Blocked In-Place Transposition with Application to Storage Format Conversion*

Lars Karlsson†
larsk@cs.umu.se

UMINF 09.01

Department of Computing Science
Umeå University and HPC2N
S-901 87 Umeå, Sweden

January 26, 2009

Abstract

We develop a prototype library for in-place (dense) matrix storage format conversion between the canonical row and column-major formats and the four canonical block data layouts. Many of the fastest linear algebra routines operate on matrices in a block data layout. In-place storage format conversion enables support for input/output of large matrices in the canonical row and column-major formats. The library uses algorithms associated with in-place transposition as building blocks. We investigate previous work on the subject of (in-place) transposition and the most promising algorithms are implemented and evaluated. Our results indicate that the Three-Stage Algorithm which only requires a small constant amount of additional memory performs well and is easy to tune. Murray Dow's V5 algorithm, which is a two-stage semi-in-place algorithm that requires a small amount of additional memory is sometimes a better choice. The write-allocate strategy of most cache-based computer architectures appears to be the cause of an observed performance problem for large matrices.

*This research was conducted using the resources of High Performance Computing Center North (HPC2N).

†Funded in part by the VR grant 70625701. The work is also a part within an IBM Shared University Research (SUR) grant.

Contents

1 Introduction

We develop a library for in-place matrix storage format conversion based on in-place transposition algorithms. In-place transposition is a well-studied problem [1, 15, 5, 4, 3, 2, 6, 14, 11]. Nonetheless, the growing gap between CPU processing speed and memory bandwidth/latency unfortunately means that most of the early algorithms for pure in-place transposition take one or more orders of magnitude longer to execute than an out-of-place algorithm. The main reason is that these in-place algorithms move individual elements that are not contiguous in memory, thereby severely stressing the memory hierarchy. Several algorithms address these issues [1, 5, 6, 14] but some are only semi-in-place and require a relatively small but still non-constant amount of additional memory.

In-place transposition has applications in FFT algorithms [9, 8] and to convert between the Column-Major (CM) and Row-Major (RM) canonical matrix storage formats. However, our main motivation is to provide a software package for fast in-place conversion between the canonical CM and RM formats and various block data layouts (see Sections 2 and 6). Such block formats are often used instead of CM/RM in linear algebra kernels to improve data locality. Conversion is required in order to support the familiar storage formats at the interface level while internally working with block data layouts.

The paper is structured as follows. In Section 2, we recall the canonical storage formats (CM/RM) and four canonical block storage formats. After discussing storage formats, we continue by recalling some of the previously published techniques for in-place transposition in Section 3. We focus on algorithms that require only a few sweeps through the matrix since memory bandwidth is the limiting factor. The connection to matrix-vector multiplication and Kronecker products is reviewed in Section 4, and in Section 5, we illustrate how some types of permutations can be implemented using in-place transposition algorithms. Implementations of these permutations are used as building blocks for our storage format conversion library. Conversion between storage formats is further discussed in Section 6 followed by a detailed description of the so-called Three-Stage Algorithm for in-place transposition in Section 7. The Three-Stage Algorithm has been previously mentioned in the literature [10, 14], but to our knowledge it has not been carefully compared with other algorithms. We introduce the conversion library in Section 8, followed by computational experiments in Section 9 and conclusions in Section 10. Finally, in Appendix A we give details on how to re-formulate some previously published algorithms using the notation developed in this paper.

We consistently use a *zero-origin indexing* convention, meaning that an $m \times n$ matrix has m rows numbered $0, \dots, m-1$ and n columns numbered $0, \dots, n-1$. The top left element of the matrix is consequently $A(0,0)$ and the first storage location is 0. We interchangeably use *(storage) format* and *data layout* to denote the scheme by which a matrix is stored in memory.

2 Matrix Storage Formats

2.1 Canonical Formats

The two canonical storage formats typically used by compilers are the Row-Major (RM) and Column-Major (CM) data layouts. Using either of these formats, element $A(i, j)$ of the $m \times n$ matrix A is stored at location

$$\begin{array}{ll} i + jm & \text{(CM), or} \\ in + j & \text{(RM).} \end{array}$$

Figure 1 is an example of a 9×6 matrix in CM format. The elements are

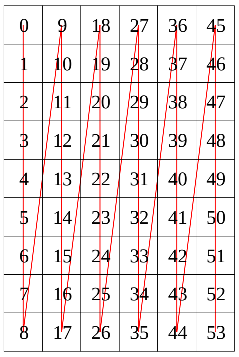


Figure 1: A 9×6 matrix in CM format.

numbered according to their location in memory. We use a polyline to highlight the storage order of the elements. In all examples, the start of the sequence is in the top left corner and the end is in the bottom right corner. In this particular example, the sequence begins with $0, 1, \ldots$ and ends with $\ldots, 52, 53$.

There is a strong connection between the CM/RM formats and matrix transposition. For example, if A is stored in RM format it is indistinguishable from A^T stored in CM format. Hence, transposing A in CM format is the same as converting it into RM format and vice versa.

2.2 Block Formats

It has long been understood that the CM and RM formats are suboptimal for a large class of algorithms, including most of linear algebra, FFT, image analysis, and more. The reason is that spatial data locality is only maintained within columns (CM) or rows (RM), whereas many algorithms need locality in both dimensions. For example, element $A(i, j)$ and $A(i + 1, j)$ are close in CM (stride 1) but $A(i, j)$ and $A(i, j + 1)$ are far apart (stride m). *Block formats* bring elements within certain submatrices (known as blocks) closer together. Accessing a submatrix in a block format typically provides more spatial data locality than accessing the same submatrix in CM or RM format [17].

Among the many proposed hybrid data layouts, the canonical block formats have most of the benefits of hybrid data layouts (e.g., see [7, 17]) while keeping a simple mapping from element to storage location (the so called *storage mapping*). Assume that an $m \times n$ matrix is partitioned into an $M \times N$ block matrix with blocks of size $m_b \times n_b$ and that each block is stored contiguously in memory. Typically, $m = Mm_b$ and $n = Nn_b$. We call such a data layout a *block format* and by choosing CM or RM as the storage format for the blocks and CM or RM as the storage format for the elements inside each block we get the four *canonical* block formats: CCRB, CRRB, RCRB, and RRRB. The suffix RB is an acronym for Rectangular Block, the first letter indicates the storage format used for the blocks and the second letter indicates the storage format for elements inside a block. For example, the RCRB format stores the blocks in RM format while the elements inside each block are stored in CM format.

The element $A_{i_1,j_1}(i_2, j_2)$ of the block matrix

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,N-1} \\ \vdots & \ddots & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,N-1} \end{bmatrix}$$

denotes the (i_2, j_2)-element of the (i_1, j_1)-block. It is the same as $A(i_1 m_b + i_2, j_1 n_b + j_2)$ and is stored at one of the following locations:

$$\begin{array}{cl} (j_1 M + i_1) n_b m_b + (j_2 m_b + i_2) & \text{(CCRB)} \\ (j_1 M + i_1) m_b n_b + (i_2 n_b + j_2) & \text{(CRRB)} \\ (i_1 N + j_1) n_b m_b + (j_2 m_b + i_2) & \text{(RCRB)} \\ (i_1 N + j_1) m_b n_b + (i_2 n_b + j_2) & \text{(RRRB)} \end{array}$$

Figure 2 illustrates a 9×6 matrix (the same matrix as in Figure 1) stored in CCRB format with blocks of size 3×2.

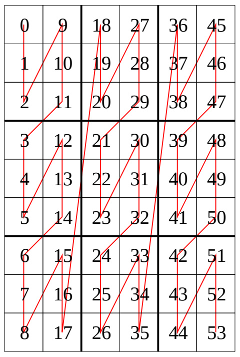


Figure 2: A 9×6 matrix in CCRB format.

3 Basics of In-Place Transposition

3.1 Algorithms for Square Matrices

Square $n \times n$ matrices are easier to transpose in-place than rectangular matrices. We describe some well-known techniques related to square in-place transposition [5].

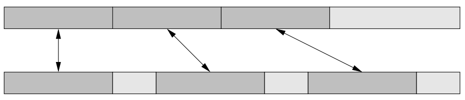
3.1.1 Basic Algorithm

Element $A(i, j)$ of a square $n \times n$ matrix in CM format is moved from location $i + jn$ to location $in + j$. Similarly, element $A(j, i)$ moves from location $in + j$ to location $i + jn$. Therefore, the diagonal elements are not moved at all, whereas the off-diagonal elements $A(i, j)$ and $A(j, i)$ are swapped. Cache blocking must be used to get good performance since in a naive implementation at least one matrix is accessed with a large stride.

