A Decomposition for In-place Matrix Transposition

Bryan Catanzaro
NVIDIA Research
bcatanzaro@nvidia.com

Alexander Keller
NVIDIA Research
akeller@nvidia.com

Michael Garland
NVIDIA Research
mgarland@nvidia.com

Abstract

We describe a decomposition for in-place matrix transposition, with applications to Array of Structures memory accesses on SIMD processors. Traditional approaches to in-place matrix transposition involve cycle following, which is difficult to parallelize, and on matrices of dimension m by n require $O(mn \log mn)$ work when limited to less than $O(mn)$ auxiliary space. Our decomposition allows the rows and columns to be operated on independently during in-place transposition, reducing work complexity to $O(mn)$, given $O(\max(m, n))$ auxiliary space. This decomposition leads to an efficient and naturally parallel algorithm: we have measured median throughput of 19.5 GB/s on an NVIDIA Tesla K20c processor. An implementation specialized for the skinny matrices that arise when converting Arrays of Structures to Structures of Arrays yields median throughput of 34.3 GB/s, and a maximum throughput of 51 GB/s.

Because of the simple structure of this algorithm, it is particularly suited for implementation using SIMD instructions to transpose the small arrays that arise when SIMD processors load from or store to Arrays of Structures. Using this algorithm to cooperatively perform accesses to Arrays of Structures, we measure 180 GB/s throughput on the K20c, which is up to 45 times faster than compiler-generated Array of Structures accesses.

In this paper, we explain the algorithm, prove its correctness and complexity, and explain how it can be instantiated efficiently for solving various transpose problems on both CPUs and GPUs.

Categories and Subject Descriptors E.1 [*Data Structures*]: Arrays; F.2.1 [*Analysis of Algorithms and Problem Complexity*]: Numerical Algorithms and Problems

PPoPP '14, February 15–19, 2014, Orlando, FL, USA.

http://dx.doi.org/10.1145/2555243.2555253

1. Introduction

In-place matrix transposition is a well-studied problem, with papers being published on the subject from 1959 [11] until the present day [1]. In-place transposition for square matrices is straightforward, but for non-square matrices, the algorithms are more involved. Traditional approaches to in-place transposition operate by following cycles in the permutation induced by the transposition. Since storing cycle descriptors requires $O(mn)$ space, cycle following in-place algorithms with auxiliary storage requirements less than $O(mn)$ elements have work complexity $O(mn \log mn)$, due to the need to recompute the cycles as the transposition proceeds [3].

In this paper, we show how the in-place transposition problem can be decomposed into independent row-wise and column-wise permutations. By decomposing the transposition, we improve the algorithmic complexity by performing smaller permutations out-of-place. With $O(\max(m, n))$ auxiliary storage, our algorithm requires $O(mn)$ work, and requires no cycle following. Our algorithm works on arrays linearized in row-major or column-major order.

In contrast to traditional cycle following algorithms, which can be difficult to parallelize due to poorly distributed cycle lengths, our decomposed transposition is straightforward to parallelize, with perfect load balancing due to the regular structure of the decomposition.

Additionally, our algorithm enables efficient SIMD transpositions on the very small arrays that arise from vector memory operations on SIMD processors. When each lane of a SIMD processor requests a vector of data, the straightforward implementation accesses elements of the vectors sequentially, which results in strided memory accesses and dramatically reduced memory throughput. There are several techniques that can ameliorate this problem, and our algorithm creates a new alternative. Firstly, the data structure itself can be transposed in memory, which removes strided memory accesses. This technique is burdensome to programmers, and cannot be applied in many circumstances, such as

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. Approved for Public Release, Distribution Unlimited.

when data structures are dictated due to interface constraints or algorithmic requirements. Alternatively, programmers access the data in transposed order to ensure vectorized memory accesses, performing transpositions in on-chip memory to route the data to each SIMD lane. This technique is effective, but allocating on-chip memory in order to perform this transpose out-of-place can be difficult, especially when scarce on-chip memory resources are occupied with other tasks. Our transposition algorithm adds a new technique: perform the transposition in-place in the register file, without requiring additional on-chip memory.

This paper makes three main contributions. Firstly, we present and prove a new algorithm for in-place matrix transposition. We show it has optimal work complexity with reduced auxiliary storage requirements compared to traditional algorithms. Secondly, we discuss several practical implementations of this algorithm, on both parallel CPUs and GPUs, including optimizations to improve cache performance. Finally, we present an implementation that allows SIMD processors to efficiently perform arbitrary length vector memory loads and stores, without relying on scarce on-chip memory resources for an out-of-place transpose.

2. Decomposition

The core of our technique is a decomposition for in-place matrix transposition that reduces the overall transposition into a series of independent row and column permutations.

Traditional approaches to in-place transposition view the problem as a single, monolithic permutation of elements in an array. In contrast, our algorithm retains the natural view of the data as a two-dimensional array rather than as a linearized one-dimensional structure, operating on the rows and columns of the original array, until the data movement has been completed for the transposition. The data is then reinterpreted as a two-dimensional array with transposed dimensions.

Viewing the data as a two-dimensional array, the transposition can be accomplished in two directions, which we call "Rows to Columns" (R2C) and "Columns to Rows" (C2R). The R2C and C2R transposes are inverses of each other. These two permutations are illustrated in Figure 1.

We are not the first to view transposition in this manner, for example, see the description of Columnsort in Leighton [4], where the C2R permutation is called "transpose", and the R2C permutation is called "untranspose".

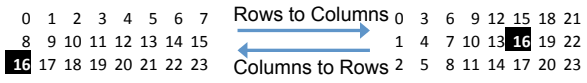


Figure 1: C2R and R2C transpositions, $m = 3, n = 8$

We begin by discussing out-of-place versions of these transpositions, and showing how they relate to traditional matrix transposition.

Define the standard row-major linearization, where $l_{rm}(i, j)$ creates a linear index from a row and column index, and $i_{rm}(l)$ and $j_{rm}(l)$ decompose a linear index into a row and column index, respectively.

$$l_{rm}(i, j) = j + in \tag{1}$$

$$i_{rm}(l) = \left\lfloor \frac{l}{n} \right\rfloor \tag{2}$$

$$j_{rm}(l) = l \bmod n \tag{3}$$

With the observation that $l_{rm}(i_{rm}(l), j_{rm}(l)) = l$.

And the standard column-major linearization:

$$l_{cm}(i, j) = i + jm \tag{4}$$

$$i_{cm}(l) = l \bmod m \tag{5}$$

$$j_{cm}(l) = \left\lfloor \frac{l}{m} \right\rfloor \tag{6}$$

With the observation that $l_{cm}(i_{cm}(l), j_{cm}(l)) = l$.

To define the R2C and C2R transpositions, we will use the following four functions:

$$s(i, j) = l_{rm}(i, j) \bmod m \tag{7}$$

$$c(i, j) = \left\lfloor \frac{l_{rm}(i, j)}{m} \right\rfloor \tag{8}$$

$$t(i, j) = \left\lfloor \frac{l_{cm}(i, j)}{n} \right\rfloor \tag{9}$$

$$d(i, j) = l_{cm}(i, j) \bmod n \tag{10}$$

We then define the transpositions:

$$A^{C2R}[i, j] = A[s(i, j), c(i, j)] \tag{11}$$

$$A^{R2C}[i, j] = A[t(i, j), d(i, j)] \tag{12}$$

Equations 11 and 12 define the transpositions in terms of gather operations. Since the C2R and R2C transpositions are inverses of each other, and scatter and gather transpositions are inverses of each other, we can also define the transpositions in terms of scatter operations:

$$A^{C2R}[t(i, j), d(i, j)] = A[i, j] \tag{13}$$

$$A^{R2C}[s(i, j), c(i, j)] = A[i, j] \tag{14}$$

For example, consider the element with value 16 highlighted in Figure 1, where $m = 3, n = 8$. On the left, this element is located at $i = 2, j = 0$, After the R2C transposition, the element is located at $i' = 1, j' = 5$. Looking at Equation 14, we can compute the destination indices: $i' = s(i, j) = (j + in) \mod m = (0 + 2 \cdot 8) \mod 3 = 1$, and $j' = c(i, j) = \left\lfloor \frac{j+in}{m} \right\rfloor = \left\lfloor \frac{0+2 \cdot 8}{3} \right\rfloor = 5$.

