LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

A Stream Algorithm for the SVD

Technical Memo
MIT-LCS-TM-641
October 22, 2003

Volker Strumpen, Henry Hoffmann, and Anant Agarwal

{strumpen,hank,agarwal}@cag.lcs.mit.edu

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A Stream Algorithm for the SVD

Volker Strumpen, Henry Hoffmann, and Anant Agarwal
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{strumpen,hank,agarwal}@cag.lcs.mit.edu

October 22, 2003

Abstract

We present a stream algorithm for the Singular-Value Decomposition (SVD) of an $M \times N$ matrix A. Our algorithm trades speed of numerical convergence for parallelism, and derives from a one-sided, cyclic-by-rows Hestenes SVD. Experimental results show that we can create $O(M)$ parallelism, at the expense of increasing the computational work by less than a factor of about 2. Our algorithm qualifies as a stream algorithm in that it requires no more than a small, bounded amount of local storage per processor and its compute efficiency approaches an optimal 100 % asymptotically for large numbers of processors and appropriate problem sizes.

1 Background

We have designed *stream algorithms* as a particular class of parallel algorithms that offer a good match for the technological constraints of future single-chip microarchitectures. An introduction to our notion of stream algorithms can be found in [12]. In short, stream algorithms emphasize computational efficiency in space and time, and were developed in conjunction with a decoupled systolic architecture that we call *stream architecture*. Processing elements of our stream architecture are assumed to have a small yet fast amount of local storage, and mass memory is available on the periphery of a two-dimensional grid of processing elements with fast networks connecting next neighbors only.

We characterize stream algorithms by means of three key features. First, stream algorithms achieve an optimal 100 % compute efficiency asymptotically for large numbers of processors. Second, stream algorithms use no more than a small, bounded amount of storage on each processing element. Third, data are streamed through the compute fabric from and to peripheral memories. The number of these memories is asymptotically insignificant compared to the number of compute processors, so that the work performed by the memory modules represents an insignificant portion of the total amount of consumed energy.

The stream algorithms we presented in [12] tackle problems whose algorithmic solutions are independent of the data values and are determined by problem size, that is the amount of data only. These problems include a convolution, a matrix multiplication, a triangular solver, an LU and a QR factorization. In this paper, we tackle the data-dependent problem of computing singular values of a matrix. The data dependency manifests itself in an iterative algorithm whose termination depends on the explicit computation of a convergence criterion.

In the following, we review the singular-value decomposition. We discuss various algorithms and computational aspects, and develop the basic insights leading up to our stream algorithm. Our stream SVD is presented in Section 4. Finally, we shed some light on the convergence behavior by reporting experimental results in Section 5.

2 Singular-Value Decomposition

We consider the singular-value decomposition (SVD) of a dense $M \times N$ matrix A. The singular-value decomposition of A is

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

where U is an orthogonal[1] $M \times M$ matrix, and V is an orthogonal $N \times N$ matrix, that is $U^T U = I_M$ and $V^T V = I_N$, and the $M \times N$ matrix Σ is a diagonal matrix $\mathrm{diag}(\sigma_0, \ldots, \sigma_{N-1})$ on top of $M - N$ rows of zeros. The σ_i are the singular values of A. Matrix U contains N left singular vectors, and matrix V consists of N right singular vectors. The singular values and singular (column) vectors of U and V form the relations

$$Av_i = \sigma_i u_i \qquad \text{and} \qquad A^T u_i = \sigma_i v_i.$$

The SVD is closely related to the eigen-decomposition of $M \times M$ matrix AA^T and $N \times N$ matrix $A^T A$, because

$$A^T A v_i = \sigma_i^2 v_i \qquad \text{and} \qquad AA^T u_i = \sigma_i^2 u_i.$$

In fact, the σ_i are the square roots of the eigenvalues and the v_i are the eigenvectors of $A^T A$. Furthermore, the σ_i are the square roots of the eigenvalues and the u_i are the eigenvectors of AA^T. For $M > N$, at least $M - N$ singular values will be zero. If A has rank $r < N$, r of the singular values will be non-zero, because U and V are rotations.

Most algorithms for the SVD are based on diagonalizing rotations. Rotations are the simplest form of orthogonal transformations that preserve angles and lengths, and also the eigenvalues and eigenvectors as well as singular values and singular vectors of a matrix. We apply a sequence of rotations to matrix $A = A^{(0)}$ to produce sequence $A^{(k)}$ which approaches diagonal matrix Σ, that is $\lim_{k\to\infty} A^{(k)} = \Sigma$. Using the Frobenius norm $||A|| = \sqrt{\sum_{i,j} a_{ij}^2}$, we can prove convergence properties. The Frobenius norm also allows us to derive termination criteria based on measuring the magnitude of the off-diagonal elements of matrix $A^{(k)}$, either in terms of absolute values or relative to the machine epsilon or to the norm of the diagonal elements. The Frobenius norm of the $A^{(k)}$ remains unchanged under orthogonal rotations.

[1] We discuss the SVD as well as the stream SVD in terms of real-valued numbers. All statements and results generalize directly to the complex domain.

Convergence of the sequence $A^{(k)}$ occurs due to a decreasing Frobenius norm of the off-diagonal elements.

Since computing the SVD of a matrix is an inherently iterative process, assessing the efficiency requires a more elaborate analysis than for data-oblivious stream algorithms [12]. The number of iterations enters the picture as an additional degree of freedom, and depends on the choice and order in which we compute and apply diagonalizing rotations. Some choices and orders expose parallelism while others do not. Due to the data-dependency, we can merely identify methods that use a reasonably small number of rotations for matrices of interest, rather than the minimum number for all matrices. Thus, any design of a stream algorithm will involve the trade-off between the number of rotations performed until convergence and the efficiency of processing. To qualify as a stream algorithm, our goal is to find a type of rotation that achieves 100 % compute efficiency even at the cost of a (negligible) deterioration of the convergence behavior.

2.1 Jacobi Rotations

In the context of the SVD we are concerned with Jacobi rotations [13]. The original ***Jacobi rotation*** has been used in the 19^{th} century by Jacobi to solve systems of linear equations derived with the least squares method by hand. This rotation is also called *Givens rotation*, or can be interpreted as a 2-by-2 *Schur decomposition* [7]. We introduce the Jacobi rotation $J(i, j, \theta)$ for an index pair (i, j) and a rotation angle θ as a square matrix that is equal to the identity matrix I plus four additional entries at the intersections of rows and columns i and j:

$$J(i,j,\theta) \;=\; \begin{array}{cc} & \begin{array}{ccccccc} & & i & & j & & \end{array} \\ \begin{array}{c} \\ \\ i \\ \\ j \\ \\ \\ \end{array} & \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & s & & \\ & & & \ddots & & & \\ & & -s & & c & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{pmatrix} \end{array},$$

where $c = \cos\theta$ and $s = \sin\theta$. It is straightforward to verify that $J(i, j, \theta)^T J(i, j, \theta) = I$ for all angles θ. Thus, the Jacobi rotation is an orthogonal matrix. In the classical Jacobi rotation [13], the rotation angle is chosen such that the off-diagonal elements $a_{ij} = a_{ji}$ of a symmetric matrix A become zero when applying J to A as an orthogonal similarity transformation $\hat{A} = J^T A J$, that is by premultiplying and postmultiplying A with J^T and J. This similarity transformation is a ***plane rotation***, because due to J's structure, only rows and columns i and j of A are modified. Considering a 2-by-2 example, we may construct J such that it annihilates the off-diagonal elements w:

$$\begin{pmatrix} \hat{x} & 0 \\ 0 & \hat{y} \end{pmatrix} \;=\; \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} x & w \\ w & y \end{pmatrix} \begin{pmatrix} c & s \\ -s & c \end{pmatrix}. \tag{2}$$

There are various ways for computing the sine s and cosine c of the rotation angle θ without computing θ explicitly or evaluating any trigonometric functions. To solve the quadratic equation

$$(c^2 - s^2)w + cs(x - y) \;=\; 0, \tag{3}$$

Rutishauser [18] proposed the following formulas, which are in use since because they diminish the accumulation of rounding errors:

$$\alpha \;=\; \frac{y-x}{2w}, \qquad \tau \;=\; \frac{\operatorname{sign}(\alpha)}{|\alpha| + \sqrt{1+\alpha^2}}, \tag{4}$$

$$\text{then} \qquad c \;=\; \frac{1}{\sqrt{1+\tau^2}}, \qquad s \;=\; \tau c.$$

This solution results from picking the smaller angle $|\theta| \leq \pi/4$.[2]

Even if A is not symmetric, we may still annihilate both off-diagonal elements. This requires two rotations, however. Forsythe and Henrici [5, 2] proposed a rotation that diagonalizes an arbitrary matrix A by means of two transformations: (1) a symmetrizing rotation, and (2) a diagonalizing rotation. The symmetrizing rotation is a one-sided orthogonal transformation:

$$\begin{pmatrix} \tilde{u} & w \\ w & \tilde{v} \end{pmatrix} \;=\; \begin{pmatrix} c_1 & s_1 \\ -s_1 & c_1 \end{pmatrix}^T \begin{pmatrix} u & x \\ y & v \end{pmatrix},$$

which is followed by a two-sided orthogonal transformation for diagonalization:

$$\begin{pmatrix} \hat{u} & 0 \\ 0 & \hat{v} \end{pmatrix} \;=\; \begin{pmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{pmatrix}^T \begin{pmatrix} \tilde{u} & w \\ w & \tilde{v} \end{pmatrix} \begin{pmatrix} c_2 & s_2 \\ -s_2 & c_2 \end{pmatrix}.$$

With a little arithmetic, we find the following solution for c_1 and s_1. Let

$$\tau \;=\; \frac{u+v}{x-y},$$

$$\text{then} \qquad s_1 \;=\; \frac{\operatorname{sign}(\tau)}{\sqrt{1+\tau^2}}, \qquad c_1 \;=\; \tau s_1.$$

The computation of c_2 and s_2 proceeds according to Equation 4.

Hestenes [11] discovered a connection between the orthogonalization of two vectors and the Jacobi rotation for annihilating matrix elements. Given two (column) vectors u and v we may arrive at orthogonal vectors $\hat{u}$ and $\hat{v}$ by means of the orthogonal transformation:

$$\begin{pmatrix} \hat{u}^T \\ \hat{v}^T \end{pmatrix} \;=\; \begin{pmatrix} c & s \\ -s & c \end{pmatrix}^T \begin{pmatrix} u^T \\ v^T \end{pmatrix}, \qquad \text{such that} \quad \hat{u}^T \hat{v} = 0. \tag{5}$$

[2]The solutions of Equation 3 can be derived by rewriting Equation 3 as

$$\alpha \;\equiv\; \frac{c^2 - s^2}{2cs} = \frac{y-x}{2w}.$$

Now, recall that $\tau = \tan\theta = s/c$ solves the quadratic equation $\tau^2 + 2\alpha\tau - 1 = 0$ in τ, yielding the solutions $\tau = -\alpha \pm \sqrt{1+\alpha^2}$.