3.1.2 Pad Transpose

If a matrix is nearly square, we can pad the matrix so that it becomes square and apply any square in-place transposition algorithm [5]. This is only practical if we can use $|m - n| \cdot \max(m, n)$ storage locations directly following the matrix in memory (something which is impossible in many codes).

```
Algorithm: PACK
for j = 1 to n-1
   for i = 0 to m-1
      [i+j*m] = [i+(j+x)*m]
```



```
Algorithm: UNPACK
for j = n-1 downto 1
   for i = m-1 downto 0
      [i+(j+x)*m] = [i+j*m]
```

Figure 3: Illustration of the PACK and UNPACK algorithms used to implement pad and cut transposes.

The notation $[x]$ in Figure 3 refers to the element at storage location x.

- If $m > n$, pad with $x = m - n$ columns. After transposition, the padded elements make up the x last rows of the matrix and are hence scattered in memory. By applying the `PACK` algorithm (Figure 3) the padded elements are removed and only the transposed original matrix remains.
- If instead $n > m$, pad with $x = n - m$ rows by applying the `UNPACK` algorithm (Figure 3). After transposition, the padded elements make up the x last columns of the matrix and can be safely ignored.

Besides having to transpose a slightly larger matrix, the pad transpose also requires an additional sweep through the matrix.

3.1.3 Cut Transpose

By removing instead of adding elements, the requirement on additional storage directly following the matrix is not mandatory. This technique is called a cut transpose [5].

- If $m > n$, cut away $x = m - n$ rows by applying the `PACK` algorithm (first making sure to save the cut-off elements). After transposition, the cut-off elements make up the x last columns and can be copied back to their correct locations.

- If instead $n > m$, cut away $x = n - m$ columns and save the cut-off elements. After transposition, the cut-off elements make up the x last rows and room for them is created by the `UNPACK` algorithm. After unpacking, the cut-off elements can be copied back to their correct locations.

The primary cost of a cut transpose is the extra sweep through the matrix.

3.2 Algorithms that Follow Cycles

Element $A(i, j)$ is stored at location $k = i + jm$ and after transposition it has moved to location $\bar{k} = in + j$. There is a simple form for the mapping from k to $\bar{k}$:

$$\bar{k} = P(k) = \begin{cases} kn \mod M & \text{if } 0 \le k < M, \\ M & \text{if } k = M, \end{cases} \tag{1}$$

where $M = mn - 1$ [3, 4]. The inverse mapping turns out to be more useful in practice and it can be shown that

$$k = P^{-1}(\bar{k}) = \begin{cases} \bar{k}m \mod M & \text{if } 0 \le \bar{k} < M, \\ M & \text{if } \bar{k} = M. \end{cases} \tag{2}$$

Transposition is a permutation and every permutation can be factored into a product of disjoint cycles. Due to the special structure of P^{-1} a cycle starting at s has a *companion cycle* (or sometimes *dual cycle*) starting at $M - s$ [3]. In some cases, the two cycles coincide and s is said to be *self-dual.* A *cycle leader* is any unique representative of a cycle (e.g., its minimum element). For example, the cycle factorization of P^{-1} for the 5×3 transposition problem is

$$(0)(1\ \ 5\ \ 11\ \ 13\ \ 9\ \ 3)(7)(2\ \ 10\ \ 8\ \ 12\ \ 4\ \ 6)(14).$$

The cycle leaders (the minimum elements) are 0, 1, 2, 7, and 14. There are three singleton cycles: 0, 7, and $14 = M$ and two self-dual cycles. The first cycle has leader $s_1 = 1$ and companion leader $M - s_1 = 13$, while the other cycle has leader $s_2 = 2$ and companion leader $M - s_2 = 12$. Cycle-following algorithms shift the elements of each cycle and previous research has focused on how to reduce the overhead of finding the cycle leaders. There is basically no spatial data locality when shifting a cycle, a fact that can be partly appreciated by observing that the definition of P^{-1} is similar to that of a Park-Miller linear congruential random number generator. Therefore, previously published cycle-following algorithms are of little practical interest on today's computer architectures with deep memory hierarchies.

A single cycle is shifted efficiently by Algorithm 1 which uses the inverse mapping P^{-1}. The notation $[x]_y$ is used to denote the vector of y contiguous elements starting at memory location xy (compare with Figure 3).

It turns out that self-dual cycles always meet in the middle [4, Theorem 7]. In other words, if one starts at s and $M - s$ and simultaneously traverses both cycles, then one will arrive at $M - s$ outgoing from s at the same step that one arrives at s outgoing from $M - s$. If the cycles are not self-dual, then one will complete their cycles at the same step since they have the same length. This symmetry result has been used in [3, 4, 11] to efficiently shift both cycles simultaneously. See Algorithm 2 for an implementation.

Algorithm 1 Cycle Shifting

```
Input: The cycle leader $s$ and the vector length $L$.
 1: $a_1 := s$
 2: $t := [a_1]_L$
 3: $a_2 := P^{-1}(a_1)$
 4: while $a_2 \neq s$ do
 5:     $[a_1]_L := [a_2]_L$
 6:     $a_1 := a_2$
 7:     $a_2 := P^{-1}(a_1)$
 8: end while
 9: $[a_1]_L := t$
```

Algorithm 2 Simultaneous Cycle and Companion Cycle Shifting

```
Input: The cycle leader $s$, $M = mn - 1$, and the vector length $L$.
 1: $a_1 := s$
    $\hat{a}_1 := M - s$
 2: $t := [a_1]_L$
    $\hat{t} := [\hat{a}_1]_L$
 3: $a_2 := P^{-1}(a_1)$
    $\hat{a}_2 := M - a_2$
 4: loop
 5:     if $a_2 = s$ then
 6:         The cycle and its companion are distinct.
 7:         $[a_1]_L = t$
            $[\hat{a}_1]_L = \hat{t}$
 8:         break
 9:     end if
10:     if $\hat{a}_2 = s$ then
11:         The cycle is self-dual.
12:         $[a_1]_L = \hat{t}$
            $[\hat{a}_1]_L = t$
13:         break
14:     end if
15:     $[a_1]_L := [a_2]$
        $[\hat{a}_1]_L := [\hat{a}_2]$
16:     $a_1 := a_2$
        $\hat{a}_1 := \hat{a}_2$
17:     $a_2 := P^{-1}(a_1)$
        $\hat{a}_2 := M - a_2$
18: end loop
```

One approach to finding the cycle leaders is to scan through the elements and use a boolean table with mn entries to record which elements have been moved. The cycle leader test reduces to a table lookup. This approach is memory intensive for floating point matrices unless the table can be embedded into an unused bit in each element.

An approach which does not require any additional memory uses the minimum element as the cycle leader. For each possible cycle leader s in the sequence $0, \ldots mn-1$, traverse its cycle until either $t < s$ is encountered (in which case s is rejected) or s is encountered again, completing the cycle and showing that s is the minimum element in its cycle. The computational cost of this approach is significant and makes this approach impractical. However, the idea to traverse the cycle to find its minimum element is useful and is called the *general cycle test* in what follows.

Brenner [3] used number theory results to study the transposition permutation. He showed that all elements in a cycle starting at s are divisible by $d = \gcd(s, M)$ and not divisible by any other larger divisor of M [3, Theorem 1]. We associate all $\phi(M/d)$ such elements with d, where ϕ is Euler's phi function. For each divisor d of M, successively larger multiples of d are considered as possible cycle leaders until all $\phi(M/d)$ elements associated with d have been shifted. Experience has showed that Brenner's algorithm can greatly reduce the overhead of finding cycle leaders. We have adopted Brenner's results as the basis for our implementation.

In practice, cycle-following algorithms are hybrid methods that use a boolean table of limited size, typically with only $(m+n)/2$ entries. The table covers the first few possible cycle leaders. For larger candidates the general test is used. Experience indicates that the transposition often completes before the table is overrun.

ACM Algorithm 302 [2] can also be categorized as a cycle-following algorithm. However, it uses a fundamentally different algorithm for shifting cycles. The algorithm does not appear to be as efficient as either ACM Algorithm 467 or ACM Algorithm 513 [4].