Now we show the connection between the R2C and C2R transposes and the linearized transposition problem.

Theorem 1. *The C2R transpose implements transposition for row-major arrays, and the R2C transpose implements transposition for column-major arrays.*

Proof. To prove this, we will examine linearized versions of a matrix A and its transpose A^T. We will show that the row-major linearization of A^T is equivalent to the row-major linearization of A^{C2R}.

By the definition of transposition,

$$A^T[i,j] = A[j,i] \tag{15}$$

Since A^T has dimensions $n \times m$, its indexing must be adjusted slightly. Define

$$i^T_{rm}(l) = \left\lfloor \frac{l}{m} \right\rfloor = j_{cm}(l) \tag{16}$$

$$j^T_{rm}(l) = l \bmod m = i_{cm}(l) \tag{17}$$

These equations are just Equations 2 and 3 with dimensions swapped.

Let A_{rm} be a row-major linearized representation of array A, indexed with $l \in [0, mn)$. Then

$$A^T_{rm}[l] = A_{rm}[l_{rm}(j^T_{rm}(l), i^T_{rm}(l))], \tag{18}$$

where we have linearized A^T and A, using the definition of transposition.

As defined in Equation 11,

$$A^{C2R}[i,j] = A[s(i,j), c(i,j)] \tag{19}$$

Linearizing,

$$A^{C2R}_{rm}[l] = A_{rm}[l_{rm}(s(i_{rm}(l), j_{rm}(l)), c(i_{rm}(l), j_{rm}(l)))] \tag{20}$$

However:

$$\begin{aligned} s(i_{rm}(l), j_{rm}(l)) &= l_{rm}(i_{rm}(l), j_{rm}(l)) \bmod m \\ &= l \bmod m = j^T_{rm}(l) \end{aligned}$$

$$\begin{aligned} c(i_{rm}(l), j_{rm}(l)) &= \left\lfloor \frac{l_{rm}(i_{rm}(l), j_{rm}(l))}{m} \right\rfloor \\ &= \left\lfloor \frac{l}{m} \right\rfloor = i^T_{rm}(l) \end{aligned}$$

And so we can substitute to show

$$A^{C2R}_{rm}[l] = A_{rm}[l_{rm}(j^T_{rm}(l), i^T_{rm}(l))] \tag{21}$$

Therefore, $A^{C2R}_{rm} = A^T_{rm}$. Symmetric reasoning shows $A^{R2C}_{cm} = A^T_{cm}$. □

Theorem 2. *Swapping dimensions m and n before performing the transpose, the C2R transpose implements transposition for column-major arrays, and the R2C transpose implements transposition for row-major arrays.*

Proof. Defining versions of l_{cm}, i_{rm}, and j_{rm} where m and n have been swapped:

$$\begin{aligned} l^s_{cm}(i,j) &= i + jn = l_{rm}(j,i) \\ i^s_{rm}(l) &= \left\lfloor \frac{l}{m} \right\rfloor = i^T_{rm}(l) \\ j^s_{rm}(l) &= l \bmod m = j^T_{rm}(l) \end{aligned}$$

The R2C transpose induces the following permutation:

$$\begin{aligned} A^{R2C}_{cm}[l] &= A_{cm}[l_{cm}(j^T_{cm}(l), i^T_{cm}(l))] \\ &= A_{cm}[l_{cm}(i_{rm}(l), j_{rm}(l)] \end{aligned}$$

If we swap m and n first, the resulting index expression is

$$\begin{aligned} l^s_{cm}(i^s_{rm}(l), j^s_{rm}(l)) &= l_{rm}(j^s_{rm}(l), i^s_{rm}(l)) \\ &= l_{rm}(j^T_{rm}(l), i^T_{rm}(l)) \end{aligned}$$

Since this index expression is the same as in Equation 21, we see that if m and n are swapped before the transposition, the R2C permutation transposes a row-major array. Symmetric reasoning shows that swapping m and n before the transposition allows the C2R permutation to transpose a column-major array.

□

We have described both the C2R and R2C transpositions and shown how they can both be used to transpose arrays of either row-major or column-major linearization.

3. Algorithm

The key insight in this work is that one need not consider the entire permutation required for performing the transposition in-place on an array. Instead, we can decompose the transposition into independent row-wise and column-wise permutations.

In this section, we prove that the decomposition underlying this technique is sound, and then present the algorithm. We will restrict our attention to the C2R transposition in this section, as the R2C transposition is merely the inverse of the C2R transposition.

As shown Equation 10, the destination column of element j in row i is:

$$d_i(j) = (i + jm) \bmod n \tag{22}$$

where we have fixed i for presentation purposes. We would like to perform row-wise permutations to send each element to the correct column required by the transposition. This can only be done if each element goes to a unique column, otherwise the row-wise operation is not a well-formed permutation, and the transposition is not decomposable.

However, in general, $d_i(j)$ is not bijective on $j \in [0, n)$, meaning each element does not go to a unique column, and so the row-wise operation is not a well-formed permutation.

In fact, $d_i(j)$ is periodic, which means there are guaranteed to be conflicts in the permutation. However, the periodicity of $d_i(j)$ gives us a clue as to how to remove these conflicts, as we will see.

Let the array have m rows and n columns. Define $c = \gcd(m, n)$, $a = \frac{m}{c}$, $b = \frac{n}{c}$.

Lemma 1. *$\forall i \in \mathbb{Z}$, $d_i(j)$ is periodic with period b.*

Proof. Given $j, k \in \mathbb{Z}$,

$$\begin{aligned} d_i(j + kb) &= \left(i + \left(j + k\frac{n}{c}\right) m\right) \bmod n \\ &= \left(i + jm + nk\frac{m}{c}\right) \bmod n \\ &= (i + jm + nka) \bmod n \\ &= (i + jm) \bmod n = d_i(j) \end{aligned}$$

Therefore, $d_i(j)$ is periodic with period b. □

Lemma 1 shows that if $c > 1$, multiple elements in each row will be sent to the same column, since in that case $b < n$. This means that the row-wise permutation that sends each element to the correct column does not exist if $c > 1$. However, note that if m and n are coprime, $c = 1$, and the period $b = n$. Later, we will prove that this means the decomposition is trivial in this case.

For the case when m and n are not coprime, we would like to find a set of column-wise permutations that ensure each element goes to a unique column. We must show that after these permutations, the destination column for each element is some new function $d'_i(j)$ that is bijective on the domain $[0, n)$.

Since $d_i(j)$ is periodic with period b, we adjust the array in groups of b columns to remove the conflicts. Consider rotating the columns of a matrix, by which we mean: for a column x of length m being rotated by k elements, the rotated column $x'[i] = x[(i + k) \bmod m]$. Consider rotating column j by $k = \left\lfloor \frac{j}{b} \right\rfloor$ elements, or equivalently, column j of the rotated array is gathered from the source array using index equation

$$r_j(i) = \left(i + \left\lfloor \frac{j}{b} \right\rfloor\right) \bmod m \tag{23}$$

Substituting, after rotating all columns of the array, the resulting destination column for each element of the new array is

$$d'_i(j) = \left(\left(i + \left\lfloor \frac{j}{b} \right\rfloor\right) \bmod m + jm\right) \bmod n \tag{24}$$

Our task is to prove that Equation 24 is a bijection, which will show that the rotations have removed conflicts, decomposing the transposition.

To do this, the following lemmas are useful.