One of the two possible solutions for c and s is

$$\alpha = \frac{v^T v - u^T u}{2u^T v}, \qquad \tau = \frac{\mathrm{sign}(\alpha)}{|\alpha| + \sqrt{1+\alpha^2}}, \tag{6}$$

$$\text{then} \qquad c = \frac{1}{\sqrt{1+\tau^2}}, \qquad s = \tau c.$$

Comparing these formulas with those in Equation 4 reveals that the only difference is in the computation of α and the fact that the orthogonalization problem is solved solely with a premultiplication whereas an annihilation by means of a Jacobi rotation requires both premultiplication and postmultiplication. The former is therefore called a ***one-sided*** and the latter a ***two-sided*** transformation. Hestenes showed that the two transformations are equivalent [11]. We call the one-sided transformation of Equation 5 a ***Hestenes transformation*** to distinguish it from the original two-sided ***Jacobi transformation*** of Equation 2.

2.2 Existing SVD Algorithms

In the following, we discuss four relevant algorithms for computing the SVD. A number of additional algorithms exist, but are not discussed here because they appear to be less suited for parallelization in our judgment. We are not aware of a clear winner among parallel SVD algorithms that would provide the single best trade-off between numerical stability, algorithmic complexity, parallelizability, efficiency, and ease of programming across a large number of machines.

Golub-Kahan-Reinsch SVD

The SVD due to Golub, Kahan, and Reinsch [6, 8] has become the standard method on sequential and vector computers. It consists of two phases, bidiagonalization and subsequent diagonalization. Bidiagonalization can be achieved by means of alternating QR and QL factorizations to annihilate column and row blocks or Householder bidiagonalization [7]. Annihilating the remaining subdiagonal is known as chasing bulges, and is an inherently sequential process.

We could apply our stream QR algorithm [12], which factorizes an $N \times N$ matrix in time $O(N)$ on a two dimensional processor array of $O(N^2)$ processors. Unfortunately, the iterative diagonalization of the bidiagonal matrix would require time $O(N)$ as well, forming a dominating critical path that appears to render the Golub-Kahan-Reinsch SVD infeasible as a candidate for a stream algorithm. Nevertheless, the Golub-Kahan-Reinsch SVD requires $O(\log N)$ less work than our stream SVD, and may be competitive in terms of execution times.

Classical Jacobi Method

The classical Jacobi method [13, 14, 7] applies to symmetric matrices. It transforms a symmetric $N \times N$ matrix A into a diagonal matrix by means of a sequence of Jacobi transformations

$$A^{(k+1)} = J^T A^{(k)} J,$$

where $A^{(0)} = A$ and

$$\lim_{k\to\infty} A^{(k)} = \Sigma.$$

Each of the Jacobi transformations is chosen so as to annihilate the off-diagonal elements $a_{ij}^{(k)} = a_{ji}^{(k)}$ of largest absolute value. The classical Jacobi method exhibits high numerical stability and quadratic convergence [10, 5, 21]. To annihilate matrix elements $a_{ij} = a_{ji}$ (independent of iteration count k) with Jacobi rotation $J(i,j,\theta)$, the choice of α in Equation 4 becomes

$$\alpha = \frac{a_{jj} - a_{ii}}{2a_{ij}}.$$

The proof of convergence is based on the observation that the off-diagonal Frobenius norm $\mathrm{off}(A) = \sqrt{\sum_{i\neq j} a_{ij}^2}$ decreases due to a Jacobi transformation, while the Frobenius norm itself remains constant under orthogonal transformations such as the Jacobi transformation. A Jacobi transformation that annihilates the off-diagonal elements, increases the norm of the diagonal elements and decreases the off-diagonal Frobenius norm. Thus, each Jacobi transformation brings A closer to diagonal form [7, 5].

Cyclic Jacobi Methods

The main disadvantage of the classical Jacobi method is the computational effort required to determine the largest off-diagonal element of $A^{(k)}$. Rather than searching for the largest element, a computationally cheaper approach has prevailed which applies Jacobi transformations in a data-independent fashion. The key insight is to organize the computation in ***sweeps*** within which each matrix element is annihilated once. For a symmetric $N \times N$ matrix, one sweep consists of $N(N-1)/2$ Jacobi transformations, which is the number of matrix elements above or below the diagonal. Since the elements annihilated by one Jacobi transformation may be filled with non-zeros by a subsequent Jacobi transformation, the cyclic Jacobi method consists of an iteration of sweeps.

Figure 1 illustrates one sweep of Jacobi transformations. For $N = 4$, each sweep consists of 6 transformations. We may write this sweep as the sequence of right-associative transformations $T(i,j)A = J(i,j,\theta)^T A J(i,j,\theta)$:

$$T(3,2)T(3,1)T(2,1)T(3,0)T(2,0)T(1,0)A.$$

Each transformation annihilates the highlighted matrix elements. Figure 2 shows the rows and columns that are modified by one Jacobi transformation.

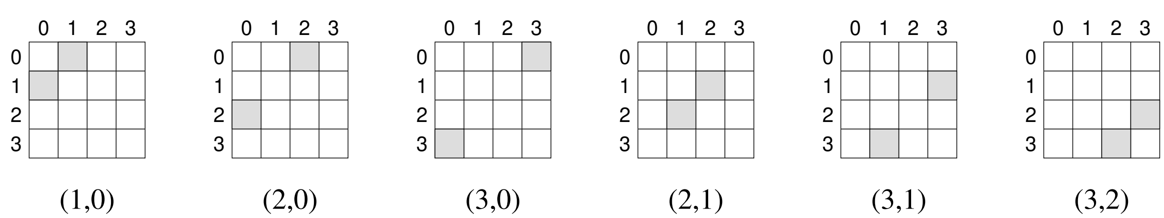


Figure 1: Sweep of Jacobi transformations for a symmetric 4×4 matrix. We use the index pair (i,j) as a shorthand for the notation $J(i,j,\theta)$.

Note that, in general, transformation $(2,0)$ in Figure 1 will fill the zero elements created by transformation $(1,0)$ with non-zero elements, $(3,0)$ will fill those created by $(2,0)$, and so on. The Jacobi method guarantees, however, that the magnitude of the off-diagonal elements decreases from sweep to sweep [5, 10, 21].

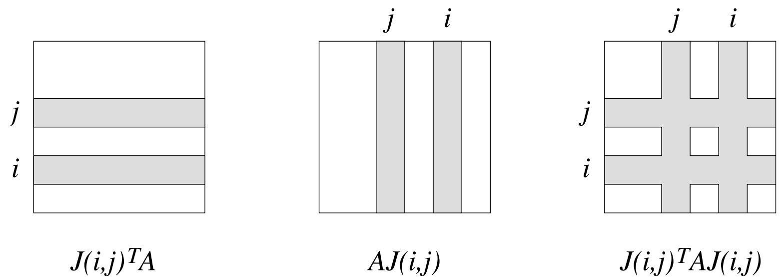


Figure 2: Jacobi transformation $J(i,j)$ modifies rows and columns i and j of matrix A. Premultiplication $J(i,j)^T A$ effects rows i and j, and postmultiplication $AJ(i,j)$ columns i and j.

The primary problem with the cyclic Jacobi method is the two-sidedness of the Jacobi rotation. Matrices are stored either in row-major or column-major format. As obvious from Figure 2, the two-sided Jacobi method traverses both. Thus, one of the two traversals will be less efficient on conventional memory architectures. This problem effects the classical Jacobi method as well, but is of secondary concern only.

Concerning parallelization, the cyclic Jacobi method is superior to the classical method, which is inherently sequential if applied as is. Since the premultiplication and postmultiplication of the cyclic Jacobi method modify two rows and columns only, we may actually perform $n/2$ Jacobi updates simultaneously. However, some care is required due to data dependencies involving the eight black elements in Figure 3 that are modified by two otherwise independent updates. In fact, we may compute the Jacobi rotations $J(i_1, j_1)$ and $J(i_2, j_2)$ independently, but the application of the corresponding transformations $T(i_1, j_1)$ and $T(i_2, j_2)$ is not commutative:

$$T(i_1, j_1)T(i_2, j_2)A \;\neq\; T(i_2, j_2)T(i_1, j_1)A,$$

as simple algebraic rewriting shows for the eight black elements in Figure 3. However, as long as we do not intermingle the two transformations, the order is immaterial indeed.

Several variations of the cyclic Jacobi method have been proposed in the past. They center around two aspects: (1) the type of orthogonalizing rotation, and (2) the order in which the rotations are applied. In the following, we point to some of the prominent variations. Forsythe and Henrici [5] extended Jacobi's method for the SVD of complex matrices. The order of the Jacobi transformation has generated quite some interest. Eberlein [4] appears to be one of the first to propose orderings for arbitrary matrices, although they turn out to be numerically unstable. Stewart [20] proposes improved versions for the ordering of transformations. The parallel ordering proposed by Brent, Luk, and Van Loan [2] provides the shortest distance between mutual rotations. They use the order for computing the SVD on a systolic array. The efficiency of their array is just 30 % in steady state, however.

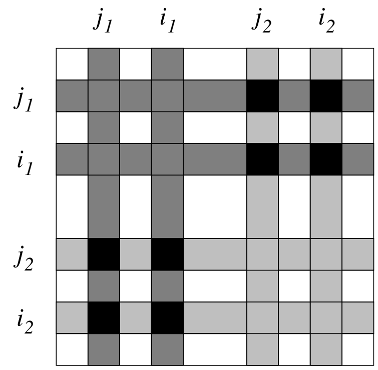


Figure 3: Two Jacobi transformations $J(i_1, j_1)$ and $J(i_2, j_2)$ are nearly independent with the exception of the eight black elements that are modified by both Jacobi updates.

Schreiber [19] proposes a block Jacobi method. Luk [16] published yet a another variant of the cyclic Jacobi method that has gained popularity for parallel machines. Luk assumes that A is an upper triangular matrix, possibly computed by means of a QR factorization, and introduces an odd-even ordering for the sequence of rotations, which preserves the triangular structure of the matrix.

Hestenes-Jacobi Method

Hestenes [11] introduced the one-sided Jacobi method by discovering the equivalence between orthogonalizing two vectors and annihilating a matrix element by means of orthogonal plane rotations. We will call the one-sided Jacobi method *Hestenes-Jacobi method* in the following.

The Hestenes-Jacobi method generates an orthogonal matrix U as a product of plane rotations. Premultiplying A with U yields a matrix B

$$UA = B,$$

whose rows are orthogonal, that is

$$b_i^T b_j = 0 \quad \text{for } i \neq j.$$

Note that B is not a diagonal matrix as obtained from a two-sided Jacobi method. However, we may normalize matrix B by computing the square of the row norms $s_i = b_i^T b_i$, and writing B as $B = SS^{-1}B = SV$, where V is computed from B by dividing row b_i by $s_i = b_i^T b_i$, and S is a diagonal matrix $\mathrm{diag}(s_0, \ldots, s_{M-1})$. The resulting matrix V is orthonormal, and the s_i are the non-negative squares of the singular values. Of course, care must be exercised with zero-valued singular values. As a consequence of this normalization, and because U is orthogonal, we may write:

$$UA = SV \quad \Leftrightarrow \quad A = U^T SV.$$

Interpreting U and V as transposed forms, this equation equals Equation 1 which defines the SVD.