3.3 Variants of Eklundh's Algorithm

Eklundh [6] developed an algorithm for transposing large square $2^n \times 2^n$ matrices out-of-core. His method uses only a small amount of additional in-core memory. The general idea starts with a 2×2 block partitioning:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

After transposing each block in-place recursively the storage contains the matrix

$$\begin{pmatrix} A_{11}^T & A_{12}^T \\ A_{21}^T & A_{22}^T \end{pmatrix}.$$

To complete the transposition the blocks A_{12}^T and A_{21}^T are swapped. Eklundh pointed out that the entire process can be performed from the simple building block of reading two rows into core memory, swapping some elements, and writing the two rows back in the same place. He combined this simple operation with some intricate index manipulations and arrived at an ingenious non-recursive implementation.

Eklundh's algorithm has since been extended to rectangular matrices and more general composite dimensions in [18]. Variants of Eklundh's algorithm appear to require a larger number of sweeps than cycle-following algorithms and is therefore unlikely to be competitive when the matrix fits in main memory. See [14] for more variants of Eklundh's algorithm.

3.4 Other Algorithms

Murray Dow reviewed several transposition techniques in [5], including the pad and cut transposes. Two block algorithms suitable for vector computers were also presented. The first algorithm (V4), which Dow attributes to Markus Hegland, applies when $m = Mm_b$. The matrix is partitioned into an $M \times n$ block matrix with blocks of size $m_b \times 1$. The blocks are first transposed and then their elements are reordered to complete the transposition. For an example, see [5, Algorithm V4] or Appendix A.

Dow's second block algorithm (V5) partitions both dimensions and applies when $D \equiv \gcd(m, n) > 1$ [5, Algorithm V5]. The dimensions are factored into $m = Dm_b$ and $n = Dn_b$ and partitions the matrix into a square $D \times D$ block matrix with blocks of size $m_b \times n_b$. The first step of the algorithm transposes each block. This is reported to require mn_b additional memory locations. The second step transposes the block matrix using a square in-place transpose algorithm requiring no additional storage.

A three-stage transposition algorithm is presented by Alltop in [1]. It also factors $m = Dm_b$ and $n = Dn_b$ and partitions the matrix into a $D \times D$ block matrix with blocks of size $m_b \times n_b$. The first step transposes the square $D \times D$ block matrix. The second and third steps taken together transpose the individual blocks and are implemented by out-of-place rectangular transpositions using $Dn_bm_b = nm_b$ and $Dn_b = n$ additional elements, respectively.

4 Transposition as Matrix-Vector Multiplication

The Kronecker product $A \otimes B$ of the $m \times n$ matrix A and the $p \times q$ matrix B is an $mp \times nq$ matrix with $a_{ij}B$ as its (i, j)-th element. The vec operator forms a vector by stacking the columns of a matrix underneath eachother [12]. Thus with $a_{\bullet j}$ denoting the j-th column of the matrix A

$$\operatorname{vec} A = \begin{bmatrix} a_{\bullet 0} \\ \vdots \\ a_{\bullet n-1} \end{bmatrix}.$$

The order of the elements corresponds to the layout in memory when A is stored in CM format. Transposing A amounts to permuting $\operatorname{vec} A$ into $\operatorname{vec} A^T$ by multiplication with a permutation matrix

$$\operatorname{vec} A^T = L_m^{n \cdot m} \operatorname{vec} A.$$

The so-called *vec-permutation* matrix $L_m^{n \cdot m}$ is $nm \times nm$ and permutes an nm-vector by taking every m-th element of the vector starting with the first, then every m-th element starting with the second, and so on. An alternative definition of $L_m^{n \cdot m}$ is as the permutation matrix which verifies

$$L_m^{n \cdot m}(e_j^n \otimes e_i^m) = e_i^m \otimes e_j^n,$$

where, for example, e_i^n denotes the i-th unit vector (counting from zero) of length m. Yet another definition is constructive:

$$L_m^{n \cdot m} = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} e_i^m (e_j^n)^T \otimes e_j^n (e_i^m)^T.$$

The vec-permutation matrix $L_3^{5 \cdot 3}$ is illustrated in Figure 4. For a review of

$$\left[\begin{array}{ccc|ccc|ccc|ccc|ccc}
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\hline
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\hline
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1
\end{array}\right]$$

Figure 4: The vec-permutation matrix $L_3^{5 \cdot 3}$. The dot-elements are zeroes.

the history and properties of the Kronecker product and the vec-permutation matrix in particular see [12].

5 Blocked Transposition

With blocked transposition we consider algorithms that primarily read and write contiguous storage locations. In this section, we explain how certain permutations can be implemented with in-place transposition algorithms (adapted to move contiguous vectors) as building blocks.

We follow the approach of Fraser [8] and others and view storage locations in a *mixed-radix number system* and consider digit permutations. We use the notation $[r_1, r_0](d_1, d_0)$ to specify the radices r_1 and r_0 as well as the digits d_1 and d_0 in a mixed-radix number system. Another way to specify mixed-radix numbers is to subscript each digit with its radix. We stick to the former notation since it separates radices from digits.

For numbers with four positions, which is primarily what we are using, the definition of a mixed-radix number is

$$[r_3, r_2, r_1, r_0](d_3, d_2, d_1, d_0) = d_3 r_2 r_1 r_0 + d_2 r_1 r_0 + d_1 r_0 + d_0, \tag{3}$$

where the radices are greater than one ($r_i > 1$) and $d_i \in \{0, \ldots, r_i - 1\}$. With these conditions, every decimal number $x \in \{0, \ldots, r_3 r_2 r_1 r_0 - 1\}$ has a unique representation. Note that $r_3 = r_2 = r_1 = r_0 = 10$ corresponds to our decimal number system. The example below also illustrates an alternative way of presenting mixed-radix numbers using subscripted radices:

$$[4, 6, 5, 7](2, 5, 3, 4) = 2_4 5_6 3_5 4_7 = 2 \cdot (6 \cdot 5 \cdot 7) + 5 \cdot (5 \cdot 7) + 3 \cdot (7) + 4 = 620.$$

Consider again the block matrix

$$A = \begin{bmatrix} A_{0,0} & \cdots & A_{0,N-1} \\ \vdots & \ddots & \vdots \\ A_{M-1,0} & \cdots & A_{M-1,N-1} \end{bmatrix}$$

where each block is of size $m_b \times n_b$. The element $A_{i_1,j_1}(i_2, j_2)$ is stored (assuming CM format) in the storage location with the mixed-radix number representation

$$[N, n_b, M, m_b](j_1, j_2, i_1, i_2) = (j_1 n_b + j_2) M m_b + (i_1 m_b + i_2). \tag{4}$$

If the same element instead was stored at the location

$$[N, M, n_b, m_b](j_1, i_1, j_2, i_2) = (j_1 M + i_1) n_b m_b + (j_2 m_b + i_2), \tag{5}$$

then the matrix A would have been stored in the CCRB storage format.

There is a connection between mixed-radix numbers and certain vectors built by Kronecker products. The number in (4) gives the position of the only non-zero element in the vector

$$e^N_{j_1} \otimes e^{n_b}_{j_2} \otimes e^M_{i_1} \otimes e^{m_b}_{i_2}. \tag{6}$$

Multiplying (6) with a particular permutation matrix

$$(I_N \otimes L^{n_b \cdot M}_M \otimes I_{m_b})(e^N_{j_1} \otimes e^{n_b}_{j_2} \otimes e^M_{i_1} \otimes e^{m_b}_{i_2}) = e^N_{j_1} \otimes e^M_{i_1} \otimes e^{n_b}_{j_2} \otimes e^{m_b}_{i_2}$$

commutes the two vectors in the middle of the Kronecker product (6) and the position of the only non-zero element of the resulting vector is now given by the number in (5).

Many algorithms can be expressed in terms of factorizations of the vec-permutation $L^{n \cdot m}_m$, including most algorithms discussed in this paper. See [14] (and also [13]) for an extensive study on the subject with many existing and some new algorithms and their relations to factorizations of the vec-permutation.

5.1 In-Place Blocked Transposition

In this section, we investigate three classes of permutations and their implementation with in-place transposition algorithms as building blocks. We show that swapping two adjacent digits can be implemented by a set of independent in-place transpositions that move contiguous vectors. Furthermore, we also show that swapping two non-adjacent digits can be implemented by a set of independent *square* in-place transpositions if the radices of the two digits are equal. Finally, we show how to fuse two adjacent digit swaps together provided they operate on different digits. This allows for an implementation which sweeps through the matrix once instead of two times.