Lemma 2. *$\forall x, y \in \mathbb{N} \mid 0 \le x < b, 0 \le y < b$, $mx \bmod n = my \bmod n$ implies $x = y$.*

Proof. Proof by contradiction. Assume $\exists x, \exists y \mid 0 \le x < b, 0 \le y < b, x \neq y$ and also that $mx \bmod n = my \bmod n$. Substituting, $acx \bmod bc = acy \bmod bc$. By cancellability of congruences, this implies $ax \bmod \frac{bc}{\gcd(c,bc)} = ay \bmod \frac{bc}{\gcd(c,bc)}$. Since $\gcd(c, bc) = c$, then $ax \bmod b = ay \bmod b$ must be true. Since a and b are coprime, the modular multiplicative inverse of a and b exists. Therefore, $x \bmod b = y \bmod b$ must be true. Since we assumed $0 \le x < b$ and $0 \le y < b$, the modulus is extraneous, and so $x = y$. But this is a contradiction, since we assumed earlier that $x \neq y$. □

Lemma 3. *Let $S = \bigcup_{h=0}^{b-1} \{hm \bmod n\}$, and let $T = \bigcup_{h=0}^{b-1} \{hc\}$. Then $S = T$.*

Proof. By Lemma 2, we know $|S| = b$. We also know $|T| = b$ by inspection. Next, we show that $S \subseteq T$. To do this, we show that $\forall h \in [0, b), \exists k \in [0, b) \mid hm \bmod n = kc$. By the definition of modulus, $hac \bmod bc = hac - bc \left\lfloor \frac{hac}{bc} \right\rfloor = (ha - b \left\lfloor \frac{hac}{bc} \right\rfloor)c = kc$, where $k \in \mathbb{Z}$. To bound k, we note that since kc is a remainder with respect to bc, $0 \le k < b$. Accordingly, $S \subseteq T$, and since we already showed $|S| = |T|$, it must be true that $S = T$. □

Theorem 3. *$d'_i(j)$ is a bijection on $j \in [0, n)$ for any fixed $i \in [0, m)$.*

Proof. Observing that $\left\lfloor \frac{j}{b} \right\rfloor = l$ is constant for $j \in [lb, (l + 1)b)$, we first analyze the sets

$$\begin{aligned} S_{i,l} &= \bigcup_{j=lb}^{(l+1)b-1} \{d'_i(j)\} \\ &= \bigcup_{j=lb}^{lb+b-1} \{((i + l) \bmod m + jm) \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{((i + l) \bmod m + (lb + h)m) \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{((i + l) \bmod m + hm) \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{((i + l) \bmod c + hm \bmod n) \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{(i + l) \bmod c + hm \bmod n\} \\ &= \bigcup_{h=0}^{b-1} \{(i + l) \bmod c + hc\}, \end{aligned}$$

where we first replace $d'_i(j)$ by its definition, followed by removing the offset lb from the index, which allows one to

cancel the resulting additive term

$$lbm \bmod n = lbac \bmod bc = 0$$

We then distribute the modulus over both remaining terms. We can replace the expression $((i+l) \bmod m) \bmod n$ by $(i+l) \bmod c$ by defining $k_m = \left\lfloor \frac{i+l}{m} \right\rfloor$ and $k_n = \left\lfloor \frac{i+l-k_m m}{n} \right\rfloor$, and $r = i + l - (k_m m + k_n n)$. Then

$$((i+l) \bmod m) \bmod n = r \text{ and } (i+l) \bmod c = r$$

due to $m = ac$ and $n = bc$. Noting that $(i+l) \bmod c \in [0, c)$, and $hac \bmod bc$ is kc, for $k \in [0, b)$, we see that $(i+l) \bmod c + hm \bmod n < bc = n$, so the external modulus is unnecessary. Then the last line follows from Lemma 3, noting that the term $(i+l) \bmod c$ is independent of h and so can we can replace the set $\bigcup_{h=0}^{b-1} \{hm \bmod n\}$ with $\bigcup_{h=0}^{b-1} \{hc\}$.

Now, for any fixed $i \in [0, m)$, the range of $d'_i(j)$ over the entire domain $[0, n)$ is

$$\begin{aligned} \bigcup_{j=0}^{n-1} \{d'_i(j)\} &= \bigcup_{l=0}^{c-1} S_{i,l} \\ &= \bigcup_{l=0}^{c-1} \bigcup_{h=0}^{b-1} \{hc + ((i+l) \bmod c)\} \\ &= [0, n) \end{aligned}$$

because $((i+l) \bmod c)$ enumerates all values in $[0, c)$ on the domain $l \in [0, c)$. Therefore $d'_i(j)$ is a bijection on $[0, n)$. □

Note that for $c = \gcd(n, m) = 1$, $\left\lfloor \frac{j}{b} \right\rfloor = 0$, yielding

$$d'_i(j) = (i + jm) \bmod n = d_i(j)$$

This implies that if m and n are coprime, $d_i(j)$ is naturally bijective.

Theorem 4. *In-place transposition can be decomposed into independent row-wise and column-wise operations.*

Proof. Since $d'_i(j)$ is bijective on the domain $j \in [0, n)$, then after pre-rotating columns of the array, each element can be sent to a unique destination column during independent row-wise permutations. Once each element is in the correct destination column, it necessarily has a unique row to which it should be sent to complete the transposition. Since the indices in both steps are unique, the row and column wise permutations are decomposable. □

We have already described the column-wise rotations, and given the set of independent row-wise permutations. Now we will give the column-wise permutations necessary to finish the transposition. Since the decomposition ensures each element is directed to the correct column via row-wise permutations, we need only consider permuting elements within the columns.

For the C2R transposition, Equation 7 shows that the source row of element i in column j is

$$s_j(i) = (j + in) \bmod m \tag{25}$$

However, since we rotated the original array to create $d'_i(j)$, the correct source row is a different function. Define:

$$s'_j(i) = \left(j + in - \left\lfloor \frac{i}{a} \right\rfloor \right) \bmod m \tag{26}$$

Theorem 5. *$s'_j(i)$ computes the correct source row indices to complete the transposition.*

Proof. From Equation 8, the source column of element i in column j for a C2R transposition is

$$c_j(i) = \left\lfloor \frac{(j + in)}{m} \right\rfloor \tag{27}$$

Also note that $\frac{mn}{c} = bm = an$. When we rotated the columns of the original array to enable the decomposition, we rotated groups of b columns together. Each of those b columns formed a subarray of bm elements. Now, examine groups of a rows of the array, each of which form subarrays of an elements. These subarrays have a one-to-one correspondence with the subarrays that were rotated earlier.

To see this, we will show that $\forall i \in [0, m), \forall j \in [0, n)$, $c_j(i) \in [kb, (k+1)b)$, where $k = \left\lfloor \frac{i}{a} \right\rfloor$.

First, note that $c_j(i)$ is monotonic in both i and j, so we can bound it over a domain of interest by its values at the extrema of the domain. Decompose $i = ak + y$, where $k = \left\lfloor \frac{i}{a} \right\rfloor$, and note that due to the definition of k, $y \in [0, a)$. Accordingly, $c_0(ka) \leq c_j(ka) \leq c_j(i) \leq c_j((k+1)a-1) \leq c_{n-1}((k+1)a-1)$

Evaluating the bound,

$$\begin{aligned} c_0(ka) &= \left\lfloor \frac{0 + (ka)n}{m} \right\rfloor \\ &= \left\lfloor \frac{akb}{a} \right\rfloor = kb \end{aligned}$$

Similar reasoning shows that the upper bound $c_{n-1}((k+1)a-1) = (k+1)b - 1$, Accordingly, over the domain $0 \leq j < n$, it must be true that $kb \leq c_j(i) < (k+1)b$. Then it is also true that over this domain, $\left\lfloor \frac{c_j(i)}{b} \right\rfloor = k$.

In other words, the source columns for all elements in group k were rotated by k elements.

k then establishes a one-to-one correspondence between subarrays comprised of the original columns of the array that were rotated by k places, and the rows of the array that are reading from those rotated columns.