The difference in both functional effect and computational effort between a two-sided and a one-sided Jacobi rotation is one polynomial degree. A two-sided rotation uses $\Theta(n)$

operations to annihilate $\Theta(1)$ elements, and the one-sided rotation employs $\Theta(n)$ operations to orthogonalize two rows consisting of $\Theta(n)$ elements. However, while a two-sided rotation modifies both two rows and two columns of a matrix, the one-sided rotation modifies two rows only. This is a major advantage of the one-sided rotation, that we will exploit below. Not only are the matrix rows the smallest unit of computation, but also, two plane rotations involving distinct row pairs are truly independent.

Chartres [3], perhaps unaware of Hestenes work, derived a very similar variant of Jacobi's method. He observed that the classical Jacobi method requires access to two rows and two columns for each Jacobi update, and realized the poor match for machines with small random access memories, which store matrices either in row major or in column major form on a backup tape. Chartres also introduced the notion of the *one-sided* transformation $A^{(k+1)} = J^T A^{(k)}$ as an alternative to the *two-sided* transformation $A^{(k+1)} = J^T A^{(k)} J$ of the classical Jacobi method. In fact, Chartres uses the row-orthogonalizing formulation that we use above, while Hestenes' original discussion is based on the equivalent column orthogonalization $AV = B$. The Hestenes-Jacobi method became popular in the 1970's. Nash [17] derived the plane rotation from a maximization problem. Luk [15] implemented the one-sided Hestenes-Jacobi method on the ILLIAC IV, a SIMD array computer. Interestingly, when Chartres published his work twenty years earlier, he appeared to be engrossed in implementing eigenvalue computations on the SILLIAC, the Sydney University version of the first ILLIAC at Urbana Champaign.

3 Parallelism in the Hestenes-Jacobi Method

We are aware of two ways for harvesting parallelism from the Hestenes-Jacobi method. We discuss the second method below. The first method has been realized by Chartres [3] and Luk [15] already, and is based on the reordering of the sequence of Hestenes transformations. Since two Hestenes transformations with distinct row pairs are independent, up to $M/2$ plane rotations can be performed in parallel. However, not every sequence of Hestenes transformations constitutes a sweep that leads to convergence. This problem has been discussed by Hansen [9] already. We may interpret it as limiting the available parallelism.

Observe that the quadratic convergence of the sequential cyclic-by-rows scheme can be viewed intuitively as being due to a bidirectional information flow through Hestenes transformations. Consider the example $M = 3$ of a matrix A with three rows. A sweep consists of three Hestenes transformations, that is pairs of rows to be rotated. In general, a sweep in the Hestenes-Jacobi method may be viewed as a combinatorial task of pairing each row with every other row. From elementary combinatorics we know that there are $\binom{M}{2} = M(M-1)/2$ such rotations. In the cyclic-by-rows ordering we choose the sequence $[(0,1),(0,2),(1,2)]$ of transformations for each sweep. The first transformation $(0,1)$ deposits information about row 0 in row 1 and vice versa. Thus, when computing the second transformation $(0,2)$, information about row 2 propagates via row 1 to row 2. Therefore, after two transformations, information about all three rows has propagated across the whole matrix. The sequence of transformations is inherently sequential for $M = 3$, because there exist no two pairs of distinct rows.

Now, consider the case $M = 4$ of a matrix A with four rows. There are $\binom{4}{2} = 6$ pairs of rows to be rotated. With the cyclic-by-rows order, the sequence of a sweep is

$$[(0,1),(0,2),(0,3),(1,2),(1,3),(2,3)]. \tag{7}$$

Another possible sequence for a sweep groups independent pairs and executes them in parallel. Here is an example

$$[\{(0,1),(2,3)\},\{(0,3),(1,2)\},\{(0,2),(1,3)\}], \tag{8}$$

where the pairs in curly brackets are independent. The convergence behavior of these two sequences appears to be similar in practice. This is not obvious, however, because the Hestenes transformations do not commute in general, and the two sequences 7 and 8 do not produce identical results. Let us investigate the commutativity of Hestenes transformations in more detail to understand why the order of the transformations is important.

Lemma 1 ***(Rotation Commutativity)***
Given a matrix A and two Jacobi rotations $J(i,j,\theta)$ and $J(p,q,\zeta)$ corresponding to Hestenes transformations of rows a_i^T, a_j^T and a_p^T, a_q^T of A. Then, the following statements hold:

1. *$J(i,j,\theta)^T J(p,q,\zeta)^T A = J(p,q,\zeta)^T J(i,j,\theta)^T A$ for $i \neq p,q$ and $j \neq p,q$, that is rotations involving distinct pairs of rows commute, and*
2. *$J(i,j,\theta)^T J(j,k,\zeta)^T A \neq J(j,k,\zeta)^T J(i,j,\theta)^T A$, that is two rotations that share at least one row do not commute.*

The proof is straightforward. For case 1, we can show that

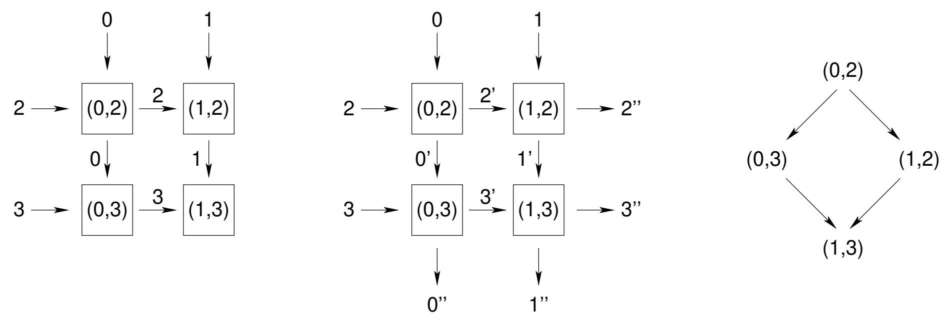
$$\begin{pmatrix} c_\theta & -s_\theta \\ s_\theta & c_\theta \end{pmatrix} \begin{pmatrix} c_\zeta & -s_\zeta \\ s_\zeta & c_\zeta \end{pmatrix} = \begin{pmatrix} c_\zeta & -s_\zeta \\ s_\zeta & c_\zeta \end{pmatrix} \begin{pmatrix} c_\theta & -s_\theta \\ s_\theta & c_\theta \end{pmatrix},$$

and for case 2, we show that there exist angles θ and ζ such that

$$\begin{pmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_\zeta & -s_\zeta \\ 0 & s_\zeta & c_\zeta \end{pmatrix} \neq \begin{pmatrix} 1 & 0 & 0 \\ 0 & c_\zeta & -s_\zeta \\ 0 & s_\zeta & c_\zeta \end{pmatrix} \begin{pmatrix} c_\theta & -s_\theta & 0 \\ s_\theta & c_\theta & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Thus, the two sequences 7 and 8 above are not identical, for example because transformation $(0,2)$ is applied *before* $(0,3)$ in the strictly sequential sequence 7, whereas $(0,2)$ is applied *after* $(0,3)$ in the parallel sequence 8. Brent and Luk [1] presented a parallel implementation of the parallel ordering scheme on a linear array of up to $M/2$ processors that is also referred to as *ring procedure* [7, Section 8.5.10]. In this implementation each processor receives two rows, performs the Hestenes transformation and sends the transformed rows to its left and right neighbors. Since each processor receives different rows, however, there is no temporal locality to be exploited by streaming data. In short, the linear array scheme is not a candidate for a stream algorithm.

The second form of parallelism inherent in the Hestenes-Jacobi method reveals itself by splitting the computation of the rotation from the transformation, that is the application of the rotation. If we are willing to sacrifice a hopefully insignificant percentage of the quadratic convergence behavior, we can create parallelism in a different way.[3] Figure 4 shows the basic idea for a 2×2 array of compute processors. In a stream architecture [12], the array would be supplemented by memory processors at the periphery. The computation proceeds in two phases. First, we compute the plane rotations by streaming rows 0 through 3 into the array, forming the inner products $(0,2)$, $(1,2)$, $(0,3)$ and $(1,3)$ on the processors where the rows cross, and storing the sines and cosines of the rotations locally. In the second phase, we stream row 0 through 3 through the array once more, this time to apply the rotations. Each processor receives a value of a row from the top and the left, applies the transformation, and outputs the result at the bottom and the right, respectively. In effect, each row entering from the left is transformed twice before exiting the array on the right. Analogously, each row entering at the top is transformed twice before exiting the array at the bottom.



(a) compute rotations (b) apply rotations (c) dependencies

Figure 4: A block transformation of our parallel Hestenes-Jacobi method. In a first phase (a) we *compute* the rotations $(0,2)$, $(1,2)$, $(0,3)$, and $(1,3)$, and store them locally. During the second phase (b) rows 0 to 3 are streamed into the array once again, this time in order to *apply* the rotations.

Note that the computation of the four rotations in Figure 4(a) occurs in parallel, while the transformations are computed in an ordered fashion in Figure 4(b). With respect to commutativity, the rotations $(0,2)$ and $(1,3)$ as well as $(1,2)$ and $(0,3)$ are mutually independent, while $(0,3)$ and $(0,2)$ are not, and $(1,2)$ and $(1,3)$ are neither. The dependencies are illustrated in Figure 4(c). Our parallel computation differs from the sequential cyclic-by-rows order, because it computes all rotations before computing the transformations. However, the order of the transformations matches that of the dependency graph.

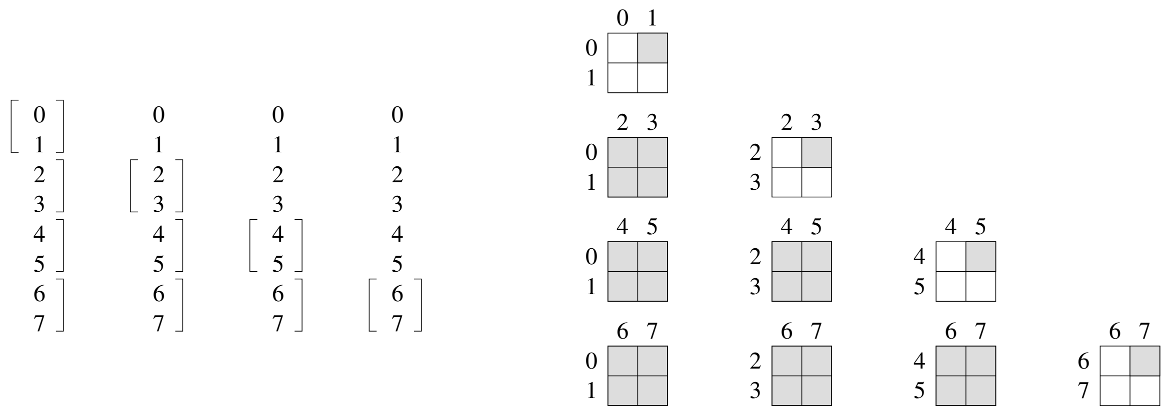
Let us emphasize the fact that we compute multiple rotations independently and apply all transformations thereafter, rather than computing and applying each rotation sequentially as done in the conventional cyclic-by-rows order. Formally, we replace the sequence $A^{(k)}$ of

[3]The second way of creating parallelism was motivated by the design goal of achieving 100 % compute efficiency on a stream architecture [12]. In fact, we postulated the structure of the data streams before analyzing the resulting numerical behavior.

the conventional transformation

$$A^{(k+1)} = J(i,j,\theta)A^{(k)} \qquad \text{with} \qquad A^{(k+1)} = J(i_r,j_r,\theta_r)\dots J(i_1,j_1,\theta_1)A^{(k)},$$

where r denotes the number of rotations computed from the values of $A^{(k)}$. Informally, we distribute information about individual matrix elements across the matrix without exploiting the full potential for orthogonalization that the strictly sequential scheme offers. To distinguish our transformation from the conventional cyclic-by-rows sequence of rotations, we call it a ***block transformation***.



