline matrix to a band matrix. It then converts the skyline to a band data structure and uses the band reduction algorithm (PIRO_BAND) to compute the bidiagonals.

The symbolic factorization in PIRO_SKY computes the column counts required for sparse skyline matrices. It also reports the total number of plane rotations that would be required to do the reduction and the number of swaps that would take place. The swaps which look cheaper because they do not do any floating point work are actually expensive. When two columns are not adjacent to each other in memory the swap cannot be done in place and we may not be able to fit one column in the place of another. This could happen when we skip symbolic factorization and operate in the dynamic mode. PIRO_SKY dynamically reallocates space to continue with the reduction then.

The MATLAB interfaces for PIRO_SKY are summarized in Table 5-1. The C interface to PIRO_SKY provides a set of functions. The functions and a brief description of them are shown in Table 5-2. The C functions are listed in the exact order as they

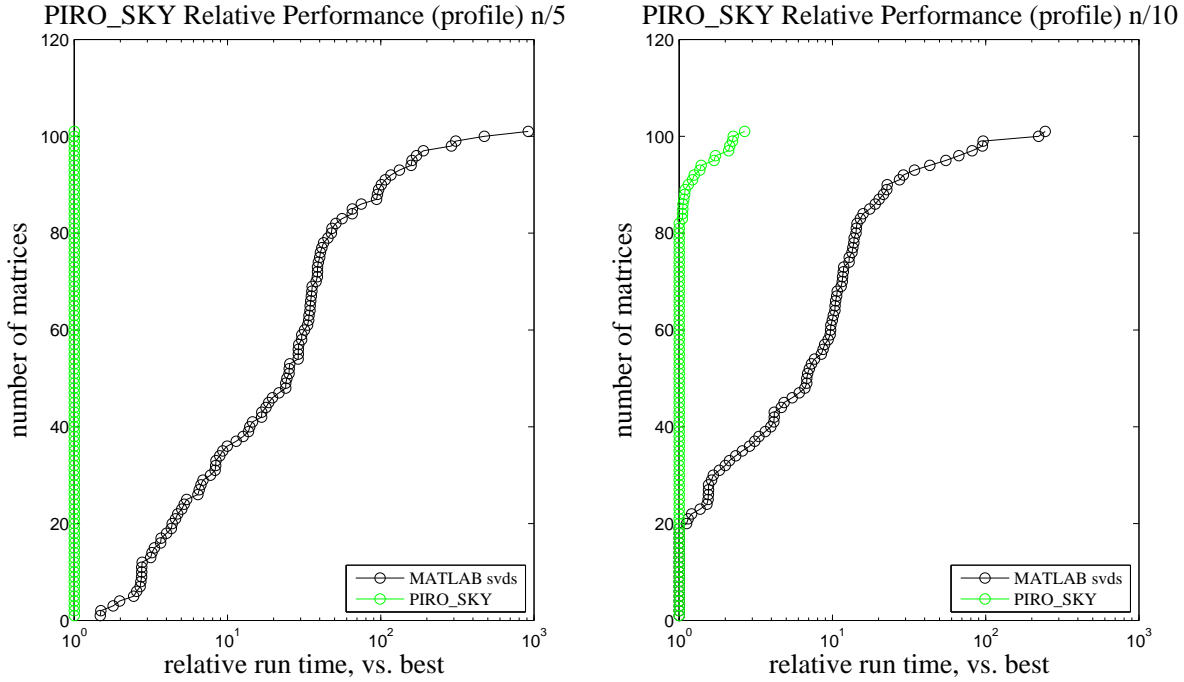


Figure 5-15. Performance of PIRO_SKY vs MATLAB svds

will be used in an example. The users can set an array of parameters using the data structures in the C interface to fine tune the performance of PIRO_SKY including size of the work space, block size, static/dynamic memory allocation, elbow room for columns, approximate bandwidth and functions used to allocate and free memory. PIRO_SKY is not publicly available yet. We plan to make it publicly available as an ACM Transactions of Mathematical Software collected algorithm.