Having established this correspondence, we need to adjust the source row indices to compensate for the rotation. Adding the term $-k = -\left\lfloor \frac{i}{a} \right\rfloor$ to the original $s_j(i)$ function counteracts this rotation. Accordingly, $s'_j(i)$ is the correct set of row indices to use for the column opersations. □

Summarizing, the C2R algorithm is performed in three steps:

- If $\gcd(m,n) > 1$: Rotate columns by gathering from each column using $r_j(i)$ from Equation 23 into a temporary vector, then copy the result over the original column.
- Row shuffle: scatter each row into a temporary vector using indices $d'_i(j)$ from Equation 24, then copy the result over the original row.
- Column shuffle: gather from each column into a temporary vector using $s'_j(i)$ from Equation 26, then copy the result over the original column.

Combining these three steps leads to a straightforward statement of the C2R transposition algorithm, using out-of-place permutations in a temporary buffer of size $\max(m,n)$. This is presented as algorithm 1.

Algorithm 1 In-place C2R transposition of array A

```
if gcd(m, n) > 1 then
  for j in [0, n) do
    for i in [0, m) do
      tmp[i] = A[r_j(i), j] {Gather per eq. 23}
    end for
    for i in [0, m) do
      A[i, j] = tmp[i]
    end for
  end for
end if
for i in [0, m) do
  for j in [0, n) do
    tmp[d'_i(j)] = A[i, j] {Scatter per eq. 24}
  end for
  for j in [0, n) do
    A[i, j] = tmp[j]
  end for
end for
for j in [0, n) do
  for i in [0, m) do
    tmp[i] = A[s'_j(i), j] {Gather per eq. 26}
  end for
  for i in [0, m) do
    A[i, j] = tmp[i]
  end for
end for
```

Figure 2 shows the state of a matrix as it is transposed using a C2R transposition. Each of the three steps corresponds to one of the three outermost loops in algorithm 1.

The R2C transposition algorithm is the inverse of the C2R algorithm. It can be derived by reversing the order of the permutation steps in the C2R algorithm and interchanging gather and scatter permutations.

Theorem 6. *The decomposed in-place transpose algorithm has optimal work complexity $O(mn)$, when given auxiliary space of $O(\max(m,n))$.*

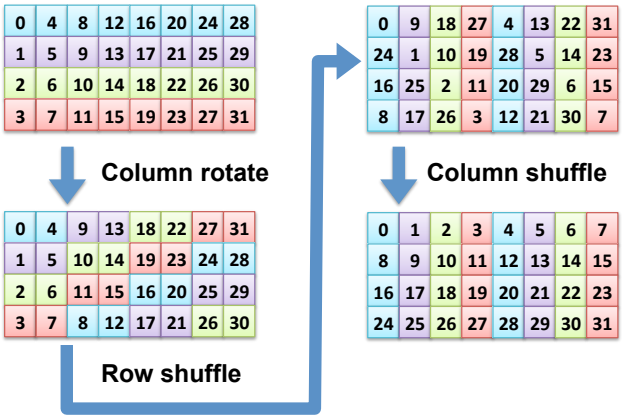


Figure 2: C2R transpose of 4×8 matrix

Proof. In the worst case, the algorithm reads and writes each element 6 times, performing row and column permutations out-of-place. This gives the work complexity of $O(mn)$, which is known to be optimal. The algorithm requires a temporary vector of size $\max(m,n)$ in order to carry out these out-of-place permutations. □

4. Optimizations

The C2R transpose shown in algorithm 1 and its R2C inverse are defined in terms of both scatter and gather based permutations on both the rows and columns of the array. Practical considerations of these algorithms may motivate the use of alternative implementations. For example, gather based formulations are sometimes more efficient, or required due to functional restrictions. Additionally, we have found it useful to restrict the column operations: rather than allowing unrestricted column shuffles, we perform the column operations using a composition of two more restricted primitives. Restricting the column operations allows us to optimize memory access patterns, and enables the in-register implementation using SIMD instructions.

We also observe that we are free to choose either row-major or column-major linearization during C2R and R2C transposes, which is an important optimization.

Theorem 7. *The linearization assumed while performing C2R or R2C transposes does not affect the permutation they induce*

Proof. Let B represent a row-major array that is created by a C2R transposition using column-major indexing on a row-major array A.

$$B[l] = A_{rm}[l_{cm}(s(i_{cm}(l), j_{cm}(l)), c(i_{cm}(l), j_{cm}(l)))] \tag{28}$$

Noting that

$$l_{cm}\left(x \bmod m, \left\lfloor \frac{x}{m} \right\rfloor\right) = x \tag{29}$$

and substituting the C2R source equations from Equations 7 and 8, as well as from Equations 16 and 17 into the indexing function above,

$$\begin{aligned} l_{cm}(s(i_{cm}(l), j_{cm}(l)), c(i_{cm}(l), j_{cm}(l))) &= \\ l_{rm}(i_{cm}(l), j_{cm}(l)) &= \\ l_{rm}(j^T_{rm}(l), i^T_{rm}(l)) & \end{aligned}$$

This proves

$$B[l] = A^{C2R}_{rm}[l] \tag{30}$$

Similar reasoning holds for using row-major indexing on a column-major array. □

Theorem 7 gives us the freedom to index arrays in row-major or column-major order, regardless of their native storage order. Although the intermediate state during the transposition differs depending on the choice of linearization used to perform C2R or R2C transposes, the fact that the final result does not depend on this choice simplifies implementation. This is an important performance optimization, since we can design the implementation so that row and column operations always run in fixed directions, regardless of whether the array was given to us in row or column major order. This enables us to optimize memory access patterns to fit cache lines.

4.1 Restricted Column Operations

Instead of implementing arbitrary column shuffles, we have found it useful to restrict column operations to column rotation and row permutation.

In column rotation, each column of the array is rotated by some rotation amount, such that the gather based index equation of the column operation is of the form $f_j(i) = (i + g(j)) \bmod m$.

In row permutation, all rows of the array are permuted, such that the gather based index equation of the column operation is of the form $f(i)$, with no dependence on the column index j. Since the rows are all permuted identically, the effect is a particular kind of column-wise permutation, where every column is permuted identically.

4.2 Columns to Rows Optimizations

As specified earlier, if $c > 1$, we first rotate by gathering indices $r_j(i)$ specified in Equation 23.

The row shuffle indices $d'_i(j)$ were specified as a scatter permutation in Equation 24. To transform it into a gather permutation, we must find its inverse $d'^{-1}_i(j)$.

We will use the modular multiplicative inverse function $\mathrm{mmi}(x, y)$, which is defined for coprime integers x and y:

$$(x \cdot \mathrm{mmi}(x, y)) \bmod y = 1$$

Define a helper function

$$\mathrm{f}(i, j) = \begin{cases} j + i(n-1) & i - (j \bmod c) + c \le m \\ j + i(n-1) + m & i - (j \bmod c) + c > m \end{cases}$$

and compute the modular multiplicative inverse $a^{-1} = \mathrm{mmi}(a, b)$. Then

$$d'^{-1}_i(j) = \left(a^{-1} \left\lfloor \frac{\mathrm{f}(i, j)}{c} \right\rfloor\right) \bmod b + (\mathrm{f}(i, j) \bmod c) \cdot b \tag{31}$$

To decompose the column shuffle given by $s'_j(i)$ in Equation 26 into a column rotation and a row permutation, we note that for gather-based permutation functions $f(i)$ and $g(i)$, gathering with indices $(f \circ g)(i)$ is equivalent to first gathering with indices $f(i)$, followed by a second gather with indices $g(i)$. Scatter-based permutations have the opposite ordering under composition. The column shuffle indices $s'_j(i)$ can be decomposed into a column rotation followed by a row permutation, where the column rotation is:

$$p_j(i) = (i + j) \bmod m \tag{32}$$

And the row permutation is:

$$q(i) = \left(i \cdot n - \left\lfloor \frac{i}{a} \right\rfloor\right) \bmod m \tag{33}$$

This decomposition of a column shuffle into these two more restricted primitives is correct because $(p_j \circ q)(i) = s'_j(i)$.