(a) row grouping (b) block transformations

Figure 5: One block sweep of our stream Hestenes algorithm. The numbers identify entire rows of matrix A. The grouping of rows into blocks is shown on the left, and the corresponding block transformations on the right. The group of rows marked by the left bracket is paired with all groups marked by the brackets on the right. One column or row groupings in (a) corresponds to one column of block transformations in (b).

We may use a sequence of block transformations to form a ***block sweep***. Figure 5 shows a block sweep for a block size of 2×2 and a matrix A with $M = 8$ rows. For a block size of 1, this algorithm is a standard cyclic-by-rows Hestenes-Jacobi method. During the first four block transformations in the leftmost column of Figure 5(b), which correspond to the row groupings in the leftmost column of Figure 5(a), we perform the sequence of block Hestenes transformations

$$\begin{aligned}
A^{(1)} &= J(0,1)^T A^{(0)}, \\
A^{(2)} &= J(1,3)^T J(1,2)^T J(0,3)^T J(0,2)^T A^{(1)}, \\
A^{(3)} &= J(1,5)^T J(1,4)^T J(0,5)^T J(0,4)^T A^{(2)}, \\
A^{(4)} &= J(1,7)^T J(1,6)^T J(0,7)^T J(0,6)^T A^{(3)}.
\end{aligned}$$

Note that transformations $J(0,0)$ and $J(1,1)$ are undefined, and $J(1,0)$ is identical to $J(0,1)$ by symmetry of the Hestenes transformation. Hence, block transformations of a row group with itself perform fewer Hestenes transformations than distinct row groups. Only the shaded squares in Figure 5(b) represent processors performing Hestenes transformations. In general, only the processors above (or, equivalently, below) the diagonal perform transformations when a row group is paired with itself.

Now, we wish to increase the block size from a 2×2 array to an arbitrarily large $R \times R$ array. The larger the network size R, the more operations we can execute concurrently. In doing so, we also increase the number of dependency violations compared to the execution order of the strictly sequential cyclic-by-rows scheme, however. We may view these violations as missed opportunities for propagating information across the matrix. Let us assess the magnitude of this problem. Consider one block transformation on an $R \times R$ array with distinct row blocks. This block transformation involves $2R$ rows of matrix A. Ideally, a sequential order such as the cyclic-by-rows scheme would pair all rows with each other for orthogonalization, resulting in $\binom{2R}{2}$ transformations. In contrast, using our square array of R^2 processors, we compute only R^2 transformations, one per processor. Thus, we miss out on $\binom{2R}{2} - R^2 = (2R^2 - R) - R^2$ opportunities for propagating information via Hestenes transformations. Nevertheless, even for large processor arrays with network size R, our parallel scheme is no more than a factor of two off the sequential cyclic-by-rows scheme:

$$\lim_{R\to\infty} \frac{\binom{2R}{2}}{R^2} \;=\; \lim_{R\to\infty} \frac{2R^2 - R}{R^2} \;=\; 2.$$

Encouraged by this plausibility argument, we now present our stream SVD algorithm.

4 The Stream SVD Algorithm

Our stream SVD algorithm is based on the Hestenes-Jacobi method and on the ideas developed in Section 3. The key idea is to sacrifice speed of convergence for parallelism. Since Hestenes transformations are orthogonal transformations which leave singular values unchanged, it does not matter in principle how many we apply. There is the issue of a potential loss of numerical accuracy, however. Since even the cyclic-by-rows Hestenes method sacrifices convergence behavior compared to the classical Jacobi method, we may push the envelope even further by sacrificing convergence behavior in favor of parallelism.

Algorithm 1 shows the pseudocode of the sequential version of our Hestenes-Jacobi method for computing the singular values of an $M \times N$ matrix A, christened ***Stream Hestenes SVD*** or just ***Stream SVD*** for short. The algorithm receives as arguments array A in row-major layout, array S for storing the squares of the row norms, and array P for storing $R \times R$ sines and cosines representing R^2 Jacobi rotations. The singular values are stored in array σ in lines 31 and 32. We discuss the organization of the data flow through a parallel stream architecture as well as the computation of U and V below.

The body of the **repeat** loop constitutes one *block sweep*. During each block sweep, we orthogonalize all $\binom{M}{2}$ pairs of rows of matrix A. The outer **for** loops (lines 6 and 8) implement the *block sweep*. In lines 10–13, we compute the square of the norms of the rows within the block. These two inner products can be saved in favor of a computationally less expensive update, as explained below. *Block phase 1* consists of lines 14–23. Here we compute the Jacobi rotations for one block, and store the rotations in array P. Thereafter,

Algorithm 1 Stream Hestenes SVD (row-major arrays: $A[M][N], S[M], P[R][R]$)

```
for i = 0 to M do
    S[i] = A[i]^T A[i]
δ = ε · Σ_{i=0}^{M-1} S[i]
repeat
    converged ← true
    for lb_i = 0 to M by R do
        ub_i ← min(lb_i + R, M)
        for lb_j = lb_i to M by R do
            ub_j ← min(lb_j + R, M)
            for i = lb_i to ub_i do
                S[i] ← A[i]^T A[i]
            for j = lb_j to ub_j do
                S[j] ← A[j]^T A[j]
            for i = lb_i to ub_i do
                for j = lb_j to ub_j do
                    if i < j then
                        g ← A[i]^T A[j]
                        if |g| > δ then
                            converged ← false
                        if |g| > ε then
                            P[i mod R][j mod R].c, P[i mod R][j mod R].s ← jacobi(S[i], S[j], g)
                        else
                            P[i mod R][j mod R].c, P[i mod R][j mod R].s ← 1, 0
            for i = lb_i to ub_i do
                for j = lb_j to ub_j do
                    if i < j then
                        c, s ← P[i mod R][j mod R].c, P[i mod R][j mod R].s
                        for k = 0 to N do
                            A[i][k], A[j][k] ← cA[i][k] − sA[j][k], sA[i][k] + cA[j][k]
until converged = true
for i = 0 to M do
    σ[i] ← sqrt(A[i]^T A[i])
```

during *block phase 2*, we perform the Hestenes transformations in lines 24–30. The guard $i < j$ in lines 16 and 26 selects the upper triangular set of pairs in row block $[lb_i, ub_i] \times [lb_i, ub_i]$, cf. Figure 5(b).

Procedure *jacobi* is called in line 21 of Algorithm 1, and computes the Jacobi rotation associated with the Hestenes transformation. Note that a division by zero is prevented in line 1 of Procedure 2 by the guard in line 20 of Algorithm 1.

Procedure 2 $c, s = \text{jacobi}(a, b, g)$

1: $w \leftarrow (b - a)/2g$
2: $t \leftarrow \mathbf{sign}(w) / \left(|w| + \sqrt{1 + w^2}\right)$
3: $c \leftarrow 1/\sqrt{1 + t^2}$
4: $s \leftarrow t\,c$

Our convergence criterion accounts for the condition of matrix A. Value δ, computed in lines 1–3, is the sum of the inner products $A[i]^T A[i]$ of the rows i of A, scaled by a small number ϵ, which may be as small as the machine epsilon. Our iteration terminates if all $\binom{M}{2}$ pairs of rows are sufficiently orthogonal, that is if $|A[i]^T A[j]| \leq \delta$ for all $i \neq j$.

Accumulating the Row Norm

In the Hestenes-Jacobi method, we do not need to recompute the square of the row norms ${a_i}^T a_i$ at the beginning of each block sweep in lines 10–13 of Algorithm 1. Instead, we may update the norm on demand, that is after applying Hestenes transformation $J(i, j)$ to rows i and j. Then, the square of the norms can be updated by means of the sine s and cosine c of the Jacobi rotation as follows:

$$\begin{aligned} ||a_i^{(k+1)}||^2 &= c^2 ||a_i^{(k)}||^2 + s^2 ||a_j^{(k)}||^2 - 2cs {a_i^{(k)}}^T a_j^{(k)}, \\ ||a_j^{(k+1)}||^2 &= s^2 ||a_i^{(k)}||^2 + c^2 ||a_j^{(k)}||^2 + 2cs {a_i^{(k)}}^T a_j^{(k)}. \end{aligned}$$

The drawback of the computational savings may be a loss of accuracy. In fact, for numerical stability, we may have to compute the row norms periodically rather than update them only. However, we have not experienced any such problems with our experiments thus far.

In principle, we could also update the value of the inner product

$${a_i^{(k+1)}}^T a_j^{(k+1)} = (c^2 - s^2) {a_i^{(k)}}^T a_j^{(k)} + cs(||a_i^{(k)}||^2 - ||a_j^{(k)}||^2).$$

However, this update requires $(M-1)^2/2$ storage and would require updating all memoized inner products ${a_i^{(k+1)}}^T a_j^{(k+1)}$ for $j \neq i$ rather than for row j only. Thus, recomputing the inner product $a_i^T a_j$ is clearly the preferred alternative.

Computing U and V

In the following, we discuss the computation of the orthogonal matrices U and V within the context of the Hestenes-Jacobi method. We compute U^T by applying each Hestenes transformation that we apply to A to U^T as well:

$$U^{(k+1)^T} = J^T U^{(k)^T},$$

where $U^{(0)} = I_M$.

Since, the premultiplications with J^T are orthogonal transformations, $U^T = \lim_{k\to\infty} U^{(k)^T}$ is orthonormal. Now, we construct matrix $U^T A = B$ with Algorithm 1. Matrix B consists of mutually orthogonal rows by construction, and since $U^T A = \Sigma V^T$, we can compute the singular values and the orthonormal matrix V from the computed matrix B row by row:

$$\sigma_i = ||B_i|| \qquad \text{and} \qquad v_i^T = \frac{B_i}{||B_i||}.$$

We normalize the rows of B, transforming the matrix of mutually orthogonal rows into an orthonormal matrix. Since, $B = \Sigma V^T$, the singular values are nothing but the row norms of B. If we maintain the rows norms by updating them as described above, we can also save the inner products within the square root in line 32 of Algorithm 1.