5.6 Performance Results

We compare our PIRO_SKY implementation to `svds` in MATLAB. MATLAB uses ARPACK [Lehoucq et al. 1997; Lehoucq and Sorensen 1996] to compute the singular values of a sparse matrix. As MATLAB `svds` is not designed compute all the singular values of a sparse matrix and our algorithm is designed to compute all the singular values we do not compare them in the case of computing all the singular values. Instead, given a sparse matrix A of size n -by- n , we compute $\frac{n}{k}$ singular values for various values of k using

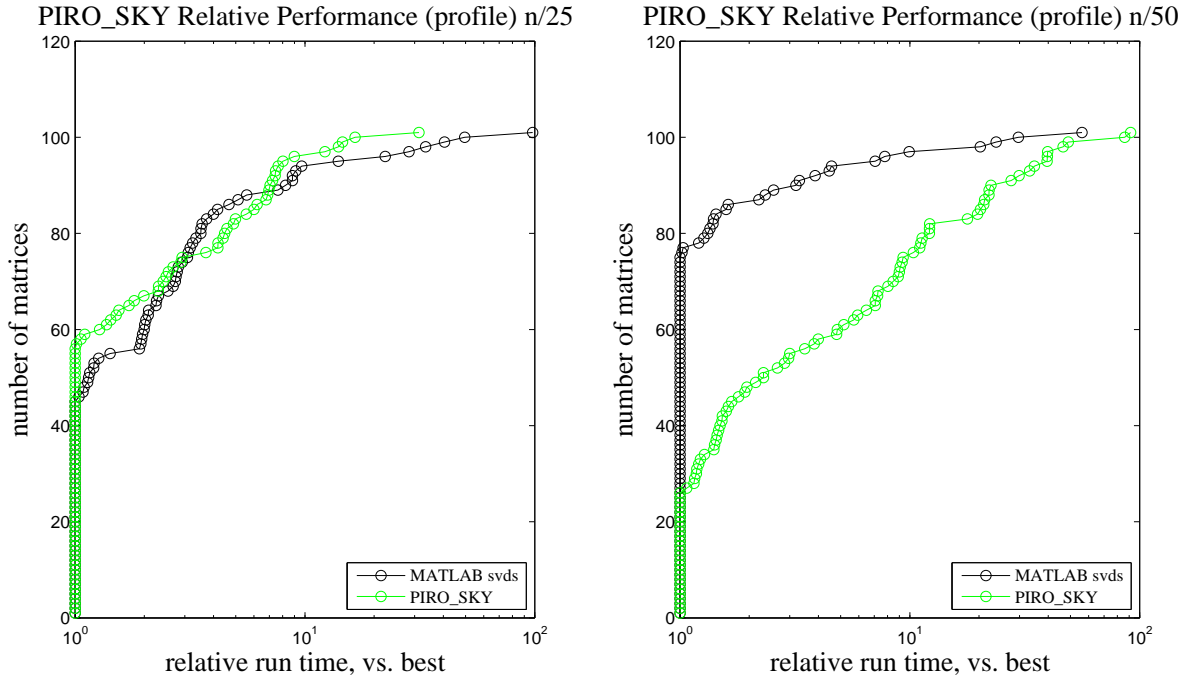


Figure 5-16. Performance of PIRO_SKY vs MATLAB svds

MATLAB `svds` and compute *all* the singular values using our method and compare the results. This should give a better idea of when we can use the iterative method and when we can switch to our direct method and compute all the singular values instead.

All the tests were done with double precision arithmetic with real matrices from the UF sparse matrix collection [Davis 2009c]. We used real matrices with $n < 5000$. The performance measurements were done on a machine with 8 dual core 2.2 GHz AMD Opteron processors and 64GB of main memory. PIRO_BAND was compiled with `gcc` 4.1.1 with the options `-O3`. We used `svds` in MATLAB 7.6.0.324 (R2008a).

PIRO_SKY was configured to always use the symbolic factorization, allocate the space for the skyline and operate on a static data structure. Furthermore, we allowed 50% more space than the original number of non-zeros in R (not in the skyline of R) to get an approximate bandwidth in R below which most entries lie. We then use the bandwidth in the symbolic and numerical reduction to reduce the matrix to a band matrix. We then use PIRO_BAND library to reduce the band matrix to bidiagonal form. We use MATLAB's

sparse `qr` and `symrcm` for the QR factorization and profile reduction steps. We reduced the bidiagonal matrix to diagonal form using LAPACK. PIRO_SKY's time to compute the SVD includes the time to compute all these steps.