4.3 Rows to Columns Optimizations

The row shuffle step in the R2C transpose is simple when formulated as a gather, since it can just use $d'_i(j)$ directly without the need for inversion.

However, the gather-based indices for the row permute step require $q^{-1}(i)$. Compute the modular multiplicative inverse $b^{-1} = \mathrm{mmi}(b, a)$. Then

$$q^{-1}(i) = \left(\left\lfloor \frac{c - 1 + i}{c} \right\rfloor b^{-1}\right) \bmod a + (((c-1)i) \bmod c) \cdot a \tag{34}$$

Instead of perfoming a scatter rotation to invert the rotation in the C2R algorithm, we can do a gather rotation with inverted indices:

$$p^{-1}_j(i) = (i - j) \bmod m \tag{35}$$

And the final rotation indices are also inverted from the C2R pre-rotation indices:

$$r^{-1}_j(i) = \left(i - \left\lfloor \frac{j}{b} \right\rfloor\right) \bmod m \tag{36}$$

4.4 Arithmetic Strength Reduction

Evaluating the index equations, such as Equation 31, involves repeated calculations of integer division and integer modulus. We found a significant performance improvement by using a strength reduction technique that involves computing a fixed-point reciprocal, and then converting integer division into a multiplication by the reciprocal followed by a

shift [10]. The modulus can then be computed with an additional multiplication and a subtraction. This technique amortizes the calculation of the reciprocal across many repeated divisions or modulus operations.

4.5 On-chip Row Shuffle

Implementing arbitrary row shuffle operations requires two passes over each row along with the use of temporary storage, as shown in algorithm 1. If on-chip storage is sufficient, whether in caches or in register files, we can perform row shuffle operations in a single pass, without writing the intermediate result to temporary storage in memory. For example, each streaming multiprocessor on the NVIDIA Tesla K20c processor contains 256 kB of register file—in practice we found we could use this storage to process rows with up to 29440 64-bit elements in a single pass.

4.6 Cache-aware Rotate

We can improve the performance of column rotations on the array by ensuring all cache-lines read and written to and from memory are utilized efficiently. We use a row-major linearization during the transpose operations, regardless of the native linearization of the array, in order to ensure that our indexing maps to cache-lines in a canonical way.

A naive column rotation would involve reading the column from memory, then storing it in rotated order to a temporary buffer, then copying the temporary buffer back over the original column. This utilizes cache-lines poorly, especially when neighboring columns are being rotated by different amounts.

Instead of performing the rotation in this manner, we break the rotation into two phases, both of which use no temporary storage and thus save the cost of reading and writing to a temporary buffer. The first phase performs a coarse rotation in place, using cycle following. The coarseness is determined by the size of the cache-line: if we rotate groups of columns together so that the a sub-row selected from this group is one cache-line wide, we improve the efficiency of reading and writing such a sub-row. The sub-row from such a group may span one cache-line, if it happens to be aligned to cache-line boundaries, or it may span two cache-lines if it is not aligned. If the size of one row of the array is evenly divisible by the cache-line size, we are guaranteed that all sub-rows will be aligned; otherwise some sub-rows will be aligned, and others will not. In any case, reading and writing sub-rows is much more efficient than reading and writing elements from each column independently.

Cycle-following for rotation is straightforward: when rotating a vector of m elements by r places, there are $z = \gcd(m, r)$ cycles, each of which with length $\frac{m}{z}$. The elements in these cycles are also straightforward to compute analytically: $l_y(x) = (y + x(m - r)) \bmod m$, for cycle $y \in [0, z)$, and element $x \in [0, \frac{m}{z})$. Having an analytic solution to the cycles makes it straightforward to perform the coarse rotation, since there is no need to precompute cycle descriptors.

The goal of the coarse rotation is to ensure that the residual rotation for each column is bounded. This is true for the rotations we perform, since both $f(j) = \left\lfloor \frac{j}{b} \right\rfloor$ and $f(j) = j \bmod b$ have the property that $0 \leq (f(j + w) - f(j)) \bmod m < w$, where w is the width of a sub-row, or the number of columns being rotated together.

Since the residual rotation is bounded, we can then proceed with a fine in-place rotation pass that reads in the array block by block, using on-chip memory to store blocks of the array, rotate it, and write it out block by block. This ensures off-chip memory bandwidth is efficiently utilized. The fine rotation pass for a block of columns can be omitted if the residual rotation amounts for all columns in a block are identically 0. This is often the case for the C2R prerotation or the R2C postrotation performed if $c > 1$, since $r(j) = \left\lfloor \frac{j}{b} \right\rfloor$ is a slow-changing function when $b > w$.

4.7 Cache-aware Row Permute

The row permute operation can be made cache-aware through cycle-following. We do not have an analytic solution for cycles resulting from $q(i)$ and $q^{-1}(i)$, so instead we compute the cycles dynamically and store them in our temporary memory. Since all rows are permuted identically, we have only one set of cycles to compute. The number of cycles of length greater than 1 element is bounded at $\frac{m}{2}$, and so we are guaranteed to have enough storage to hold the cycle leaders and cycle length descriptors. Because cycle-following is most naturally understood through scatter permutations, we use $q^{-1}(i)$ for the C2R permutation and $q(i)$ for the R2C permutation.

As with the cache-aware rotation, this primitive operates on groups of columns chosen such that one sub-row selected from such a group is the same size as a cache-line, ensuring that reading and writing a sub-row is efficient.

5. Implementation

To test our algorithm, we wrote parallel CPU and GPU implementations and compare them against contemporary in-place matrix transposition routines. Since an ideal matrix transpose would read the array once and write the array once, we calculate throughput in this section as

$$throughput(m, n, s, t) = \frac{2mns}{t} \tag{37}$$

Where s is the size of an array element, and t is the time for the complete transposition.

5.1 Parallel CPU Implementation

Our CPU implementation of in-place matrix transposition is a straightforward OpenMP parallelization of algorithm 1. We performed two optimizations: using a completely gather based implementation using $d_i'^{-1}(j)$ during the row shuffles,

and using arithmetic strength reduction to lower the cost of index calculations. We leave cache-aware optimizations for this implementation to future work.

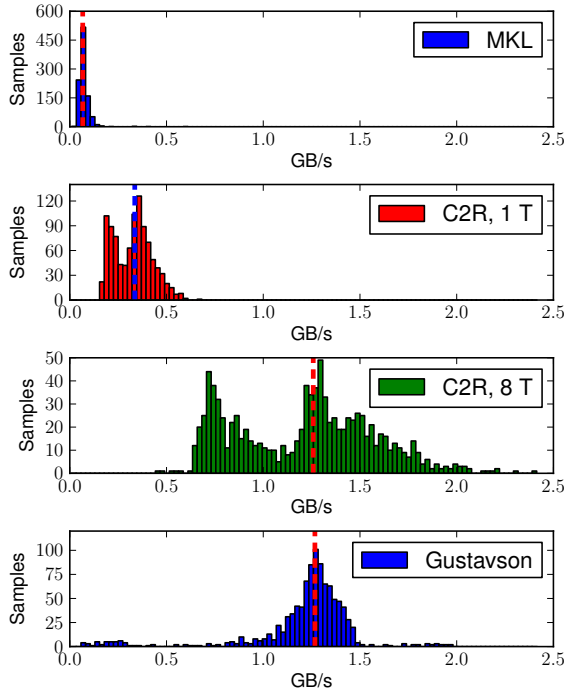


Figure 3: Throughput Histograms of In-Place Matrix Transposition CPU Implementations. Median throughput indicated with dashed line. 1 T indicates sequential execution, 8 T indicates execution with 8 threads.