We start by looking at swapping two adjacent digits d_k in (3). To indicate which digits we intend to swap we will make use of a so-called *transposition pattern.* The pattern $(j, i, \cdot, \cdot)$, for example, simply indicates that we intend to swap the third and fourth digits. A pattern for swapping two adjacent digits is called *regular.* Without loss of generality we apply the pattern $(\cdot, j, i, \cdot)$ to storage locations expressed in the mixed-radix number system

$$[r_3, r_2, r_1, r_0](d_3, \mathbf{j}, \mathbf{i}, d_0). \tag{7}$$

All elements addressed by $\mathbf{i} \in \{0, \ldots, r_1 - 1\}$ and $\mathbf{j} \in \{0, \ldots, r_2 - 1\}$ (keeping d_3 and d_0 fixed) are interpreted as an *embedded matrix* of size $r_1 \times r_2$. There are $d_3 d_0$ such embedded matrices in the example above, one for each choice of d_3 and d_0. We use this interpretation to highlight that swapping two digits can be implemented by transposing all of the embedded matrices in-place.

In the example above, we wish to permute the elements so that the element at location (7) is moved to location

$$[r_3, r_1, r_2, r_0](d_3, \mathbf{i}, \mathbf{j}, d_0). \tag{8}$$

Equation (7), which is the storage location *before* permutation, expands to

$$d_3 r_2 r_1 r_0 + d_0 + (\mathbf{j} r_1 + \mathbf{i}) r_0, \tag{9}$$

and (8), the storage location *after* permutation, expands to

$$d_3 r_2 r_1 r_0 + d_0 + (\mathbf{i} r_2 + \mathbf{j}) r_0. \tag{10}$$

The only difference is in the parenthesized expressions. Notice the connection between the parenthesized expressions and the CM storage mapping of the associated $r_1 \times r_2$ embedded matrix (call it B). The expression $(\mathbf{j} r_1 + \mathbf{i})$ in (9) is the location of $B(\mathbf{i}, \mathbf{j})$ prior to transposition and the expression $(\mathbf{i} r_2 + \mathbf{j})$ in (10) is the location of the same element after transposition. The key point is that we can implement an adjacent digit swap by adapting the memory references of any (in-place or out-of-place) transposition algorithm and repeat the algorithm for every choice of the digits d_3 and d_0. Moreover, from (10) we make the following observation. With d_3, $\mathbf{i}$, and $\mathbf{j}$ given, the elements corresponding to all choices of d_0 are contiguous and maintain their relative order after the permutation. Therefore, we can move all r_0 such contiguous elements at once. Figure 5 is an illustration of an embedded matrix associated with the regular pattern $(\cdot, j, i, \cdot)$ when we interpret the matrix as being block-partitioned and in CM format. In summary, swapping two adjacent digits amounts to a set of independent (in-place) transpositions that move contiguous elements.

A pattern associated with swapping two non-adjacent digits is called *separated.* An example of a separated pattern is $(\cdot, j, \cdot, i)$ which swaps the first and the third digits. We use the mixed-radix number representation

$$[r_3, r_2, r_1, r_0](d_3, \mathbf{j}, d_1, \mathbf{i})$$

and expand the storage location before the permutation to

$$d_3 r_2 r_1 r_0 + \underline{d_1 r_0} + (\mathbf{j} r_1 r_0 + \mathbf{i}) \tag{11}$$

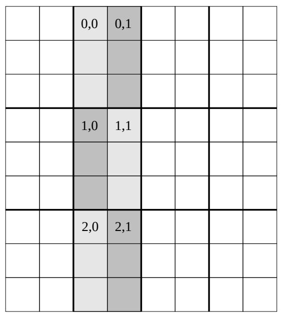


Figure 5: An illustration of the 3×2 embedded matrix (inside a 9×8 block-partitioned matrix) associated with $[3, 2, 3, 3](1, \mathbf{j}, \mathbf{i}, d_0)$, $d_0 = 0$. Each shaded vector of length three groups together the contiguous elements obtained by varying the choice of d_0 between 0 and $r_0 - 1 = 2$.

and the storage location after the permutation to

$$d_3 r_2 r_1 r_0 + \underline{d_1 r_2} + (\mathbf{i} r_2 r_1 + \mathbf{j}). \tag{12}$$

The embedded matrices are $r_0 \times r_2$. Note that the underlined term is slightly changed. This implies that if $r_0 \neq r_2$, then the first storage locations ($\mathbf{i} = \mathbf{j} = 0$) differ before and after the permutation is applied. Therefore, it is impossible to independently tranpose each embedded matrix in such cases. However, if the embedded matrices are indeed square (i.e., $r_0 = r_2$ in the example) then the two underlined terms are equal and it is easy to verify that independent in-place transpositions of each embedded matrix is possible. We remark that swapping two non-adjacent digits can be performed by a sequence of adjacent digit swaps and that a Kronecker product factorization of the permutation matrix can be used to express the same result. In summary, swapping two non-adjacent digits (i.e., applying a separated transposition pattern) is possible to implement using square in-place transposition of the embedded matrices assuming they are square.

The final type of transposition pattern that we have identified is the *fused* pattern (j, i, j, i). The double set of indices indicate that there are two sets of embedded matrices and the fused pattern is indeed just a combination of the two regular patterns $(j, i, \cdot, \cdot)$ and $(\cdot, \cdot, j, i)$. Each contiguous set of elements in the former pattern corresponds to an embedded matrix in the latter pattern. Hence, an implementation of the fused pattern could perform the transpositions associated with the latter pattern $(\cdot, \cdot, j, i)$ while moving elements as a part of the former pattern $(j, i, \cdot, \cdot)$. The cost would essentially be half that of applying the two patterns individually since each element is fetched from memory only once. In Kronecker product notation, the idea of a fused pattern stems from

$$\underbrace{(L_{r_2}^{r_3 \cdot r_2} \otimes I_{r_1 r_0})}_{(j,i,\cdot,\cdot)} \underbrace{(I_{r_3 r_2} \otimes L_{r_0}^{r_1 \cdot r_0})}_{(\cdot,\cdot,j,i)} = \underbrace{L_{r_2}^{r_3 \cdot r_2} \otimes L_{r_0}^{r_1 \cdot r_0}}_{(j,i,j,i)}.$$

The table below summarizes the seven possible transposition patterns.

Pattern	Comment	Name
$(\cdot,\cdot,j,i)$	Pointwise	
$(\cdot,j,i,\cdot)$		Regular
$(j,i,\cdot,\cdot)$		
$(\cdot,j,\cdot,i)$	$r_2 = r_0$	
$(j,\cdot,\cdot,i)$	$r_3 = r_0$	Separated
$(j,\cdot,i,\cdot)$	$r_3 = r_1$	
(j,i,j,i)	$r_1 r_0$ is small	Fused

We remark that the separated pattern $(\cdot,j,\cdot,i)$ can be implemented with limited additional memory as a sequence of out-of-place transpositions even if the embedded matrix is not square. This fact is used in, for instance, Dow's V5 algorithm.

6 Matrix Storage Format Conversions

All six of the storage formats reviewed in Section 2 can be succinctly described by a mixed-radix number system as in Section 5 using a block-partitioned matrix (i.e., $m = Mm_b$ and $n = Nn_b$). The ability to permute the digits is therefore all we need to do in-place conversion between each pair of formats. The six storage formats and some of the permutations that convert between them are shown in Figure 6. The following permutations are referred to in the diagram:

1 regular pattern $(\cdot,j,i,\cdot)$,

2 regular pattern $(j,i,\cdot,\cdot)$,

3 regular pattern $(\cdot,\cdot,j,i)$.

The diagonal arrows in the middle of the diagram represent the fused pattern (j,i,j,i) which is the combination of 2 and 3.

Conversion between two block formats with mismatching block sizes can be implemented by first converting to CM/RM (whichever is closer in the lattice) and from there to the output format. This is the strategy we use in our conversion software. It is an open question whether special relationships between mismatching block sizes (such as the output block size being a multiple of the input block size) can be exploited to improve performance.