The results when we compute $\frac{n}{5}$ and $\frac{n}{10}$ singular values are presented in Figure 5-15. We use a performance profile to compare the two methods. In both cases PIRO_SKY is faster than MATLAB's `svd` for all the matrices and computes all the singular values. Figure 5-16 shows the results for computing $\frac{n}{25}$ and $\frac{n}{50}$ singular values. When $\frac{n}{25}$ singular values are computed by the `svds` the performance of both algorithms are competitive except in a few matrices where PIRO_SKY does not perform very well. When we compute $\frac{n}{50}$ singular values `svds` performs better in more than half the matrices and the performance is similar in the other half of the matrices. To summarize, PIRO_SKY computes *all* the singular values in the time it takes to compute just 4% of singular values using `svds` and does much better when more singular values are required.

CHAPTER 6 CONCLUSION

The singular value decomposition is a problem that is used in wide range of applications. We studied this problem for two specific class of matrices : a band matrix and a sparse matrix. Our major contributions to these problems are:

- We developed a blocked algorithm for band reduction that generates no more fill than any scalar band reduction method using pipelining. The work space required for our algorithm is much smaller (equal to the size of the block) than the work space required by competing methods ($O(n)$ to $O(nnz)$). Our blocked methods perform slightly more work than the Schwarz's row reduction method but less than both LAPACK and SBR band reduction algorithms. When the plane rotations are not accumulated, our algorithm is about 2 times faster than SBR and 3.5 times faster than LAPACK. Our blocked algorithm manages to balance all the three requirements, cache utilization, number of floating point operations and work space requirement.
- We developed an algorithm to accumulate the plane rotations while utilizing the non zero structure of U and V for our blocked algorithm. Our algorithm reduces the number of plane rotations by half, hence the number of flops in accumulations by half, as compared to LAPACK's accumulation algorithm. When the plane rotations are accumulated our speed up is about 2x and 2.5x against LAPACK and SBR.
- We developed a library for band reduction that supports all the required data types with both C and MATLAB interfaces. PIRO_BAND will be made available as a collected ACM TOMS algorithm.
- We developed the theory for sparse skyline bidiagonal reduction, and both blocked and non-blocked algorithms for the problem. We also developed the theory and the algorithm for symbolic factorization for the bidiagonal reduction method. Our bidiagonal reduction method uses efficient sparse data structures and computes all the singular values of a sparse matrix in the same time it takes for the iterative method to compute just 4% of the singular values. It performs far better when more singular values are required. No prior algorithm exists for efficiently computing *all* the singular values of a sparse matrix.
- We developed a library for bidiagonal reduction of sparse matrices and finding the sparse singular value decomposition. PIRO_SKY supports both C and MATLAB interfaces. PIRO_SKY will be made available as a collected ACM TOMS algorithm. Our library for sparse bidiagonal reduction still does not support complex matrices. Complex support will be part of the library when it is made publicly available.

It is not expensive to compute the symbolic factorization algorithms described in Section 5.4, but a theoretical question still remains open. Given the fill graph \mathcal{F} it will be interesting to compute the symbolic factorization without simulating the rotations of numerical reduction. However, for all practical purposes our symbolic factorization algorithm works well.

The techniques to adopt our blocked reduction methods to multicore architectures is another interesting problem that still remains to be studied. The problem has been studied in the context of reducing a dense matrix to band form [Ltaief et al. 2009]. The band reduction algorithm could couple the blocking with the technique of moving the blocks a fixed distance before reducing the next block, as it was done in [Kaufman 2000] for single entries. However, the synchronization between the blocks have to be worked out carefully for adopting the blocked algorithm to multicore architectures. Band reduction in parallel architectures has been studied for sometime though [Lang 1993, 1996]. Efficient bidiagonal reduction of the sparse skyline matrix in multicore architectures still remains an open problem.