Figure 3 shows histograms of in-place transpose throughput. We benchmarked transpose throughput on 1000 randomly sized matrices where m, n were chosen uniformly at random from the interval $[1000, 10000)$. Each array is comprised of 64-bit elements. We used an Intel Core i7 950 processor with 4 cores and 8 threads. We show performance for our method running sequentially, as well as with 8 threads. We also show performance of `mkl_dimatcopy()`, the corresponding method from the Intel MKL library, version 11.0.1 [2]. Median performance of our sequential implementation on this machine achieved 336 MB/s, which compares well to MKL's median performance of 67 MB/s. Additionally, since our algorithm parallelizes trivially, we were able to see a median performance of 1.26 GB/s when using 8 threads. In contrast `mkl_dimatcopy()` is not parallelized, likely due to the complexity of parallelizing traditional cycle-following algorithms. On this test set, the implementation described in Gustavson et al. [1] achieves median performance of 1.27 GB/s, including overhead for packing and unpacking the array into the tiled format required for use with their algorithm.

Median Throughputs	GB/s
Intel MKL	0.067
C2R, 1 Thread	0.336
C2R, 8 Threads	1.26
Gustavson et al. [1] (double)	1.27

Table 1: Median In-Place Transposition Throughputs on Intel Core i7 950 on Arrays of 64-bit Elements

Our performance is comparable to the performance described in Gustavson et al. [1], despite the simplicity of our implementation, which doesn't employ any of the cache-aware permutations we outlined earlier, and the complexity of Gustavson's implementation, which is highly optimized for cache accesses. Additionally, our algorithm has theoretical advantages over Gustavson's algorithm: our work complexity is $O(mn)$, while Gustavson's algorithm is $O(mn \log mn)$, given less than $O(mn)$ auxiliary space. Similarly to our algorithm, Gustavson's algorithm also requires $O(m)$ auxiliary space: arrays that are not conveniently tiled must be transformed through a packing and unpacking operation. These results are summarized in Table 1.

5.2 Parallel GPU Implementation

Our GPU implementation is also built using gather permutations, with strength reduction, and additionally uses the cache-aware permutations described in section 4.

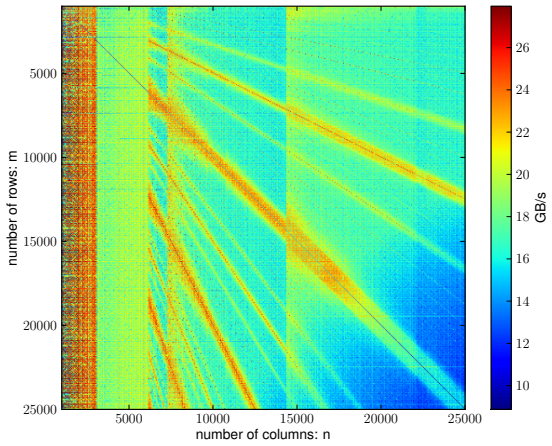


Figure 4: C2R Performance Landscape on Tesla K20c

Figure 4 shows a sampling of the performance landscape of the C2R algorithm for 250000 row-major arrays with sizes $m, n \in [1000, 25000]$. To better visualize structure, outlier samples that performed faster than the 99th percentile had their values clamped to the 99th percentile throughput. The high-performing band on the left of the graph shows that when the number of columns is small, a row fits in on-chip memory, significantly improving performance.

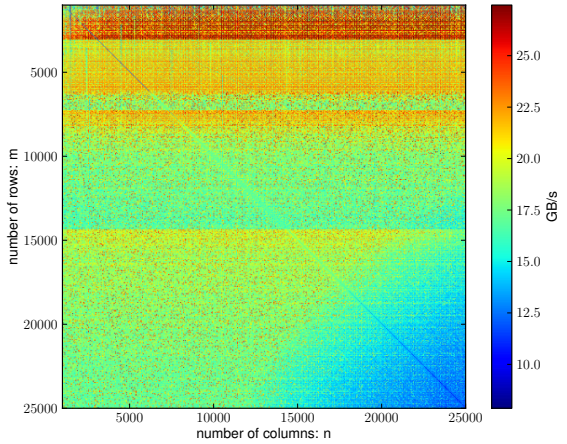


Figure 5: R2C Performance Landscape on Tesla K20c

Figure 5 shows the same performance landscape sampling as Figure 4, but using the R2C algorithm, with fast outliers clamped as explained earlier. The high-performing band on the top of the graph shows that when the number of rows is small, an entire column fits in on-chip memory.

Since the C2R and R2C algorithms can both be used for transposing any array, but their performance characteristics differ, we combined them using a simple heuristic: if $m > n$, use the C2R algorithm, otherwise use the R2C algorithm. This improves the performance of our transposition routine and makes it more efficient than either the C2R algorithm or the R2C algorithm on their own.

Figure 6 shows a histogram of throughput results for several implementations running on the NVIDIA Tesla K20c processor. We benchmarked these implementations on arrays where m, n were chosen uniformly at random from the interval $[1000, 20000)$.

First, we benchmarked the implementation from Sung [6]. We tested this implementation on 2500 arrays, of which 2155 completed correctly and are reported in Figure 6. The algorithm published in Sung [6] operates on arrays in a tiled manner, with the restriction that the dimensions of the tile must evenly divide the dimensions of the array. However, it does not choose tile sizes automatically, but instead requires the user to supply them. In order to test this code on arbitrary arrays, we used the following heuristic: sort the factors of the array dimension, then starting with the smallest factors, multiply them until the tile dimension equals or exceeds some threshold t. The motivation behind this heuristic is to find a tile size that is not too small nor too large for the hardware.

For these experiments, we set $t = 72$, so that the maximum tile size was 72×72. We did not tune over the many possible tile sizes for each array, since autotuning over data-dependent parameters is not practical for many applications. We note this heuristic was able to replicate the maximum

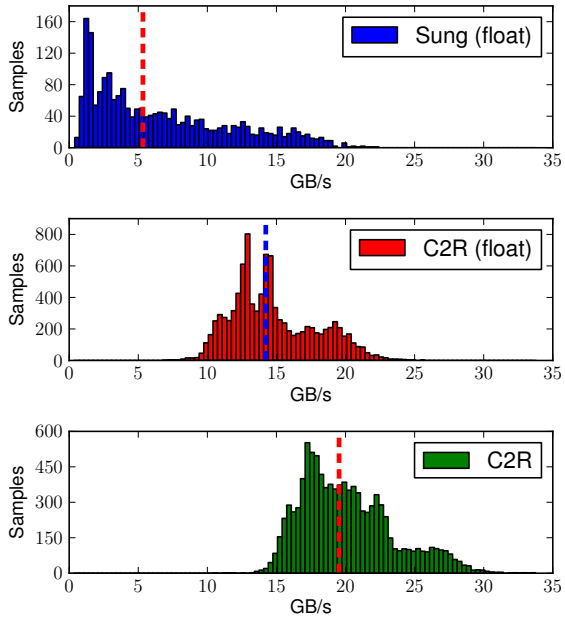


Figure 6: Throughput Histograms of In-Place Matrix Transposition GPU Implementations. Median throughput indicated with dashed line. (float) represents implementations operating on 32-bit elements

performance of 20.8 GB/s reported in Sung [6] on an array of dimension 7200×1800, with tile size 32×72. Indeed, we saw performance of 22.35 GB/s on an array of dimension 7223×10368, with tile size 31×64.

As shown in Figure 6, the implementation published in Sung [6] achieves a median throughput of 5.33 GB/s, when operating on arrays of 32-bit elements. Tiled algorithms perform poorly on arrays with inconvenient dimensions, which explains why this measurement is lower than the throughput on the six arrays measured in Sung [6]. We also note that this implementation works only on arrays of 32-bit elements, and so its throughput can only be compared to other implementations that work on arrays of 32-bit elements. Additionally, we note that the implementation in Sung [6] has auxilary space requirement of $O(mn)$, since it may in the worst case require one bit per element of the array. Our asymptotically reduced auxiliary space requirement is an important advantage of our algorithm. We refer the reader to Sung et al. [8] for further development of this algorithm and implementation.