7 Three-Stage Algorithm for Transposition

Below we describe an interesting algorithm for in-place transposition, which was mentioned already in [14] and brought to our attention by Fred Gustavson [10]. Bold radices and digits indicate which digits that are about to be swapped.

$$\begin{array}{ll}
[N, \mathbf{n_b}, \mathbf{M}, m_b](j_1, \mathbf{j_2}, \mathbf{i_1}, i_2) & \text{Stage A} \\
[\mathbf{N}, \mathbf{M}, n_b, m_b](\mathbf{j_1}, \mathbf{i_1}, j_2, i_2) & \text{Stage B1} \\
[M, N, \mathbf{n_b}, \mathbf{m_b}](i_1, j_1, \mathbf{j_2}, \mathbf{i_2}) & \text{Stage B2} \\
[M, \mathbf{N}, \mathbf{m_b}, n_b](i_1, \mathbf{j_1}, \mathbf{i_2}, j_2) & \text{Stage C} \\
[M, m_b, N, n_b](i_1, i_2, j_1, j_2) &
\end{array}$$

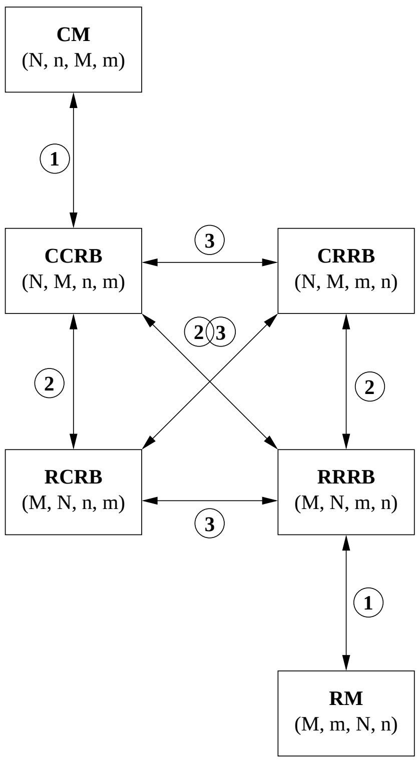


Figure 6: Conversion lattice showing six storage formats, their radices in a mixed-radix number system, and some of the permutations (circled numbers) that convert between them.

Assuming that m_b and n_b are reasonably large, say between 50 and 100 (see Section 9.3.1), then the following item list includes some of the reasons to why this algorithm can be expected to be efficient.

- Stage A results in N embedded matrices with vector length m_b.
- Stage B1 results in one embedded matrix with vector length $m_b n_b$.
- Stage B2 results in MN embedded matrices that are small enough ($m_b n_b$ elements) to be transposed out-of-place.
- Stage C results in M embedded matrices with vector length n_b (compare with Stage A).
- Stages B1 and B2 can be fused, reducing the number of sweeps through the matrix from four to three.
- Small cuts on the rows and/or columns can be used to ensure that the block sizes m_b and n_b are suitable (neither too small nor too large).
- As few as three sweeps (no cuts) and at most five sweeps (cuts of both rows and columns) through the matrix are required.

Note that all stages result in in-place transposition problems that are substantially smaller than the original $m \times n$ problem.

The algorithm is easily understood with the help of the conversion lattice in Figure 6. Stage A implements a conversion from CM to CCRB format, Stage B converts from CCRB to RRRB, and finally Stage C converts from RRRB to RM format (i.e., the matrix has been transposed). Take a 9×6 matrix with blocks of size 3×2 as an example (Figure 1). Stage A converts to CCRB format (Figure 2). Stages B converts from CCRB to RRRB (Figure 7). Finally, Stage C

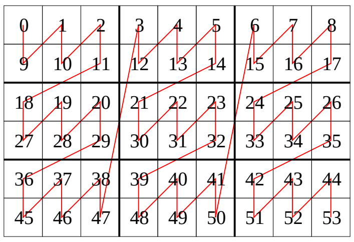


Figure 7: Configuration of a 9×6 matrix after Stage B. The matrix is now transposed but in CCRB format (i.e., the original matrix is in RRRB format).

converts from RRRB to RM (Figure 8).

8 Software

We have developed a software package written in C99 for the purpose of providing fast in-place transposition of large rectangular matrices and in-place conversion between storage formats. Some of the features of the package are listed below.

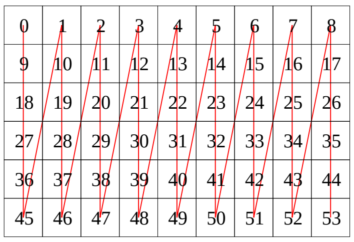


Figure 8: Configuration of a 9×6 matrix after Stage C. The matrix is now transposed and in CM format (i.e., the original matrix is in RM format).

- Portable since it is written in C99.
- Uses the Three-Stage Algorithm for transposition and cycle-following algorithms for in-place permutation.
- Uses the ideas of Brenner [3] to prune the search for cycle leaders.
- Exploits square transposition algorithms when any intermediate matrix is square.
- Automatically selects suitable block sizes m_b and n_b by finding the largest divisor of m such that $b_{\text{low}} \leq m_b \leq b_{\text{high}}$ and similarly for n_b.
- The parameters b_{low} and b_{high} enable tuning to a particular machine.
- Uses small cuts of rows and columns to give improved performance when no appropriate block sizes can be chosen (for example, when m and/or n are prime).
- Driver routines for in-place conversion back and forth between all 15 pairs of these storage formats:
  - CM/RM
  - CCRB/CRRB
  - RCRB/RRRB

9 Computational Experiments

Extensive experiments have been carried out on two different architectures. On each machine, all cores of a node were reserved for the application and only one core was actually used during each test. We present performance figures for the Three-Stage Algorithm together with comparisons with Dow's V5 algorithm, Alltop's three-stage algorithm, and out-of-place transposition. We also present a qualitative study of various cycle-following algorithms on a large set of problems.

9.1 Machines

We have performed our experiments using two different machines at the HPC2N facility in Umeå, Sweden. Since the execution of matrix transposition is ultimately memory bound we have benchmarked both systems using two benchmarks that are inspired by the STREAM benchmark [16]:

- **Copy** measures the time it takes to copy a large vector of double precision numbers ($y \leftarrow x$).
- **Scale** measures the in-place scaling of a double precision vector ($x \leftarrow \alpha{\cdot}x$). The multiplication with a scalar is merely to hinder the compiler from optimizing away the entire loop body. There should be plenty of spare clock cycles available to hide the overhead of the multiplication.

It is important to notice that the benchmarks are not intended to measure the peak hardware memory bandwidth but to establish the practical peak bandwidth obtainable by our particular combination of hardware, compiler, and compiler optimizations. Both benchmarks are implemented as one-statement for-loops in C99 and verification of the assembler output shows that the compiler does unrolling and vectorization. In contrast with the STREAM benchmark, our code uses dynamically allocated memory and the vector size is determined at runtime. This makes the benchmark more similar to a typical usage pattern.

We chose these two benchmarks to capture two fundamentally different usage patterns. In an out-of-place transposition, the matrix is copied from one memory area to another and then copied back. If the matrix is large, then the memory written to does not reside in the cache. This usage pattern is captured by the Copy benchmark. A cycle-following in-place transposition, on the other hand, moves data around cycles. The memory that is written to was recently read and hence is likely to be found in the cache. This usage pattern is similar to that in the Scale benchmark.

Some characteristics of the two machines are given in Table 1. The benchmark figures are the time divided by the length of the vectors. The Copy benchmark is roughly 58% slower than the Scale benchmark on Akka and roughly 43% slower on Sarek.

Name	*Akka*	*Sarek*
Processor	Dual Intel Xeon QC L5420	Dual AMD Opteron 248
Frequency	2.5 GHz	2.2 GHz
Memory	16 GB	8 GB
Compiler	PathScale 3.1	PathScale 3.1
Switches	`-O3 -march=auto`	`-O3 -march=auto`
BM: Copy (t_{copy})	4.845 ns (3098 MB/s)	6.731 ns (2265 MB/s)
BM: Scale (t_{scale})	3.067 ns (4925 MB/s)	4.708 ns (3240 MB/s)

Table 1: Characteristics of the HPC2N machines used for the experiments.

Many cache-based systems use a *write-allocate* strategy when writing to memory that is not cached. The write-allocate strategy means that the hardware reads the cache line into cache prior to the write. This has the side-effect that a write to uncached memory requires twice the memory bandwidth compared to a memory read or a cached write. The vector extension SSE (used on our machines) provides special instructions to write directly to memory (so-called *non-temporal* instructions). Nonetheless, the Copy and Scale benchmarks indicate that a dramatic difference can be observed.