REFERENCES

- ANDA, A. A. AND PARK, H. 1994. Fast plane rotations with dynamic scaling. *SIAM J. Matrix Anal. Appl.* 15, 1, 162–174.
- ANDERSON, E., BAI, Z., BISCHOF, C., BLACKFORD, S., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., AND SORENSEN, D. 1999. *LAPACK Users' Guide*, Third ed. Society for Industrial and Applied Mathematics, Philadelphia, PA.
- BARLOW, J. L., BOSNER, N., AND DRMAC, Z. 2005. A new stable bidiagonal reduction algorithm. *Linear Algebra Appl.* 397, 35–84.
- BERRY, M. W. 1992. Large scale sparse singular value decompositions. *Inter. J. of Supercomputer Appl.* 6, 1, 13–49.
- BERRY, M. W., DUMAIS, S. T., AND O.BRIEN, G. W. 1994. Using linear algebra for intelligent information retrieval. *SIAM Rev.* 37, 4, 573–595.
- BILLSUS, D. AND PAZZANI, M. J. 1998. Learning collaborative information filters. In *ICML '98: Proc. Fifteenth Inter. Conf. Mach. Learn.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 46–54.
- BINDEL, D., DEMMEL, J., KAHAN, W., AND MARQUES, O. 2002. On computing Givens rotations reliably and efficiently. *ACM Trans. Math. Soft.* 28, 4, 206–238.
- BISCHOF, C. H., LANG, B., AND SUN, X. 2000a. Algorithm 807: The SBR toolbox—software for successive band reduction. *ACM Trans. Math. Softw.* 26, 4, 602–616.
- BISCHOF, C. H., LANG, B., AND SUN, X. 2000b. A framework for symmetric band reduction. *ACM Trans. Math. Soft.* 26, 4, 581–601.
- BLACKFORD, L. S., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., POZO, R., REMINGTON, K., AND WHALEY, R. C. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* 28, 2, 135–151.
- BOSNER, N. AND BARLOW, J. L. 2007. Block and parallel versions of one-sided bidiagonalization. *SIAM J. Matrix Anal. Appl.* 29, 3, 927–953.
- CHAN, T. F. 1982. An improved algorithm for computing the singular value decomposition. *ACM Trans. Math. Soft.* 8, 1, 72–83.
- CUTHILL, E. AND MCKEE, J. 1969. Reducing the bandwidth of sparse symmetric matrices. In *Proc. 1969 24th Nat. Conf.* ACM, New York, NY, USA, 157–172.

- DAVIS, T. A. 2009a. Algorithm 8xx: SuiteSparseQR, a multifrontal multithreaded sparse QR factorization package. Submitted to *ACM Trans. Math. Softw.*, Retrieved October 7, 2009 from <http://www.cise.ufl.edu/~davis/>.
- DAVIS, T. A. 2009b. Multifrontal multithreaded rank-revealing sparse QR factorization. Submitted to *ACM Trans. Math. Softw.*, Retrieved October 7, 2009 from <http://www.cise.ufl.edu/~davis/>.
- DAVIS, T. A. 2009c. The University of Florida sparse matrix collection. Submitted to *ACM Trans. Math. Softw.*, Retrieved October 7, 2009 from <http://www.cise.ufl.edu/~davis/>.
- DAVIS, T. A., GILBERT, J. R., LARIMORE, S. I., AND NG, E. G. 2004a. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.* 30, 3, 377–380.
- DAVIS, T. A., GILBERT, J. R., LARIMORE, S. I., AND NG, E. G. 2004b. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.* 30, 3, 353–376.
- DAVIS, T. A. AND GOLUB, G. H. 2003. Personal communications.
- DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. 1990. Indexing by latent semantic analysis. *J. Amer. Soc. Inf. Sci.* 41, 391–407.
- DEIFT, P., DEMMEL, J., LI, C., AND TOMEI, C. 1991. The bidiagonal singular value decomposition and hamiltonian mechanics. *SIAM J. Numer. Anal.* 28, 1463–1516.
- DEMMEL, J. AND KAHAN, W. 1990. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.* 11, 873–912.
- DONGARRA, J. J., DU CROZ, J., HAMMARLING, S., AND DUFF, I. S. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.* 16, 1, 1–17.
- DRINEAS, P., KANNAN, R., AND MAHONEY, M. W. 2004. Fast monte carlo algorithms for matrices II: Computing a low-rank approximation to a matrix. *SIAM J. Comp.* 36, 2006.
- GENTLEMAN, W. M. 1973. Least squares computations by Givens transformations without square roots. *IMA J. Appl. Math.* 12, 329–336.
- GIBBS, N. E., POOLE, WILLIAM G., J., AND STOCKMEYER, P. K. 1976. An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM J. Numer. Anal.* 13, 2, 236–250.
- GIVENS, W. 1958. Computation of plane unitary rotations transforming general matrix to triangular form. *J. SIAM* 6, 1, 26–50.