We also provide results for our algorithm. To make a direct comparison to Sung [6], we instantiated our algorithm for arrays of 32-bit elements, and measured a median throughput of 14.23 GB/s. When operating on arrays of 64-bit elements, our implementation measured a median throughput of 19.53 GB/s. The double-precision ar-

Median Throughputs	GB/s
Sung [6] (float)	5.33
C2R (float)	14.23
C2R (double)	19.53

Table 2: Median In-place Transposition Throughputs on NVIDIA Tesla K20c on Arrays of 32-bit Elements (float) and 64-bit Elements (double)

rays transpose at higher throughput because the unstructured reads of array elements required for our row shuffle operation are more efficient when operating on 64-bit elements. These results are summarized in Table 2.

6. SIMD Vector Memory Accesses

Many algorithms mapped onto SIMD processors require vector loads and stores. Programmers often strive to increase the amount of sequential work that can be mapped onto a SIMD lane, in order to reduce the algorithmic overhead of parallelization; this requires vector loads and stores because each SIMD lane is consuming or producing a vector of data. Similarly, directly operating on Arrays of Structures (AoS) is convenient for programmers, but also requires arbitrary length vector loads and stores, as each SIMD lane loads or stores a structure. Although most processors provide limited vector loads and stores for a few fixed datatypes, using them can be inconvenient and suboptimal, since the size of a desired vector load or store may not map cleanly to the vector loads and stores provided by the hardware. Using compiler generated loads and stores for arbitrarily sized vector accesses often interacts poorly with the memory subsystem, since the vector loads and stores are implemented as a sequential series of strided memory operations, leading to poor memory bandwidth utilization.

6.1 Data Layout Conversion

As we explained in the introduction, one technique for dealing with this problem is to convert Arrays of Structures into Structures of Arrays, to eliminate the strided memory accesses. Our algorithm can perform this conversion in-place. Consider an Array of Structures of m elements, each of which is a structure containing n fields. Then the data is laid out in memory as a row-major $m \times n$ array, and transposing it into an $n \times m$ array corresponds to the Structure of Arrays data layout.

This can be done directly with the array transposition implementation we outlined earlier, but it performs poorly in practice because it is parallelized expecting m and n to both be relatively large, and for data layout conversion one of the two dimensions (the structure size) is very small, while the other (the array dimension) is very large.

We created specialized implementations of the transpose algorithm for arrays where one of the dimensions is large and the other is very small. These implementations perform all column operations in on-chip memory, since we can guarantee that the number of rows is very small by choosing the C2R or R2C algorithm appropriately.

This specialization is faster than the general implementation we described earlier, thanks to its better use of on-chip memory.

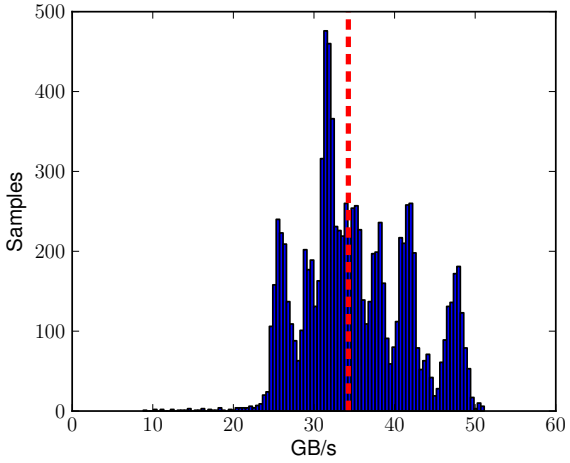


Figure 7: In-place transpose throughput for Array of Structure to Structure of Array conversion.

Figure 7 shows Array of Structures to Structures of Arrays conversion performance, where the structures are comprised of 64-bit elements. We tested performance on 10000 randomly sized Arrays of Structures, where the structure size was between $[2, 32)$ elements, and the number of structures in the array was between $[10^4, 10^7)$ We achieve median performance of 34.3 GB/s, and maximum performance of 51 GB/s.

6.2 SIMD Vector Memory Accesses

In some circumstances, Arrays of Structures do not need to be converted to Structures of Arrays at all: the transposition can be performed lazily as data is accessed. Our algorithm enables efficient, arbitrary length vector loads and stores, without the need to allocate on-chip memory to perform an out-of-place transpose.

Consider a SIMD vector of n SIMD lanes, each holding m items. This forms an array of dimension $m \times n$ in the register file. Using a shuffle instruction to interchange data between SIMD lanes, we can perform row shuffles, and using register operations locally to each SIMD lane, we can perform column operations. We will explain how this is done.

6.2.1 Row Shuffle

Most SIMD instruction sets provide a row shuffle that allows processing elements to communicate with other elements in their array. We can use this shuffle instruction directly to implement the row shuffles described in the algorithm. For

SIMD processors that do not provide a shuffle instruction, the shuffle can be simulated using a very small amount of on-chip memory that can hold one register for each SIMD lane.

6.2.2 Dynamic column rotation

In this primitive, each processing element rotates its vector by some distance, determined dynamically. Since each SIMD lane may rotate by a different amount, if this rotation were implemented with branching based on the rotation amount, this primitive would introduce SIMD divergence, dramatically reducing efficiency. To avoid this problem, we note that the rotation can be performed analogously to a barrel rotation implemented as a VLSI circuit. We can perform the rotation in-place in $\lceil \log_2 m \rceil$ steps, by statically iterating over the bits of the rotation amount, and conditionally rotating each SIMD lane's vectors by distance $d = 2^k$ at each step. This eliminates branches, even when each SIMD lane rotates its array by a different amount. This approach uses completely static register indexing, using conditional moves to perform the dynamic rotation. This comes at a cost: we must do $\lceil \log_2 m \rceil$ select instructions per element.

6.2.3 Static row permutation

In this primitive, each processing element statically permutes its vector in the same way. Since the permutation is statically known, and is constant for all processing elements, in many cases this permutation can be implemented statically without any hardware instructions: it is performed in the compiler by logically renaming elements in each column vector.

6.2.4 Implementation

This algorithm allows SIMD processors to read and write vectors of data at full memory bandwidth. Since n is constant for a given architecture, and m, the size of the structure in registers, is static, the task of computing indices can be simplified through careful strength reduction and static precomputation.

Figures 8 and 9 show the throughput vector loads and stores using this technique achieve on the NVIDIA Tesla K20c processor. The line marked "C2R" is using our transpose algorithm based on shuffle instructions to enable efficient memory accesses, and can achieve full memory bandwidth. The line marked "Vector" is using the native 128-bit vector loads and stores provided by the K20c processor. This can be efficient when the requested vector length is equal to 16 bytes, and is general more efficient than element wise loads and stores, but is not as efficient as performing the transpose. The line marked "Direct" uses compiler-generated element wise loads and stores. Figure 8 shows throughputs on unit-stride accesses, where each SIMD lane is loading or storing contiguous structures. The technique also works for random accesses to arrays of structures: in this case, indices must also be passed between SIMD lanes using shuffles. For random access, throughput improves as the size of the structure approaches the cache-line width, as shown in Figure 9.

```
T* ptr; //Private to each CUDA thread
coalesced_ptr<T> c_ptr(ptr);
T loaded = *c_ptr; //Load and R2C transpose
*c_ptr = value; //C2R Transpose and store
```

Figure 10: High level interface

Our transpose mechanism enables higher throughput on all regimes, both when performing unit strided vector memory accesses, as well as when performing randomized vector memory addresses. The performance differential can be large, up to 45× for the case of unit-stride vector stores, compared to compiler generated stores. These benchmarks illustrates the utility of this technique for processors such as the K20c.

Although we can achieve full memory bandwidth when performing transpositions in registers, there may still be cases where performing the transposition in memory is advantageous. Since our algorithm accommodates both the register-based transpose as well as the full storage transpose, the best solution for the particular application can be exploited.