Data Flows through Stream Architecture

In this section we present an efficient implementation of the stream SVD on our stream architecture [12]. We assume that the machine consists of an array of $R \times R$ compute processors, augmented with $4R$ memory processors at the periphery. The stream SVD partitions the $M \times N$ matrix A into block transformations (see Figure 4), each of which consists of two phases, ***rotation computation*** and ***rotation application***. Lines 14–23 of Algorithm 1 specify the computation of rotations for a pair of row blocks (r_i, r_j), where $r_i = \{a_i \mid lb_i \leq i < ub_i\}$ and $r_j = \{a_j \mid lb_j \leq j < ub_j\}$. Lines 24–30 specify the application of the rotations computed in lines 14–23. As a boundary case, we must compute and apply the rotations to row blocks r_i and r_j with $i = j$, that is rotating the rows within row block r_i. This boundary case applies to the blocks on the diagonal of Figure 5(b).

Our stream SVD is based on four systolic algorithms for an $R \times R$ array of compute processors: (1) rotation computation of row blocks r_i and r_j with $i < j$ (see Figure 6), (2) rotation application to row blocks r_i and r_j with $i < j$ (see Figure 7), and the analgous computations for the boundary case: (3) rotation computation of row block r_i with itself (see Figure 8), and (4) rotation application within row block r_i (see Figure 9). These four systolic algorithms form the core of the stream SVD. There are two additional computations, that we will not discuss in detail, because they are not critical for high efficiency. First, the computation of the square of the row norm $||a_i||^2 = a_i^T a_i$ in lines 10–13 of Algorithm 1 can be accomplished on the memory processors while streaming row a_i into the array of compute processors. The square of the norm will be available after the entire row has been streamed into the array. Second, there is the computation of the convergence criterion based on the Boolean variable *converged* in Algorithm 1. During a block sweep, each compute processor maintains a local copy of this variable. At the end of each block sweep, we perform a global Boolean *and*-operation and scatter the resulting value to all processors, including the memory processors.

In the following, we assume that each row block contains R rows and N columns. Thus, row blocks are $R \times N$ matrices. A block transformation of two distinct row blocks requires R^2 rotations. During the systolic rotation computation illustrated in Figure 6, we use R^2 processors to compute and store these rotations in form of a sine and cosine value locally. Then, we execute the systolic rotation application, illustrated in Figure 7, to perform R^2 Hestenes transformations. These systolic algorithms implement the dataflows shown in Figures 4(a) and 4(b), respectively, except that the horizontal data streams are directed from right to left rather than from left to right in order to facilitate composition. Processors in the figures are marked grey at a time step if they compute and produce the bold faced values.

The systolic rotation computation in Figure 6 consists of computing $g_{ij} = a_i^T a_j$ for the cross product of rows $i \in r_i$ and $j \in r_j$ in a streamed fashion. Once g_{ij} is available and the row norms $||a_i||^2$ and $||a_j||^2$ have been received as the last items of their respective row streams, each processor computes the rotation values s_{ij} and c_{ij} according to Procedure 2. In the example of Figure 6, we show $R = 3$ rows per row block, and $N = 4$ columns per row. The row block streamed horizontally from right to left consists of rows 0–2 and the row block streamed from top to bottom consists of rows 3–5. The processor that receives row i from the right and row j from the top computes s_{ij} and c_{ij}.

The systolic rotation application shown in Figure 7 succeeds the rotation computation of Figure 6. With the rotation values in place, row blocks 0–2 and 3–5 are streamed into the array once again. Processor p_{ij} is responsible for rotating row i, received from the right and row j received from the top by executing the loop in lines 28–29 of Algorithm 1, where loop index k corresponds to the element position in the row streams. For example, processor p_{03} in the top right corner of the array receives values a_{00} and a_{30} during step (0). During step (1), it computes $a_{00}^{(1)} = c_{03}a_{00} - s_{03}a_{30}$ and passes the result to the left. During step (2), processor p_{03} computes $a_{30}^{(1)} = s_{03}a_{00} + c_{03}a_{30}$ and passes the value to the bottom neighbor. The rotated rows 0–2 exit the array on the left and the rotated rows 3–5 exit the array at the bottom.

In the following, we briefly discuss the boundary case involving row blocks on the main diagonal of Figure 5(b). Figure 8 shows the systolic algorithm for computing the rotations within row block 0–2. Since there are $R(R-1)/2$ rotations within a row block, we use only the upper triangular portion of the processor array. We stream the rows top to bottom into the array until the streams arrive at the main diagonal. There, we turn the streams to the right to generate the pattern of crossing rows used in Figure 6. Figure 8 shows the computation of the rotation on the three upper triangular processors of the array for $R = 3$ and $N = 4$. The corresponding systolic algorithm for the rotation application is shown in Figure 9. As for the rotation computation, we stream the rows from top to bottom into the array. When the stream enters the processor on the main diagonal, we turn it to the right. The rotated row streams exit the array on the right.

Our stream SVD is a composition of the four systolic algorithms shown in Figures 6–9. The composition in Figure 10 illustrates one block sweep of the stream SVD for $R = 2$, $M = 6$, and arbitrary N. The 2×2 processor array is shown at the center of each of the computational steps. The staggered organization of the rows indicates the temporal order of rows streaming through the array. This illustration does not include the details about the computation of the row norms on the memory processors and the scattering of

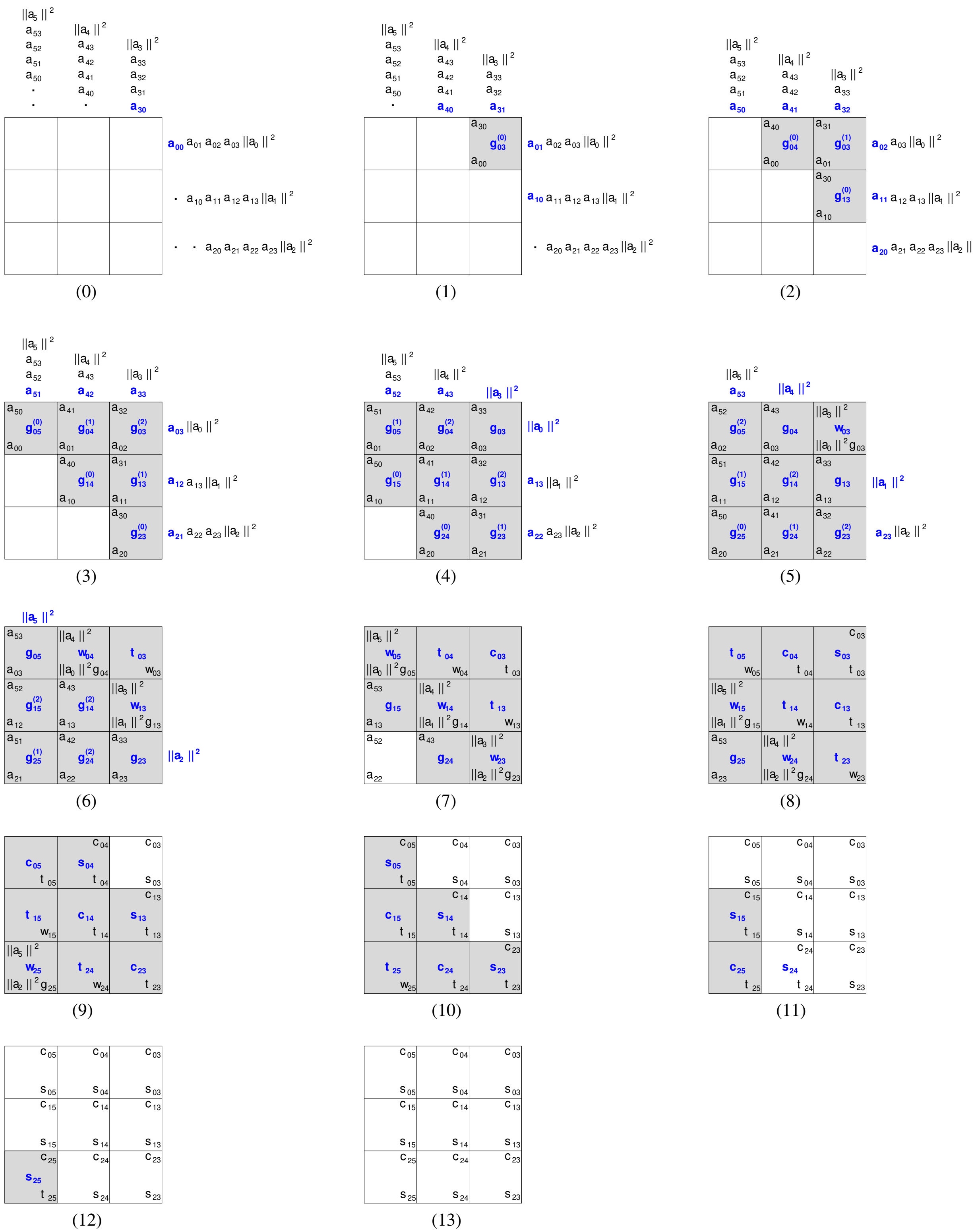


Figure 6: Systolic rotation computation for two $R \times N$ row blocks with $R = 3$ and $N = 4$. Values $g_{ij} = a_i^T a_j$, $w_{ij} = ||a_i||^2||a_j||^2/g_{ij}$, and $t_{ij} = \text{sign}(w_{ij})/\left(|w_{ij}| + \sqrt{1 + w_{ij}^2}\right)$ are the intermediate values of Procedure 2 used to compute the rotation c_{ij} and s_{ij} of rows i and j.