9.2 Qualitative Study of Cycle-Following Algorithms

The general cycle test employed by many cycle-following algorithms (recall Section 3) is one of the largest sources of overhead in such algorithms. To get an idea of how significant this overhead may be and how it can be reduced by using a hybrid method with a limited lookup table, we experimented with three different algorithms using three different table sizes on all of the 61752 rectangular matrices $m \times n$ with $m, n \in \{2, \ldots, 250\}$. We counted the number of times (α) that P^{-1} was evaluated during a general cycle test. The number α varies considerably between problems, so to get an overview we computed two statistical quantities:

$$\max\left(\frac{\alpha}{mn}\right) \quad \text{and} \quad \operatorname{avg}\left(\frac{\alpha}{mn}\right).$$

The maximum and average are taken across all 61752 problems. The ideal scenario is for both of these to be close to zero. An average of one means that we expect to calculate P^{-1} approximately twice as many times as necessary (roughly half of them during the cycle shifting and the other half during the general tests).

The algorithms we considered were ACM Algorithm 467 [3] (A467), ACM Algorithm 513 [4] (A513), and a simplified variant of ACM Algorithm 467 that does not take advantage of companion cycles (A467s). For each of these algorithms the table sizes 0, $(m+n)/2$, and 100 were used. The results are reported in Table 2. The worst case for A467 was 2.19 without any table, which is not

Algorithm	A513			A467			A467s		
Table Size	0	$\frac{m+n}{2}$	100	0	$\frac{m+n}{2}$	100	0	$\frac{m+n}{2}$	100
$\max\left(\frac{\alpha}{mn}\right)$	6.89	2.77	3.40	2.19	1.46	1.58	4.28	3.14	3.31
$\operatorname{avg}\left(\frac{\alpha}{mn}\right)$	1.79	0.72	0.78	0.42	0.07	0.07	1.24	0.24	0.27

Table 2: Comparison of the number of times that P^{-1} is evaluated as part of general cycle tests for three hybrid cycle-following in-place transposition algorithms using three different table sizes.

so alarming since evaluating P^{-1} is not so expensive. Looking at the average values, going from no table to a table of size $\frac{m+n}{2}$ does bring a substantial reduction. For A467 the average is reduced by a factor of 6.0, for A467s the factor is 5.17, while for A513 it is only 2.49. The range of problems studied above is relevant in our context since the Three-Stage Algorithm with $m_b = n_b = 50$ requires the solution of transposition problems that are within the considered range for all $m, n \in \{1, \ldots, 12500\}$.

We also measured the number of cycles (β) for each of the problems. This is an interesting problem characteristic which also varies considerably across the problem space. We found that

$$\max\left(\frac{\beta}{mn}\right) \approx 0.33 \quad \text{and} \quad \operatorname{avg}\left(\frac{\beta}{mn}\right) \approx 0.01.$$

Thus, the average cycle length is approximately $\frac{mn}{0.01mn} = 100$.

9.3 Evaluation of the Three-Stage Algorithm

The performance of pointwise cycle-following algorithms is very poor due to high overhead per element and a seemingly random memory access pattern. Our experiments indicate that the execution time is typically between 5 to 10 times longer compared to the Three-Stage Algorithm. Therefore, we instead compare with an out-of-place transposition and implementations of Dow's V5 algorithm and Alltop's three-stage algorithm.

We begin by evaluating different aspects of the Three-Stage Algorithm before moving on to algorithm comparisons in Section 9.4.

9.3.1 Block Size

The block size can not be chosen freely since the possible block sizes depend on the problem size. However, cuts can be applied to alter the problem size. It is important to understand how the problem size impacts performance in order to determine when it is economical to pay for the overhead of cutting. We designed an experiment where we modified a fixed problem size so that both dimensions are divisible by some common factor (the block size). By doing this for all block sizes from 1 to 200 we get results for a collection of almost equally large problems for a range of block sizes. By comparing the *time per element* ($\frac{T}{mn}$) instead of actual execution time we reduce the impact of the different problem sizes. The results on Akka displayed in Figure 9 show the total time as well as the time for each of the three stages. The results on Sarek are qualitatively similar to the results on Akka. There is a peak at $m_b = n_b = 128$ in Stage B which is

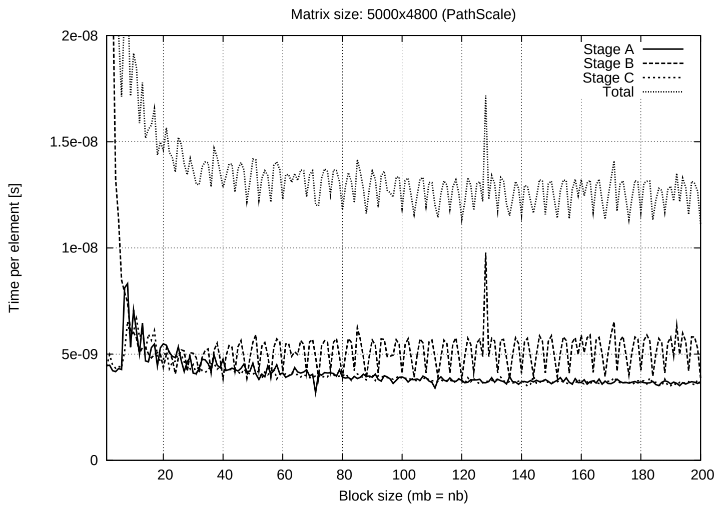


Figure 9: Performance breakdown on Akka of the Three-Stage algorithm on a fixed problem (5000×4800) for all block sizes ($m_b = n_b$) in the range $1, \ldots, 200$.

likely caused by cache thrashing in the small out-of-place transpositions. Other than that, a block size larger than 30 gives good performance whereas smaller block sizes should be avoided, mainly due to the performance of Stage B which eventually turns into a pointwise cycle-following algorithm when $m_b = n_b = 1$.

9.3.2 Cutting and Other Overhead

Cuts require an additional sweep over the matrix and will therefore impact performance. Similar to our experiment with different block sizes we started from a fixed problem size and a fixed block size and modified the problem size to induce cuts of size 1 to 99 in both rows and columns. The performance of the pre- and post-processing steps (the cuts) on Akka are reported in Figure 10 (the results on Sarek are qualitatively similar). Note that the performance is

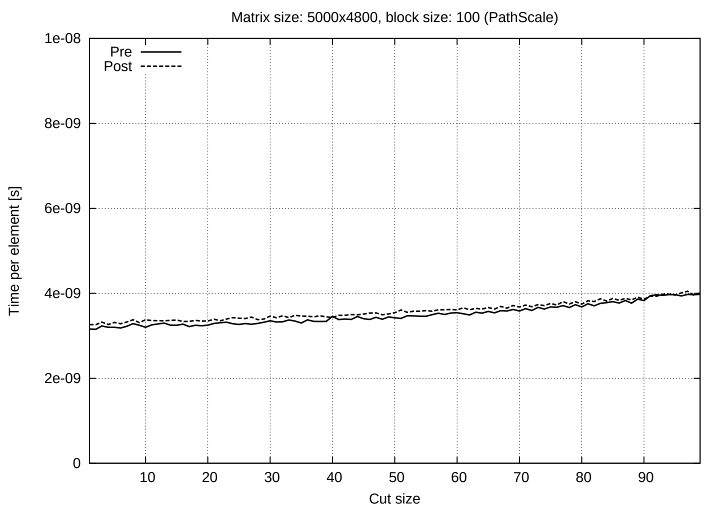


Figure 10: Performance on Akka of the pre- and post-processing steps during a symmetric cut on a fixed problem (5000 × 4800).

comparable to the three stages as reported in Section 9.3.1 and the introduction of a cut costs alot whereas a large cut does not cost significantly more than a small cut.

An interesting aspect of the implementation is the observed overhead in the cycle-following part of the algorithm. We measured the time spent in cycle shifting in addition to the total time and Figure 11 shows the results on Akka for various block sizes. For larger block sizes the overhead is insignificant whereas for smaller block sizes the implementation might be improved by introducing a lookup table.

9.4 An Evaluation of Transposition Algorithms

There are many parameters that affect the performance of transposition algorithms. Examples include the implementation of an algorithm, choice of compiler and settings, different optimizations and machine characteristics. The problem size and the shape of the matrix also affect performance. Some problem sizes might require cuts or cause severe cache thrashing. Some algorithms have additional tuning parameters such as block sizes and thresholds.