6.2.5 Interface

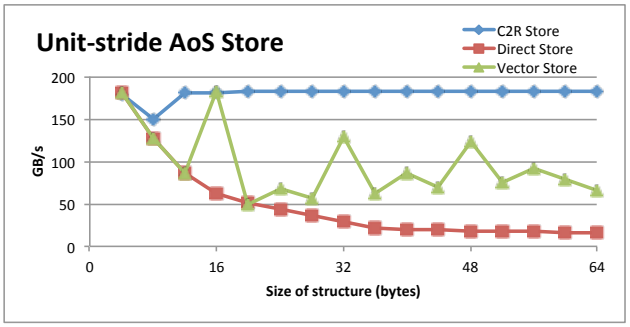
Figure 10 shows a high level interface in CUDA C++ [5] that makes use of this transpose. Because the transpose does not require allocating on-chip memory, it is straightforward to create a `coalesced_ptr<T>` wrapper type that performs transpositions internally. Every CUDA thread has its own pointer pointing to a structure of type `T` that it wishes to load or store. Directly dereferencing this pointer would lead to the undesirable performance characteristics described in the previous section. However, simply wrapping this pointer in a `coalesced_ptr<T>` type ensures that all dereferences occur through transpositions, and are therefore efficient.

7. Related Work

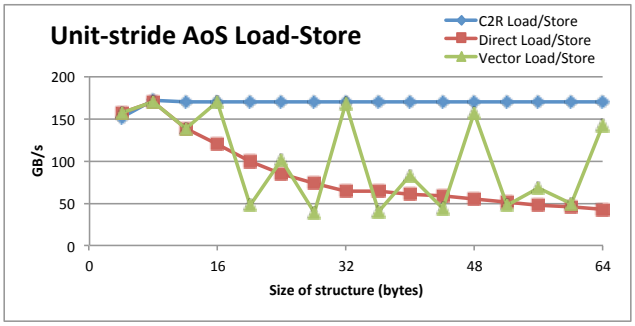
As in-place matrix transposition is a well studied field, there are several important related works with which to relate our algorithm.

We already discussed Gustavson et al. [1] and Sung [6] in Section 5.

Tretyakov and Tyrtyshnikov [9] present an algorithm for in-place matrix transposition that has optimal work complexity of $O(mn)$ while requiring only $O(\min(m, n))$ auxiliary space. This algorithm was presented without any experimental results, but we note that it requires up to 24 swaps per element, which corresponds to reading and writing each element 48 times, since each swap involves 2 reads and 2 writes. Our algorithm requires reading and writing each el-

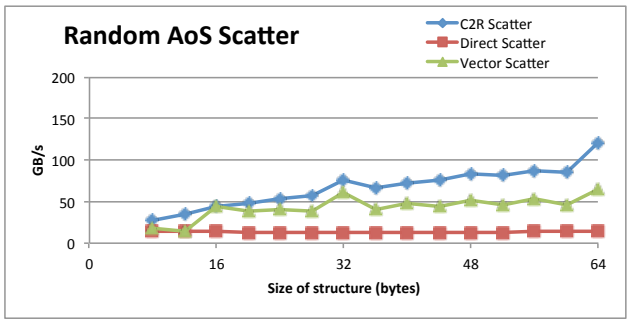


(a) Store Bandwidth

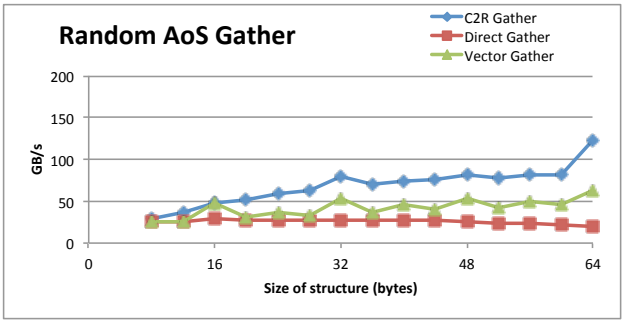


(b) Copy Bandwidth

Figure 8: Unit-stride Array of Structures Access



(a) Scatter Bandwidth



(b) Gather Bandwidth

Figure 9: Random Array of Structures Access

ement 6 times, in the worst case, which we believe gives it practical advantages. Additionally, the simplicity of our algorithm enables straightforward, efficient parallel implementation, as well as the in-register transpose that arises when SIMD processors perform vector memory accesses.

Sung et al. [7] propose a partial Arrays of Structures to Structures of Arrays transform. Because the cost of the full transposition using traditional algorithms is too high, the paper recommends modifying the program to use a hybrid Array of Structure of Tiled Array format, where the transposition cost is reduced by transposing tiles rather than elements of the array. As this introduces non-trivial complexity to the task of addressing elements of the array, the authors propose a compiler and runtime that hide this complexity from the programmer. In contrast, with our approach, we can afford to do the full transposition to convert to a true Structure of Arrays format, or alternatively, can leave the data in an Array of Structure format and perform SIMD transpositions as the data is loaded and stored.

8. Conclusion

In this paper, we have shown a decomposition for in-place matrix transposition. This decomposition has theoretical advantages compared to prior algorithms: it reduces work complexity of the transposition when auxiliary storage space is limited by reducing the scope of each permutation to a single row or column. The decomposition leads to a naturally parallel algorithm for in-place matrix transposition that has optimal work complexity of $O(mn)$, given auxiliary storage space of $O(\max(m, n))$. Our algorithm has either work or space complexity advantages over many published algorithms for in-place matrix transposition.

We have shown that the algorithm is correct, and given performance results showing its efficiency for several implementations on CPUs and GPUs. We have also shown how specializations of this algorithm can efficiently convert between Arrays of Structures and Structures of Arrays. Finally, the regular structure of our algorithm lends itself to an in-place transpose on the register file of a SIMD processor, which can allow access to Arrays of Structures at full memory bandwidth. Because the algorithm operates in-place, it is particularly easy to integrate into existing code, without requiring the user to allocate on-chip storage for the transposition. The software we have developed to illustrate this algorithm is publically available.

Acknowledgments

This research was funded in part by the DARPA PERFECT program and the U.S. Department of Energy FastForward program.

We wish to thank Manjunath Kudlur, Sean Treichler, Sean Baxter, John Owens, Brandon Lloyd, and Pace Nielsen for discussions that improved this work.

References

[1] F. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *ACM Transactions on Mathematical Software*, 38(3):1–32, Apr. 2012. doi: 10.1145/2168773.2168775.

[2] Intel. Intel MKL, 2013. URL `http://software.intel.com/en-us/intel-mkl`.

[3] D. E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, 1973. ISBN 0-201-03803-X.

[4] T. Leighton. Tight bounds on the complexity of parallel sorting. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, pages 71–80, New York, NY, USA, 1984. ACM. doi: 10.1145/800057.808667.

[5] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, pages 40–53, Mar./Apr. 2008. doi: 10.1145/1365490.1365500.

[6] I.-J. Sung. *Data layout transformation through in-place transposition*. PhD thesis, University of Illinois, Department of Electrical and Computer Engineering, May 2013. URL `http://hdl.handle.net/2142/44300`.

[7] I.-J. Sung, G. D. Liu, and W.-M. W. Hwu. DL: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar)*, May 2012. doi: 10.1109/InPar.2012.6339606.

[8] I.-J. Sung, J. Gómez-Luna, J. M. González-Linares, N. Guil, and W.-M. W. Hwu. In-place transposition of rectangular matrices on accelerators. In *Principles and Practices of Parallel Programming (PPoPP)*, PPoPP '14, 2014. doi: 10.1145/2555243.2555266.

[9] A. A. Tretyakov and E. E. Tyrtyshnikov. Optimal in-place transposition of rectangular matrices. *Journal of Complexity*, 25(4):377–384, Aug. 2009. doi: 10.1016/j.jco.2009.02.008.

[10] H. S. Warren. *Hacker's Delight*. Addison-Wesley Professional, 2002. ISBN 978-0-201-91465-8.

[11] P. F. Windley. Transposing matrices in a digital computer. *The Computer Journal*, 2(1):47–48, Jan. 1959. doi: 10.1093/comjnl/2.1.47.