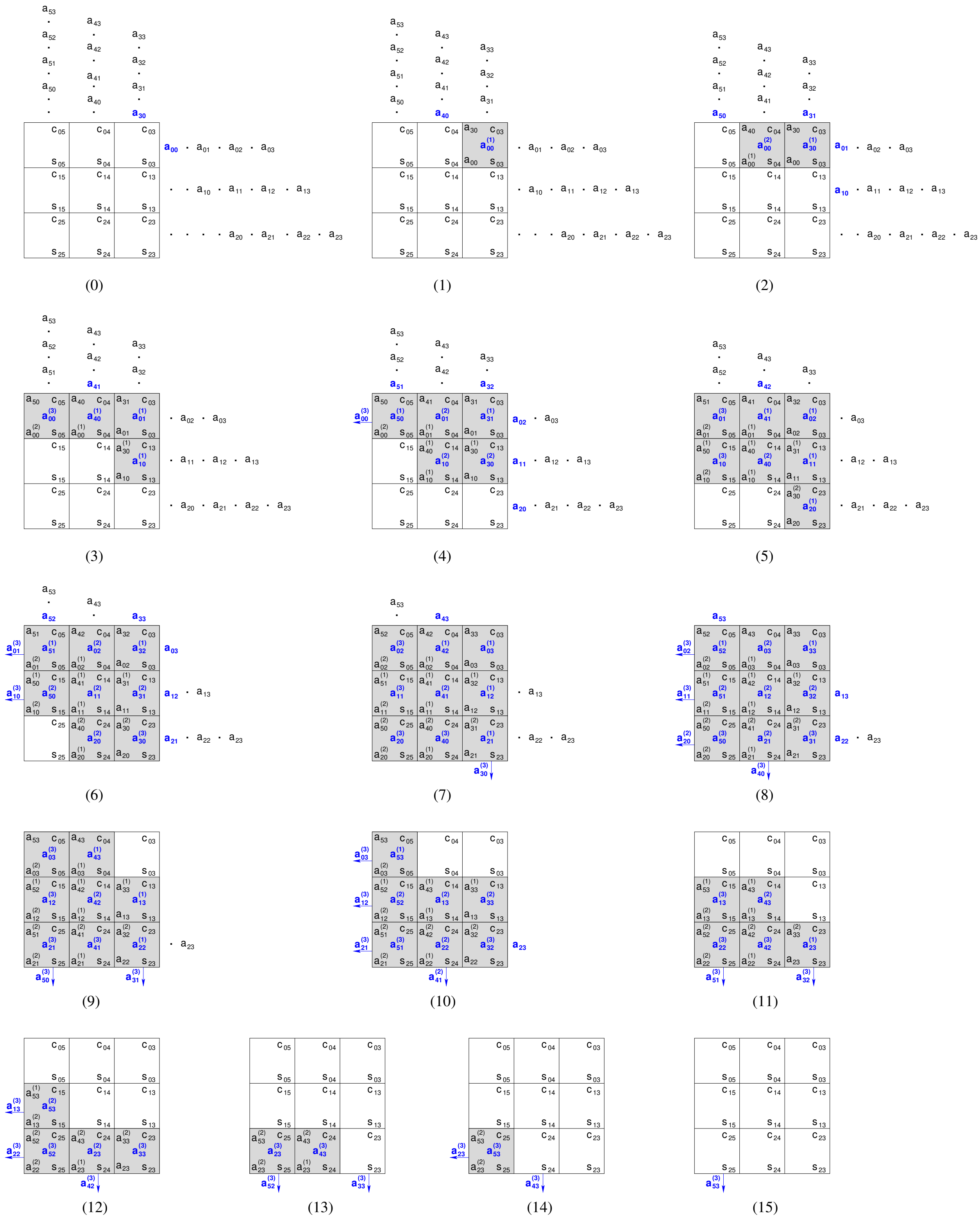


Figure 7: Systolic rotation application following the rotation computation of Figure 9.

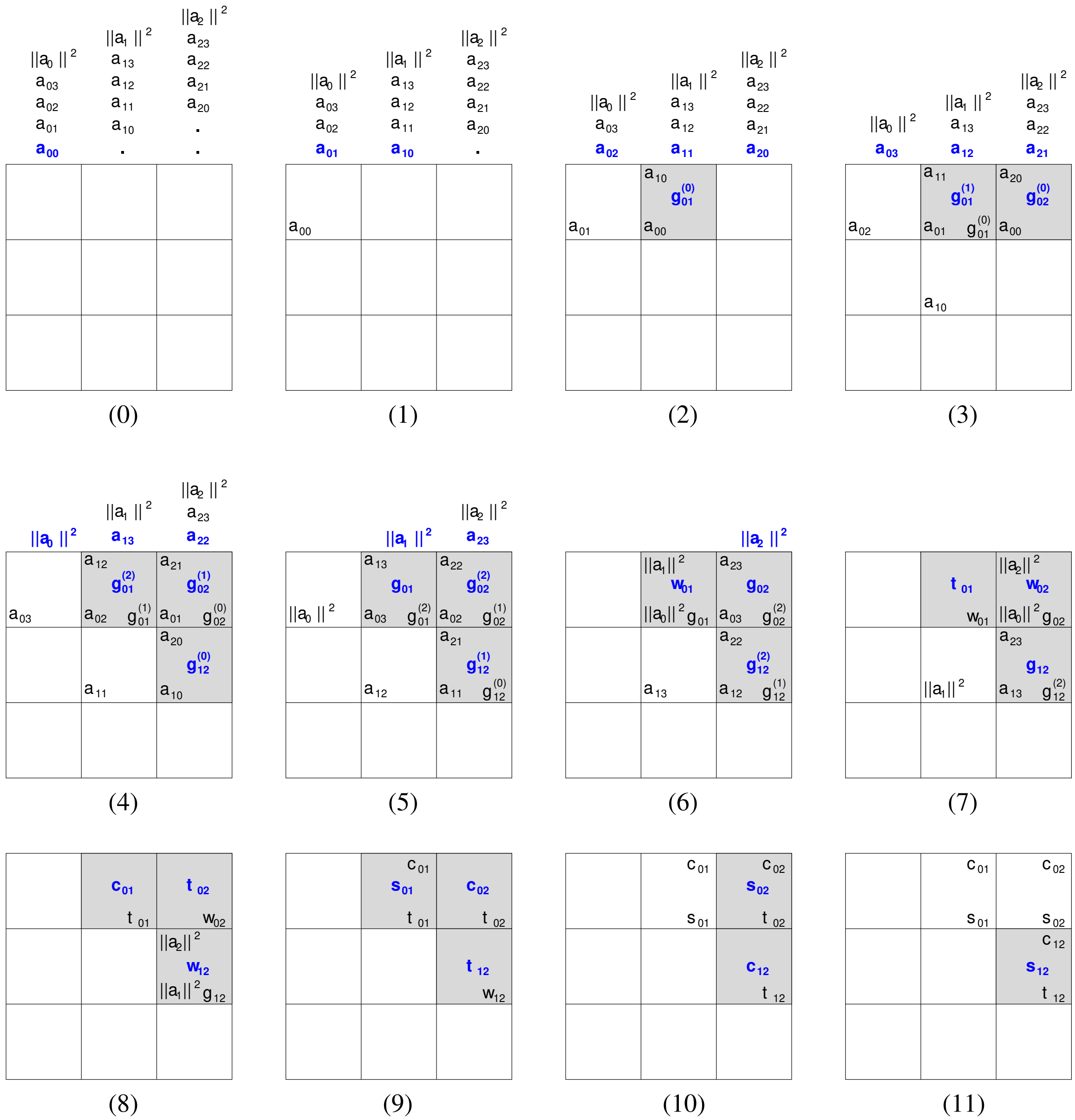


Figure 8: Boundary case for rotation computation; cf. Figure 6.

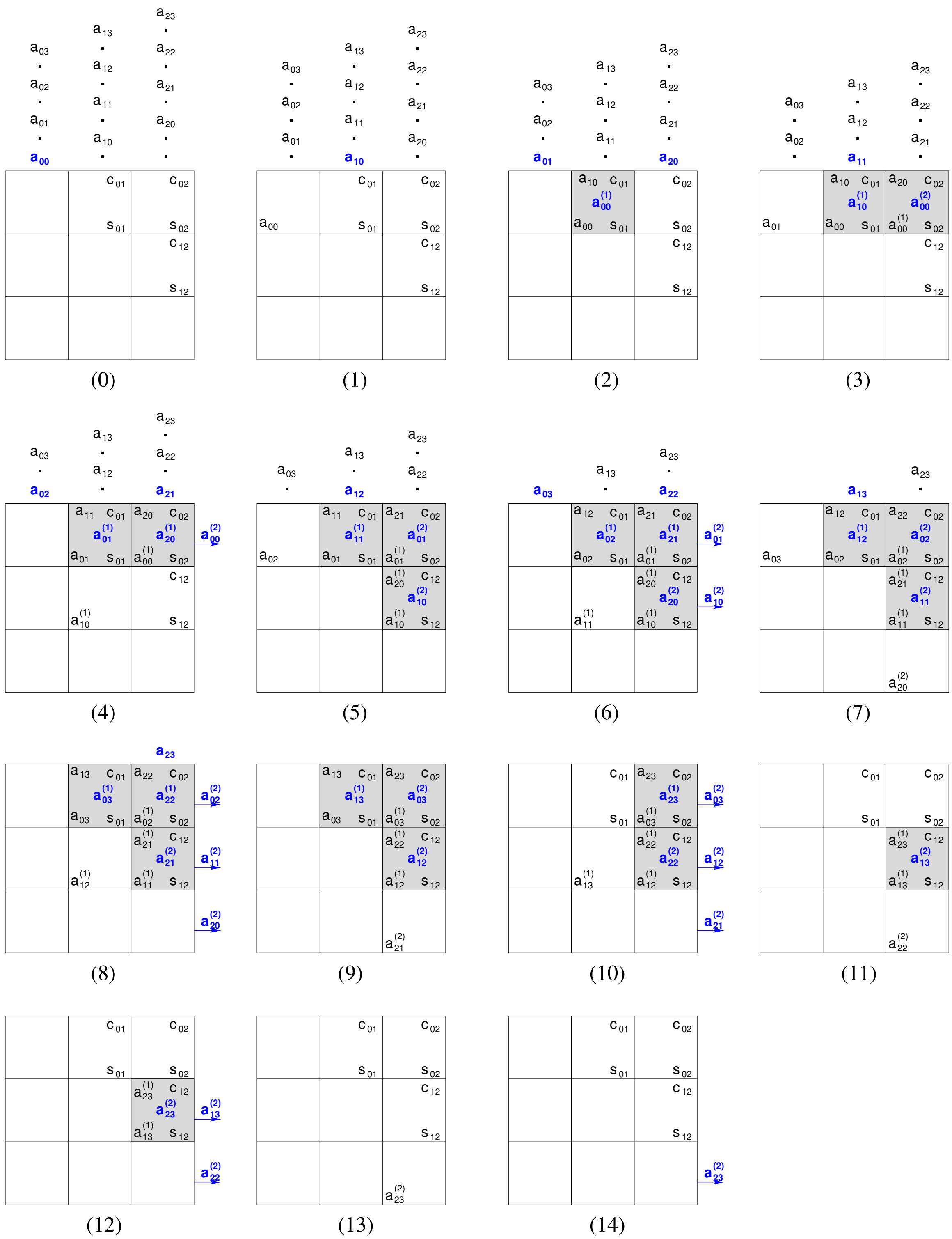


Figure 9: Boundary case for rotation application; cf. Figure 7.

the convergence criterion at the very end of the block sweep.