We compare the Three-Stage Algorithm, Murray Dow's V5 algorithm, Alltop's algorithm, and two implementations of out-of-place transposition: one naive implementation and one tuned cache-blocked algorithm. We have per-

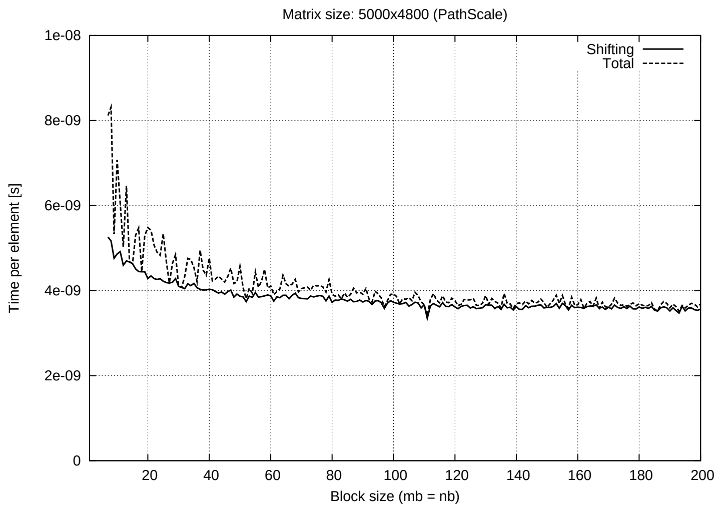


Figure 11: Performance breakdown on Akka of Stage A in the Three-Stage algorithm on a fixed problem (5000 × 4800). The time to do the actual shifting is shown together with the total time, which includes finding the cycle leaders. For larger block sizes the overhead is negligible but for some small block sizes the overheads are significant.

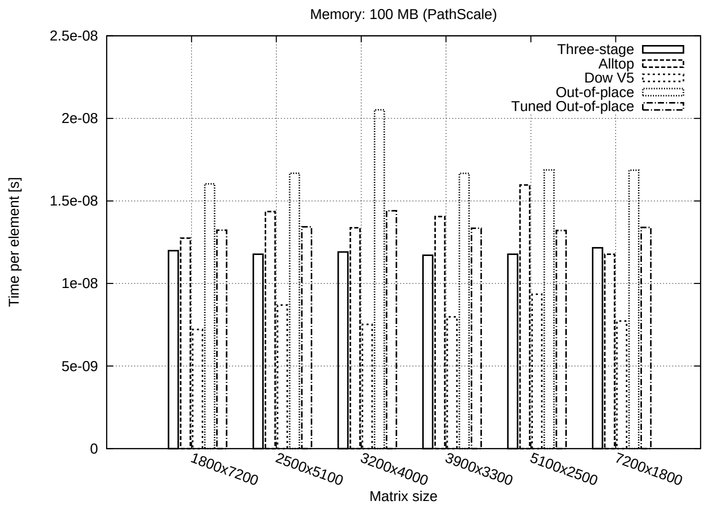


Figure 12: Performance comparison on Akka using matrices of different shapes that all require roughly 100 MB of memory.

formed many experiments on both machines to investigate and capture the behaviour of each algorithm. We have selected representative data that capture most of what we found during experimentation. In the listing below, we summarize some facts about our experiments.

- The block size for the Three-Stage Algorithm was set to $m_b = n_b = 100$ (a suitable block size based on the results in Section 9.3.1) and all problem sizes were multiples of 100. Thus, no cuts were required and a block size of 100 was used for all executions.
- The common divisor D in Alltop's algorithm and Dow's V5 algorithm was chosen optimally for each problem size.
- All experiments were repeated five times and only the best result was kept in order to minimize the noise from system activities.
- The out-of-place algorithm was optimized by the compiler, which is capable of cache-blocking transformations.
- The tuned out-of-place algorithm is an automatically tuned implementation of a blocked transposition where different block traversal schemes and cache block sizes were taken into account. We chose the best performing implementation on a 1200×1800 test problem and that implementation traverses the blocks and elements in row-major order and has a block size of 16×16.

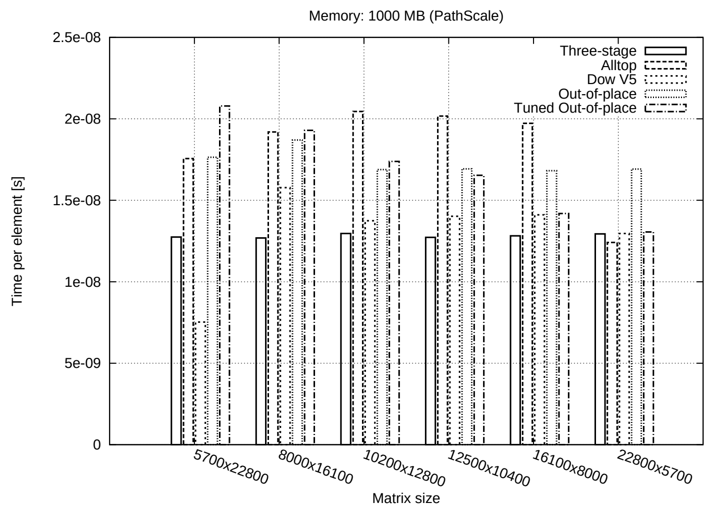


Figure 13: Performance comparison on Akka using matrices of different shapes that each require roughly 1000 MB of memory.

We observed that the memory footprint and the shape of the matrix affected performance significantly. We therefore performed experiments on matrices of different shapes having roughly the same memory footprint. For matrices requiring around 100 MB, the results on Akka are displayed in Figure 12. For

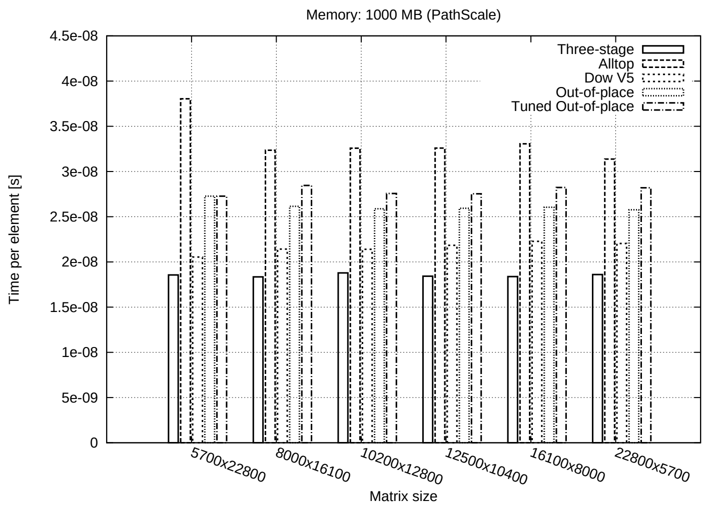


Figure 14: Performance comparison on Sarek using matrices of different shapes that each require roughly 1000 MB of memory.

larger matrices of around 1000 MB, the results on Akka are shown in Figure 13 and the corresponding results on Sarek are in Figure 14.

Dow's V5 algorithm is sometimes considerably faster than any of the other algorithms, especially for smaller matrices. The Three-Stage Algorithm has a relatively predictable performance (partly due to the fixed block size of $m_b = n_b = 100$) and is among the fastest, especially for large matrices. Alltop's algorithm appears to be rather inefficient for large matrices (see Figures 13 and 14).

The performance of Dow's V5 algorithm differs alot between small and large matrices and also for different shapes of large matrices. We think that this is partly due to data locality issues but also due to the worse performance of out-of-place transformations in general. In Table 3, we show two simple models of the execution time for each of the algorithms. The Three-Stage Algorithm

Algorithm	**Optimistic**	**Pessimistic**
Three-Stage	$3mn \cdot t_{\text{scale}}$	—
Dow's V5	$mn \cdot t_{\text{scale}} + mn \cdot t_{\text{scale}}$	$mn \cdot t_{\text{scale}} + 2mn \cdot t_{\text{copy}}$
Alltop	$2mn \cdot t_{\text{scale}} + mn \cdot t_{\text{scale}}$	$2mn \cdot t_{\text{scale}} + 2mn \cdot t_{\text{copy}}$
Out-of-place	$2mn \cdot t_{\text{copy}}$	—

Table 3: Models of the execution time under both an optimistic and a pessimistic scenario.

consists of three stages that are in-place (i.e., similar to Scale). The out-of-place algorithms have two stages that are both out-of-place (i.e., similar to Copy). The remaining algorithms have different models depending on the problem size. Dow's V5 algorithm has a first stage of in-place character and a second stage which is an out-of-place transformation of $Dm_bn_b = mn_b$ elements at a time. If the cache is large enough to retain the buffer until it is copied back, then the usage pattern is similar to the Scale benchmark in that the written memory resides in cache. This is the *optimistic* scenario. On the other hand, if the cache is not large enough, then the second stage consists of two operations similar to the situation in the Copy benchmark. This requires much more bandwidth (more than twice since $t_{\text{copy}} > t_{\text{scale}}$) and is the *pessimistic* scenario. A similar analysis has been done for Alltop's algorithm and all models are summarized in Table 3. Note that a multilevel cache hierarchy means that a transition from an optimistic to a pessimistic scenario will be gradual.