- GOLDBERG, K., ROEDER, T., GUPTA, D., AND PERKINS, C. 2001. Eigentaste: A constant time collaborative filtering algorithm. *Info. Retrieval* 4, 2, 135–151.
- GOLUB, G. H. AND KAHAN, W. 1965. Calculating the singular values and pseudo-inverse of a matrix. *SIAM J. Numer. Anal.* 2, 2, 205–224.
- GOLUB, G. H. AND REINSCH, C. 1970. Singular value decomposition and least square solutions. *Numer. Math.* 14, 5, 403–420.
- GOLUB, G. H. AND VAN LOAN, C. F. 1996. *Matrix Computations*, 3rd ed. The Johns Hopkins University Press.
- HAGER, W. W. 2002. Minimizing the profile of a symmetric matrix. *SIAM J. Sci. Comput.* 23, 5, 1799–1816.
- HAMMARLING, S. 1974. A note on modifications to the Givens plane rotation. *IMA J. Appl. Math.* 13, 215–218.
- KAUFMAN, L. 1984. Banded eigenvalue solvers on vector machines. *ACM Trans. Math. Softw.* 10, 1, 73–85.
- KAUFMAN, L. 2000. Band reduction algorithms revisited. *ACM Trans. Math. Softw.* 26, 4, 551–567.
- KOLDA, T. G. AND O’LEARY, D. P. 1998. A semidiscrete matrix decomposition for latent semantic indexing in information retrieval. *ACM Trans. on Inf. Syst.* 16, 322–346.
- KOREN, Y. 2008. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *KDD ’08: Proc. 14th ACM SIGKDD Int. Conf. Knowledge Discovery Data Min.* ACM, New York, NY, USA, 426–434.
- LANG, B. 1993. A parallel algorithm for reducing symmetric banded matrices to tridiagonal form. *SIAM J. Sci. Comput.* 14, 6, 1320–1338.
- LANG, B. 1996. Parallel reduction of banded matrices to bidiagonal form. *Parallel Comput.* 22, 1, 1–18.
- LARSEN, R. M. 1998. Lanczos bidiagonalization with partial reorthogonalization. Retrieved October 7, 2009 from <http://soi.stanford.edu/~rmunk/PROPACK/>.
- LEHOUCQ, R. AND SORENSEN, D. C. 1996. Deflation techniques for an implicitly re-started arnoldi iteration. *SIAM J. Matrix Anal. Appl* 17, 789–821.
- LEHOUCQ, R. B., SORENSEN, D. C., AND YANG, C. 1997. ARPACK users’ guide: Solution of large scale eigenvalue problems with implicitly restarted Arnoldi methods. Retrieved October 7, 2009 from <http://www.caam.rice.edu/software/ARPACK/UG/ug.html>.