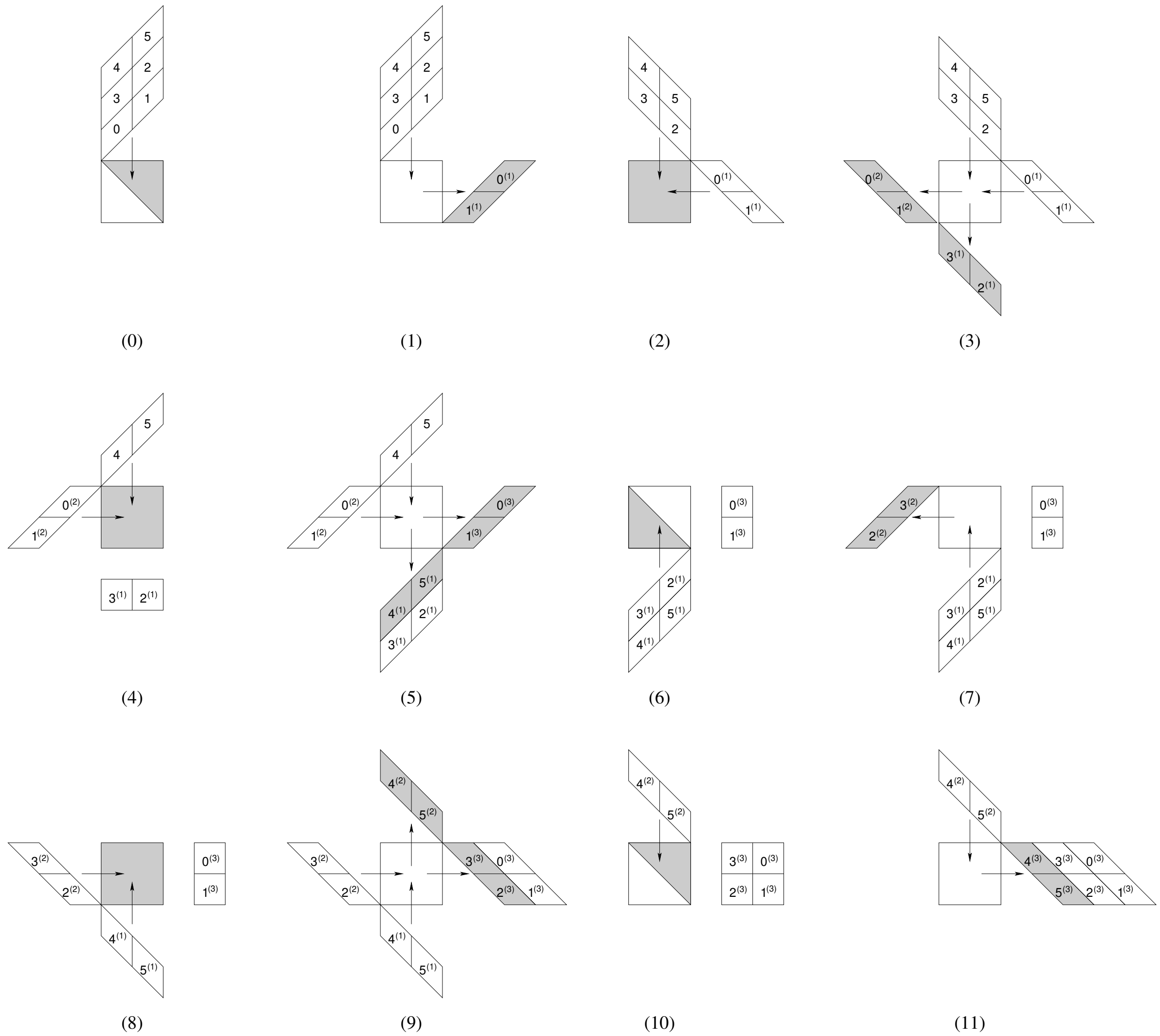


Figure 10: One block sweep of a stream SVD of $M \times N$ matrix A on an $R \times R$ array of compute processors with $R = 2$ and $M = 6$. Matrix A is partitioned into three row blocks $\{0,1\}$, $\{2,3\}$, and $\{4,5\}$. Even numbered phases denote rotation computations, and odd numbered phases rotation applications. Grey shaded areas hold newly computed values.

The key to an efficient composition is the data distribution of the matrix rows. As shown in Figure 10(0), we use a *snaked row distribution*. From this initial distribution, we can apply our systolic algorithms using merely spatial rotations and reflections in order to match computation and data streams without redundant data movements. During phase (0), we compute the rotations within block $\{0,1\}$, and apply these rotations during phase (1). As a result, we produce the rotated rows $0^{(1)}$ and $1^{(1)}$ on the right-hand side of the processor array. During phases (2) and (3), we compute and apply the rotations of block $\{0,1\}$ with block $\{2,3\}$ as illustrated in detail in Figures 6 and 7. Analogously, we compute and apply the rotations of block $\{0,1\}$ with block $\{4,5\}$ during phases (4) and (5). However, the systolic algorithm is reflected over a vertical line, because the rows $0^{(2)}$ and $1^{(2)}$ stream from left to right through the array. During phases (6) and (7), we compute and apply the rotations within row block $\{2,3\}$. Compared to our illustration in Figures 8 and 9, the systolic algorithm is rotated by 180° around the center of the processor array.

Efficiency Analysis

Since the primary goal of stream algorithms is to provide optimally efficient computations asympotically for large numbers of processors, we offer an efficiency analysis of our stream SVD. We analyze the efficiency of one block sweep over an $M \times N$ matrix A on an $R \times R$ array of compute processors with $4R$ memory processors on the periphery. We introduce the ratios $\sigma_M = M/R$ and $\sigma_N = N/R$ that have proven their usefulness in [12] already.

We count the number of operations performed during one block sweep in units of floating-point multiply-and-add operations, because they constitute the majority of operations and dominate the time complexity. When immaterial for the overall count, we resort to convenient approximations. For example, we approximate the computations of the four assignments within the Jacobi Procedure 2 as four multiply-and-add operations. In contrast, the inner products for the rotation computation and the rotation applications are counted exactly. It is obvious from Figure 5 that one block sweep comprises $M(M-1)/2$ Hestenes transformations. Each transformation consists of the rotation computation and the rotation application. The rotation computation requires three inner products of length N plus executing Jacobi Procedure 2. This amounts to $N+4$ multiply-and-add operations per rotation computation, for a total of $(N+4) \cdot M(M-1)/2$ operations. The subsequent rotation application requires two multiply-and-add operations per matrix element, resulting in $2N$ operations per row and $4N \cdot M(M-1)/2$ operations total. Therefore, the total number of multiply-and-add operations during one block sweep is

$$C(M,N) \;=\; (5N+4) \cdot M(M-1)/2.$$

For large M, we approximate the operation count, and express it as a function of σ_M, σ_N and R:

$$C(\sigma_M, \sigma_N, R) \;\approx\; (5\sigma_N R + 4)\sigma_M^2 R^2/2.$$

Next, we determine the number of time steps in units of multiply-and-add operations. To that end, we analyze block transformations. There are $\sigma_M(\sigma_M - 1)/2$ block transformations per block sweep, σ_M of which correspond to the boundary case that rotates the rows within a block. Each block transformation includes a rotation computation and its subsequent application. With the aid of Figures 6 and 7, we determine the number of timesteps of a regular block transformation as follows. The rotation computation of two $R \times N$ row blocks on an $R \times R$ processor array requires $N + 2R + 2$ time steps. N time steps are required to compute the first inner product g on the processor in the top-right corner, plus four more to execute Jacobi Procedure 2. Due to the staggering of the streams, the processor in the bottom-left corner of the array will begin and complete its computation $2(R-1)$ time steps later, yielding a total of $N + 2R + 2$ time steps.

The rotation application of a regular block transformation is more time consuming, because each update involves two multiply-and-add operations on two matrix elements. In Figure 7, we have merged two multiply-and-add operations into one step. Thus, the number of time steps in units of multiply-and-add operations is almost twice of that suggested by counting time steps in Figure 7, which is $2(N + R) + 1$ between the first element entering the top-right processor and the last element exiting the bottom-left processor. Doubling this count with the exception of one time step needed for the last element to exit the array results in a total of $4(N + R) + 1$ time steps.

The time steps for the boundary case can be determined using Figures 8 and 9. The number of time steps is $N+2R+2$ for the rotation computation and $4(N+R)$ for the rotation application. Now, recall that one block sweep consists of $\sigma_M(\sigma_M-1)/2$ block transformations with $\sigma_M(\sigma_M - 3)/2$ regular transformations and σ_M boundary cases. Therefore, the total number of time steps for one block sweep in units of multiply-and-add operations is

$$\begin{aligned} T(\sigma_M, N, R) &= (5N + 6R + 3)\sigma_M(\sigma_M - 3)/2 + (5N + 6R + 2)\sigma_M \\ &= (5N + 6R + 3)\sigma_M^2/2 - (5N + 6R + 5)\sigma_M/2. \end{aligned}$$

We approximate this count and express the number of time steps as a function of σ_M, σ_N, and R:

$$\begin{aligned} T(\sigma_M, \sigma_N, R) &= (5\sigma_N + 6)\sigma_M(\sigma_M - 1)R/2 + \sigma_M(3\sigma_M - 5)/2 \\ &\approx \sigma_M^2(5\sigma_N R + 6R + 3)/2. \end{aligned}$$

The efficiency of one block sweep is the number of operations $C(\sigma_M, \sigma_N, R)$ divided by the product of the number of time steps $T(\sigma_M, \sigma_N, R)$ and the number of processors, including memory processors, $R^2 + 4R$:

$$\begin{aligned} E(\sigma_M, \sigma_N, R) &= \frac{C(\sigma_M, \sigma_N, R)}{T(\sigma_M, \sigma_N, R)(R^2 + 4R)} \\ &\approx \frac{(5\sigma_N R + 4)\sigma_M^2 R^2}{\sigma_M^2(5\sigma_N R + 6R + 3)(R^2 + 4R)} \\ &\approx \frac{\sigma_N}{\sigma_N + 6/5} \cdot \frac{R}{R + 4}. \end{aligned} \tag{9}$$

Equation 9 shows that our stream SVD achieves an optimal 100 % efficiency if both σ_N and R approach infinity. The σ_N-term of Equation 9 assumes value 0.9 for $\sigma_N \approx 10$ and 0.99 for $\sigma_N \approx 100$, that is for problem sizes N about 10 and 100 times the network size R, respectively. The R-term of Equation 9 assumes value 0.9 for network size $R = 36$ and value 0.99 for network size $R = 396$. Consequently, we can expect to achieve nearly 100 % efficiency even for relatively small network and problem sizes in practice. Equation 9 shows furthermore that the efficiency of the stream SVD is independent of M. This fact should not be misinterpreted, however, because the amount of parallelism available to our stream SVD depends directly on M.

5 Experimental Results

We have applied our stream SVD to various matrices in order to analyze numerical stability and convergence behavior. In particular, we were interested in the increase of the number of Hestenes transformations due to an increase in parallelism. In this section, we present the results from three matrices: (1) uniformly random generated square matrices, (2) Golub-Kahan's numerically ill-conditioned square matrix [6], and (3) tall $M \times N$ matrices with uniformly random generated elements that mimic the common use of the SVD for solving least squares problems. The summary of our results is that (a) our stream SVD is numerically stable, and (b) the number of Hestenes transformations increases insignificantly for $R \leq \sqrt{M}$ while generating compute-efficient parallelism for R^2 processors.

We report the number of Hestenes transformations until convergence, using double-precision arithmetic and a value of $\epsilon = 10^{-15}$. The number of Hestenes transformations is independent of whether only singular values are desired or whether matrices U and V are computed as well, because the latter does not change the convergence behavior. Table 1 shows the number of Hestenes transformations for uniformly random generated square matrices. Each column corresponds to a problem size N, and each row to a network size R. As expected, the number of transformations increases with R. The step line through the table marks the boundary $R = \sqrt{N}$. We observe that the number of transformations increases by up to about 20 % for $R \approx \sqrt{N}$ compared to the sequential cyclic-by-rows Hestenes-Jacobi method for $R = 1$. Since we may use up to R^2 processors compute-efficiently, we can boost the performance of a 16×16 SVD with 16 processors ($R = 4$) by a factor of nearly 16, modulo a 10 % increase of work from 960 to $1,080$ transformations. If we use $1,024$ processors ($R = 32$) for a $2,048 \times 2,048$ SVD, the speedup will be very close to $1,024$ at the expense of increasing the number of transformations by 12.5 %.