These models match the data pretty well. For example, most cases where Dow's V5 algorithm is considerably faster than the others are of the optimistic type whereas the other cases tend to be of the pessimistic type. Table 4 shows the prediction of each model on both machines. The figures in that table are for comparison with Figures 12, 13, and 14.

	Akka		**Sarek**	
Algorithm	Optimistic	Pessimistic	Optimistic	Pessimistic
Three-Stage	0.920	—	1.412	—
Dow's V5	0.613	1.276	0.942	1.817
Alltop	0.920	1.582	1.412	2.288
Out-of-place	0.969	—	1.346	—

Table 4: Model predictions on both machines (all figures should be multiplied by 10^{-8}). These figures are for comparison with Figures 12, 13, and 14.

10 Conclusions and Future Work

We have demonstrated that it is possible to develop cache-efficient in-place transposition algorithms based on generalized versions of previously known cycle-following algorithms. The performance of such algorithms rivals the best known semi-in-place algorithms by Alltop and Dow as well as out-of-place transposition. The write-allocate strategy of cache-based computer architectures, although in theory partly alleviated with non-temporal instructions, can be a performance problem in practice.

Conversion between the CM, RM, and the four block formats CCRB, CRRB, RCRB, and RRRB can be performed in-place using our software package which is based upon the techniques discussed in this paper. The implementation uses the cut transpose technique to give reasonable performance for cases when m and/or n do not enable a suitable choice of block size.

Evidence suggests that a hybrid implementation of the cycle-following kernel could be used instead of the pure implementation we currently have in order to reduce overhead.

Thread parallelization of independent subproblems and/or cycle shifts might improve performance on multicore architectures. The best algorithm for a particular problem depends on many parameters and there is no single best algorithm.

Acknowledgements

I would like to acknowledge Fred Gustavson for introducing me to the subject of in-place transposition and storage format conversions. He pointed out the merits of the Three-Stage Algorithm and together with Tadeusz Swirszcz he recently published improvements to cycle-following algorithms [11]. I also acknowledge Bo Kågström for his kind support, references to and comments on the subject, and constructive criticism on drafts of this manuscript.

References

[1] W. O. Alltop. A Computer Algorithm for Transposing Nonsquare Matrices. *IEEE Transactions on Computers*, 24(10):1038–1040, 1975.

[2] J. Boothroyd. Algorithm 302: Transpose Vector Stored Array. *Communications of the ACM*, 10(5):292–293, 1967.

[3] N. Brenner. Algorithm 467: Matrix Transposition in Place. *Communications of the ACM*, 16(11):692–694, 1973.

[4] E. G. Cate and D. W. Twigg. Algorithm 513: Analysis of In-Situ Transposition. *ACM Transactions on Mathematical Software*, 3(1):104–110, 1977.

[5] M. Dow. Transposing a Matrix on a Vector Computer. *Parallel Computing*, 21(12):1997–2005, 1995.

[6] J. O. Eklundh. A Fast Computer Method for Matrix Transposing. *IEEE Transactions on Computers*, 21(7):801–803, 1972.

[7] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive Blocked Algorithms and Hybrid Data Structures for Dense Matrix Library Software. *SIAM Review*, 46(1):3–45, 2004.

[8] D. Fraser. Array Permutation by Index-Digit Permutation. *Journal of the ACM*, 23:298–309, 1976.

[9] M. Frigo and S. G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[10] F. G. Gustavson. The Relevance of New Data Structure Approaches for Dense Linear Algebra in the New Multicore/Manycore Environments. Technical Report RC24599, IBM Research, 2008. (Also submitted to PARA'08).

[11] F. G. Gustavson and T. Swirszcz. In-Place Transposition of Rectangular Matrices. In B. Kågström et al., editors, *Applied Parallel Computing. State of the Art in Scientific Computing, PARA 2006.* Lecture Notes in Computer Science, Vol. 4699, pages 560–569. Springer, 2007.

[12] H. V. Henderson and S. R. Searle. The vec-Permutation Matrix, the vec Operator and Kronecker Products: a Review. *Linear and Multilinear Algebra*, 9:271–288, 1981.

[13] J. R. Johnson. Matrix Transposition. Department of Mathematics and Computer Science, Drexel University, Philadelphia, PA 19104, 1995. (Manuscript).

[14] S. D. Kaushik, C. H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient Transposition Algorithms for Large Matrices. In *Proceedings of Supercomputing '93*, pages 656–665, 1993.

[15] S. Laflin and M. A. Brebner. Algorithm 380: In-situ Transposition of a Rectangular Matrix. *Communications of the ACM*, 13(5):324–326, 1970.

[16] J. D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, December 1995.

[17] N. Park, B. Hong, and V. K. Prasanna. Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[18] H. K. Ramapriyan. A Generalization of Eklundh's Algorithm for Transposing Large Matrices. *IEEE Transactions on Computers*, 24(12):1221–1226, 1975.

A Reformulated Algorithms

Below is the V4 algorithm by Murray Dow [5, Algorithm V4] expressed in our notation:

$$\begin{array}{c}
[\,\mathbf{n}\,, \mathbf{M}, m_b](\,\mathbf{j}\,, \mathbf{i_1}, i_2) \\
[M,\ \mathbf{n}\,, \mathbf{m_b}](i_1,\ \mathbf{j}\,, \mathbf{i_2}) \\
[M, m_b,\ n\ ](i_1, i_2,\ j\ )
\end{array}$$

The algorithm applies when $m = Mm_b$ with m_b being the block size. The final representation is correct since the memory location is

$$i_1 m_b n + i_2 n + j = (i_1 m_b + i_2)n + j = in + j$$

which agrees with the memory location of the matrix in RM format. Both steps can be performed in-place (regular transposition patterns).

Dow's V5 algorithm [5, Algorithm V5], which applies when $m = Dm_b$ and $n = Dn_b$, can be expressed as:

$$\begin{array}{c}
[D, \mathbf{n_b}, D, \mathbf{m_b}](j_1, \mathbf{j_2}, i_1, \mathbf{i_2}) \\
[\mathbf{D}, m_b, \mathbf{D},\ n_b\ ](\mathbf{j_1}, i_2, \mathbf{i_1}, j_2) \\
[D, m_b, D,\ n_b\ ](i_1, i_2, j_1, j_2)
\end{array}$$

The first step can not be performed in-place since n_b and m_b are different (or otherwise the matrix would be square) and hence the embedded matrices are rectangular. The second step can be performed in-place since the embedded matrices are square.

The three-stage algorithm by Alltop [1], which applies when $m = Dm_b$ and $n = Dn_b$ is expressed below:

$$\begin{array}{ll}
[\mathbf{D}, n_b, \ \mathbf{D}\ , m_b](\mathbf{j_1}, j_2, \mathbf{i_1}, i_2) & \text{Stage A} \\
[D, n_b, \ \mathbf{D}\ , \mathbf{m_b}](i_1, j_2, \mathbf{j_1}, \mathbf{i_2}) & \text{Stage B1} \\
[D, \mathbf{n_b}, \mathbf{m_b}, \ D\ ](i_1, \mathbf{j_2}, \mathbf{i_2}, j_1) & \text{Stage B2} \\
[D, m_b, \mathbf{n_b}, \ \mathbf{D}\ ](i_1, i_2, \mathbf{j_2}, \mathbf{j_1}) & \text{Stage C} \\
[D, m_b, \ D\ , n_b\ ](i_1, i_2, j_1, j_2) &
\end{array}$$

Stage A has a separated pattern but since it is square it can be performed in-place. Stages B1 and B2 together is actually just an adjacent digit swap in another mixed-radix number system, namely

$$[D, \mathbf{n_b D}, \mathbf{m_b}](i_1, \mathbf{j_2 D} + \mathbf{j_1}, \mathbf{i_2}).$$

Stages B and C are also possible to perform in-place but Alltop suggested using out-of-place transposition without mentioning the possibility of in-place transposition. The additional memory required is $n_b D m_b$ for Stage B and only $n_b D$ for Stage C.