- LTAIEF, H., KURZAK, J., AND DONGARRA, J. 2009. Parallel band two-sided matrix bidiagonalization for multicore architectures. *IEEE Trans. Parallel and Dist. Sys.* (to appear), Retrieved October 7, 2009 from <http://icl.cs.utk.edu/~ltaief/>.
- MURATA, K. AND HORIKOSHI, K. 1975. A new method for the tridiagonalization of the symmetric band matrix. *Infor. Proc. Japan* 15, 108–112.
- PATEREK, A. 2007. Improving regularized singular value decomposition for collaborative filtering. In *Proc. KDD Cup and Workshop*.
- PRYOR, M. H. 1998. The Effects of Singular Value Decomposition on Collaborative Filtering. Tech. Rep. PCS-TR98-338, Dartmouth College, Computer Science, Hanover, NH. June.
- RAJAMANICKAM, S. AND DAVIS, T. A. 2009a. Blocked band reduction for symmetric and unsymmetric matrices. To be submitted to *ACM Trans. Math. Softw.*, Retrieved October 7, 2009 from <http://www.cise.ufl.edu/~srajaman/>.
- RAJAMANICKAM, S. AND DAVIS, T. A. 2009b. PIRO_BAND: Pipelined plane rotations for blocked band reduction. To be submitted to *ACM Trans. Math. Softw.*, Retrieved October 7, 2009 from <http://www.cise.ufl.edu/~srajaman/>.
- RAJAMANICKAM, S. AND DAVIS, T. A. 2009c. PIRO_BAND user guide. Retrieved October 7, 2009 from <http://www.cise.ufl.edu/~srajaman/>.
- RALHA, R. 2003. One-sided reduction to bidiagonal form. *Linear Algebra Appl.* 358, 1-3, 219–238.
- REID, J. K. AND SCOTT, J. A. 1998. Ordering symmetric sparse matrices for small profile and wavefront. Tech. Rep. RAL-TR-98-016, Rutherford Appleton Laboratory.
- REID, J. K. AND SCOTT, J. A. 2002. Implementing Hager’s exchange methods for matrix profile reduction. *ACM Trans. Math. Softw.* 28, 4, 377–391.
- RUTISHAUSER, H. 1963. On Jacobi rotation patterns. *Proc. of Symp. in Appl. Math.* 15, 219–239.
- SARWAR, B., KARYPIS, G., KONSTAN, J., AND RIEDL, J. 2002. Incremental singular value decomposition algorithms for highly scalable recommender systems. In *Fifth Int. Conf. Comput. Inf. Sci.* 27–28.
- SCHWARZ, H. R. 1963. Algorithm 183: Reduction of a symmetric bandmatrix to triple diagonal form. *Commun. ACM* 6, 6, 315–316.
- SCHWARZ, H. R. 1968. Tridiagonalization of a symmetric band matrix. *Numer. Math.* 12, 4, 231–241.

- SMITH, B. T., BOYLE, J. M., DONGARRA, J., GARBOW, B. S., IKEBE, Y., KLEMA, V. C., AND MOLER, C. B. 1976. *Matrix Eigensystem Routines - EISPACK Guide, Second Edition*. Lecture Notes in Computer Science, vol. 6. Springer.
- STEWART, G. W. 1976. The economical storage of plane rotations. *Numer. Math.* 25, 137–138.
- SUN, J., XIE, Y., ZHANG, H., AND FALOUTSOS, C. 2008. Less is more: Sparse graph mining with compact matrix decomposition. *Stat. Anal. Data Min.* 1, 1, 6–22.
- TREFETHEN, L. N. AND BAU III, D. 1997. *Numerical Linear Algebra*, 1st ed. Society for Industrial and Applied Mathematics.
- VAN LOAN, C. AND BISCHOF, C. 1987. The WY representation for products of householder matrices. *SIAM J. Sci. Stat. Comput.* 8, 1.
- WALL, M., RECHTSTEINER, A., AND ROCHA, L. 2003. Singular value decomposition and principal component analysis. In *A practical Approach to Microarray Data Analysis*, D. P. Berrar, W. Dubitzky, and M. Granzow, Eds. 91–109.

BIOGRAPHICAL SKETCH

Siva Rajamanickam's research interests are in sparse matrix algorithms. Specifically, he works on direct methods for sparse eigen value and singular value decompositions. He has also worked on band reduction methods and ordering methods for sparse Cholesky and sparse LU factorizations. He is one of the authors of CCOLAMD (Constrained Column Approximate Minimum Degree Ordering) library which is part of the CHOLMOD (Cholesky Modification) package. He was also part of the development teams in Sun Microsystems and Tata Consultancy Services.