$R\backslash N$	16	32	64	128	256	512	1,024	2,048
1	960	4,464	20,160	97,536	391,680	1,700,608	7,332,864	33,538,048
2	960	4,464	20,160	97,536	391,680	1,831,424	7,856,640	33,538,048
4	1,080	4,960	22,176	97,536	424,320	1,962,240	7,856,640	33,538,048
8	1,440	5,456	24,192	121,920	456,960	1,831,424	7,856,640	33,538,048
16	2,040	8,928	30,240	113,792	489,600	2,093,056	8,904,192	35,634,176
32		28,768	88,704	243,840	750,720	2,485,504	9,427,968	37,730,304
64			395,136	1,853,184	5,940,480	15,305,472	47,663,616	127,863,808
128				35,259,264	84,178,560	490,167,552	2,817,914,880	14,306,073,600

Table 1: Number of Hestenes transformations for uniformly random generated $N \times N$ matrices, and varying N and network size R. The step line marks $R = \sqrt{N}$.

Lacking a conclusive model for the convergence behavior, we observe that the number of Hestenes transformations grows linearly with R up to $R = \sqrt{N}$ for a fixed problem size N. For $R > \sqrt{N}$, the number of transformations grows exponentially. This behavior is analogous to the transfer time of packets in a network, where congestion leads to an exponential increase in the transfer time beyond a certain traffic threshold. In case of the SVD, it appears to be the case that for larger values of R, the additional transformations do not transfer any information relevant to the orthogonalization process across the matrix. Thus, for $R > \sqrt{N}$ the majority of Hestenes transformations is redundant.

$R \backslash N$	16	32	64	128	256	512	1,024	2,048
1	960	4,464	20,160	81,280	391,680	1,962,240	8,380,416	39,826,432
2	960	4,960	24,192	105,664	456,960	2,093,056	9,427,968	37,730,304
4	1,080	4,960	24,192	105,664	456,960	2,093,056	9,427,968	37,730,304
8	1,560	5,952	26,208	113,792	522,240	2,223,872	9,951,744	41,922,560
16	1,800	8,432	34,272	121,920	554,880	2,616,320	9,951,744	44,018,688
32		17,856	70,560	203,200	685,440	2,485,504	10,475,520	48,210,940
64			233,856	609,600	2,023,680	5,232,640	12,046,848	54,499,328
128				2,787,904	11,554,560	43,823,360	142,990,848	146,728,960

Table 2: Number of Hestenes transformations for $N \times N$ Golub-Kahan matrices, and varying N and network size R. The step line marks $R = \sqrt{N}$.

Table 2 shows the number of Hestenes transformations for numerically ill-conditioned matrices. The Golub-Kahan matrix is an upper triangular $N \times N$ matrix [6, p. 206], which stresses the numerical accuracy of an SVD algorithm, because it has one exceptionally small and a couple of very large singular values. Furthermore, the condition deteriorates as N increases. The Golub-Kahan matrix is defined as

$$(a_{ij}) = \begin{cases} 1, & i = j \\ -1, & i < j \\ 0, & i > j. \end{cases}$$

Our stream Hestenes algorithm produces the singular values with reasonable accuracy (compared to various other SVD implementations), including the exceptionally small singular value. We observe that the convergence behavior resembles that of Table 1. The number of Hestenes transformations appears to be slightly larger for each of the experiments than for the numerically better behaved random generated matrices.

$R \backslash M$	128	256	512	1,024	2,048	4,096	8,192	16,384
1	0.098	0.326	1.3	5.3	21.0	83.9	335.5	1,342.1
2	0.098	0.457	1.4	5.8	25.2	92.3	369.1	1,342.1
4	0.098	0.490	1.4	6.8	23.1	100.6	402.6	1,610.5
8	0.122	0.620	2.2	7.9	31.4	109.0	503.3	2,013.1
16	0.114	0.849	2.3	8.9	37.7	134.2	671.0	2,147.4
32	0.243	1.240	2.9	11.0	39.8	151.0	536.8	2,818.4
64	1.853	2.056	3.9	12.0	41.9	151.0	469.7	2,013.1
128	35.259	8.617	9.5	17.8	52.4	176.1	536.8	2,415.8

Table 3: Number of Hestenes transformations (**in millions**) for uniformly random generated $M \times N$ matrices with $N = 128$, and varying M and network size R. The step line marks $R = \sqrt{M}$.

Table 3 shows the number of Hestenes transformations for $M \times N$ matrices with varying M and fixed $N = 128$. We observe that the exponential increase of rotations vanishes when M becomes large relative to N. However, we also notice that the number of Hestenes transformations increases by up to about 100 % for $R = \sqrt{M}$ compared to a 10 % increase for square matrices. This observation is consistent with our earlier interpretation that a number of Hestenes transformations within a single block transformation are redundant because they do not propagate relevant information.

Finally, we note that our stream SVD exhibits a different behavior than the data-oblivious stream algorithms that we have designed earlier [12]. Those stream algorithms are compute efficient, that is approach 100 % floating-point efficiency asymptotically for large numbers of processors, if $R \lesssim cM$. Typically, a value of $c \lesssim 0.2$ suffices to amortize the startup cost of the systolic pipelines. In contrast, experimental evidence suggests that our stream SVD is only efficient for $R \lesssim \sqrt{M}$ since the number of Hestenes transformations increases significantly otherwise.

Using asymptotic notation, we may characterize the relationship between the network size R and problem size N as follows. The stream algorithms designed in [12] are compute efficient if $R = O(N)$, that is if R has an asymptotic upper bound directly proportional to N. For the stream SVD we find that $R = O(\sqrt{N})$, that is R has an upper bound proportional to $\sqrt{N}$. As a different perspective on the relationship between network and problem size, we may compare the number of processors P with the problem size N. For the SVD, we may use up to $P = R^2 = O(N)$ processors efficiently. In contrast, other stream algorithms suited for use on a two-dimensional array, including matrix multiplication, triangular solver, LU and QR decomposition may use up to $P = R^2 = O(N^2)$ processors efficiently. Therefore, the amount of parallelism exhibited by our stream SVD is one polynomial degree smaller than that of the other applications. This raises the question whether we can find a stream algorithm for the Jacobi method, for example, that creates parallelism on the order $R = O(N)$ rather than $R = O(\sqrt{N})$.

Acknowledgments

This work has been funded as part of the Raw project by DARPA, NSF, the Oxygen Alliance, MIT Lincoln Laboratory and the Lincoln Scholars Program.

References

[1] Richard P. Brent and Franklin T. Luk. The Solution of Singular-Value and Symmetric Eigenvalue Problems on Multiprocessor Arrays. *SIAM Journal on Scientific and Statistical Computing*, 6(1):69–84, January 1985.

[2] Richard P. Brent, Franklin T. Luk, and Charles F. Van Loan. Computation of the Singular Value Decomposition Using Mesh-Connected Processors. *Journal of VLSI and Computer Systems*, 1(3):242–260, 1985.

[3] Bruce A. Chartres. Adaption of the Jacobi Method for a Computer with Magnetic-tape Backing Store. *The Computer Journal*, 5(1):51–60, April 1962.

[4] Patricia J. Eberlein. A Jacobi-like Method for the Automatic Computation of Eigenvalues and Eigenvectors of Arbitrary Matrix. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):74–88, March 1962.

[5] George E. Forsythe and Peter Henrici. The Cyclic Jacobi Method for Computing the Principal Values of a Complex Matrix. *Transactions of the American Methematical Society*, 94(1):1–23, January 1960.

[6] Gene H. Golub and William Kahan. Calculating the Singular Values and Pseudoinverse of a Matrix. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, 2(2):205–224, 1965.

[7] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. John Hopkins University Press, Baltimore and London, 2nd edition, 1993.

[8] Gene H. Golub and Christian Reinsch. Singular Value Decomposition and Least Square Solutions. In J. H. Wilkinson and C. Reinsch, editors, *Linear Algebra*, volume II of *Handbook for Automatic Computations*, chapter I/10, pages 134–151. Springer Verlag, 1971.

[9] Eldon R. Hansen. On Cyclic Jacobi Methods. *Journal of the Society for Industrial and Applied Mathematics*, 11(2):448–459, June 1963.

[10] Peter Henrici. On the Speed of Convergence of Cyclic and Quasicyclic Jacobi Methods for Computing Eigenvalues of Hermitian Matrices. *Journal of the Society for Industrial and Applied Mathematics*, 6(2):144–162, June 1958.

[11] Magnus R. Hestenes. Inversion of Matrices by Biorthogonalization and Related Results. *Journal of the Society for Industrial and Applied Mathematics*, 6(1):51–90, March 1958.

[12] Henry Hoffmann, Volker Strumpen, and Anant Agarwal. Stream Algorithms and Architecture. Technical Memo MIT-LCS-TM-636, Laboratory for Computer Science, Massachusetts Institute of Technology, March 2003.

[13] Carl G. J. Jacobi. Über eine neue Auflösungsart der bei der Methode der kleinsten Quadrate vorkommenden linearen Gleichungen. *Astronomische Nachrichten*, 22, 1845. *English translation by G.W. Stewart, Technical Report 2877, Department of Computer Science, University of Maryland, April 1992.*

[14] Carl G. J. Jacobi. Über ein leichtes Verfahren, die in der Theorie der Säkularstörungen vorkommenden Gleichungen numerisch aufzulösen. *Journal für die Reine und Angewandte Mathematik*, 30:51–95, 1846.

[15] Franklin T. Luk. Computing the Singular-Value Decomposition on the ILLIAC IV. *ACM Transactions on Mathematical Software*, 6(4):524–539, December 1980.

[16] Franklin T. Luk. A Triangular Processor Array for Computing Singular Values. *Linear Algebra and Its Applications*, 77:259–273, 1986.

[17] J. C. Nash. A One-sided Transformation Method for the Singular Value Decomposition and Algebraic Eigenproblem. *The Computer Journal*, 18(1):74–76, February 1975.

[18] Heinz Rutishauser. The Jacobi Method for Real Symmetric Matrices. In J. H. Wilkinson and C. Reinsch, editors, *Linear Algebra*, volume II of *Handbook for Automatic Computations*, chapter II/1, pages 202–211. Springer Verlag, 1971.

[19] Robert Schreiber. On the Systolic Arrays of Brent, Luk, and Van Loan. In Keith Bromley, editor, *Real Time Signal Processing VI*, volume 431, pages 72–76, San Diego, CA, August 1983. SPIE.

[20] G. W. Stewart. A Jacobi-like Algorithm for Computing the Schur Decomposition of a Nonhermitian Matrix. *SIAM Journal on Scientific and Statistical Computing*, 6(4):853–864, October 1986.

[21] James H. Wilkinson. Note on the Quadratic Convergence of the Cyclic Jacobi Process. *Numerische Mathematik*, 4:296–300, 1962.