# Parallel Transposition of Sparse Data Structures

Hao Wang[†], Weifeng Liu[§‡], Kaixi Hou[†], Wu-chun Feng[†]
[†]Dept. of Computer Science, Virginia Tech, Blacksburg, VA, USA, {hwang121, kaixihou, wfeng}@vt.edu
[§]Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark, weifeng.liu@nbi.ku.dk
[‡]Scientific Computing Department, STFC Rutherford Appleton Laboratory, Harwell Oxford, UK

## ABSTRACT

Many applications in computational sciences and social sciences exploit sparsity and connectivity of acquired data. Even though many parallel sparse primitives such as sparse matrix-vector (SpMV) multiplication have been extensively studied, some other important building blocks, e.g., parallel transposition for sparse matrices and graphs, have not received the attention they deserve.

In this paper, we first identify that the transposition operation can be a bottleneck of some fundamental sparse matrix and graph algorithms. Then, we revisit the performance and scalability of parallel transposition approaches on x86-based multi-core and many-core processors. Based on the insights obtained, we propose two new parallel transposition algorithms: ScanTrans and MergeTrans. The experimental results show that our ScanTrans method achieves an average of 2.8-fold (up to 6.2-fold) speedup over the parallel transposition in the latest vendor-supplied library on an Intel multi-core CPU platform, and the MergeTrans approach achieves on average of 3.4-fold (up to 11.7-fold) speedup on an Intel Xeon Phi many-core processor.

## CCS Concepts

**•Computing methodologies → Linear algebra algorithms; Parallel computing methodologies;**

## Keywords

Sparse Matrix; Transposition; CSR; SpMV; SpGEMM; Graph Algorithms; AVX; Intel Xeon Phi

## 1. INTRODUCTION

A large amount of applications in computational sciences and social sciences exploit sparsity and connectivity of acquired data. For example, finite element methods construct sparse matrices of linear systems and solve them by using direct or iterative methods [19, 40]; genomic workflows analyze the functional and connective structure of genomes assembled in various graph data structures [14, 23]. In addition, graph theory has been extensively used to analyze the sparsity of system matrices and processing them [2, 31]. Because of its importance, many sparse data structures (e.g., storage formats such as BRC [4], ACSR [3], CSR5 [34] and automatic selection tools [42]), basic linear algebra subprograms (BLAS) (e.g., sparse matrix-vector multiplication (SpMV) [8, 12, 34, 35] and sparse matrix-matrix multiplication (SpGEMM) [9, 10, 33, 38]), and graph algorithms (e.g., strongly connected components (SCC) [27, 37]) have been thoroughly researched on modern parallel platforms.

However, not all important sparse building blocks have received the attention they deserve. Computing the transposition of a sparse matrix or a graph is one such important building block. In particular, the parallel performance and scalability of the transposition operation have been largely ignored. As a result, some routines using transposition as a building block show degraded performance on parallel hardware. Such routines include, but are not limited to, transpose-based BLAS operations (e.g., sparse matrix-transpose-vector multiplication and sparse matrix-transpose-matrix multiplication) and transpose-based graph algorithms (e.g., finding strongly connected components). Later in this paper, we will further elaborate on the status of transposition in Section 2.2 and present some performance numbers in Figure 2.

In this paper, we present our research on the parallel transposition of sparse data structures, in particular, the mostly used compressed sparse row/column (CSR/CSC) formats. We first revisit the serial transposition scheme and two basic parallel transposition approaches: the atomic-based method and the sorting-based method. To overcome their shortcomings, we propose two new parallel transposition algorithms: ScanTrans and MergeTrans. We also implement the above four parallel methods on x86-based multi- and many-core systems with the latest parallel techniques. We evaluate these methods and compare with the multi-threaded sparse matrix transposition from the Intel MKL library on the Intel Haswell multi-core CPU and the Intel Xeon Phi many-core processor.

Our experimental results demonstrate that in all five aforementioned methods, ScanTrans has the best performance on Haswell, while MergeTrans has the best performance on Xeon Phi. Compared to the counterpart from the Intel MKL library, our ScanTrans method achieves up to 5.6-fold speedup and 6.2-fold speedup on Haswell for input data in single precision and double precision, respectively; while on Xeon Phi, our MergeTrans method achieves up to 11.7-fold speedup and 9.9-fold speedup, respectively.



*ICS '16, June 01-03, 2016, Istanbul, Turkey*



DOI: http://dx.doi.org/10.1145/2925426.2926291

This paper makes the following contributions:

- We identify that transposition operation can be a bottleneck of some basic sparse matrix and graph algorithms.
- We propose two new parallel transposition algorithms: ScanTrans and MergeTrans.
- We implement and optimize all known parallel transposition algorithms both on multi-core and on many-core processors.
- We conduct thorough experiments to demonstrate that the proposed methods can significantly improve the performance of parallel transposition on two very different x86 platforms, and the faster transposition can also largely accelerate higher-level algorithms.

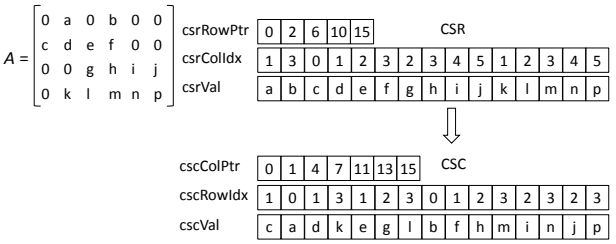
The remainder of this paper begins by describing the CSR and CSC formats and analyzing the performance issues of existing sparse transposition method on modern multi-core processors in Section 2. Next, we present the algorithm of serial sparse transposition and two parallel algorithms: atomic-based and sorting-based in Section 3. We propose ScanTrans and MergeTrans in Section 4 as well as present the parallel implementation details in Section 5. We conduct the evaluations in Section 6 and discuss the relevant related work in Section 7. We conclude by summarizing this paper and arguing that higher-level routines should consider sparse transposition as a first-class concern in Section 8.

# 2. BACKGROUND AND MOTIVATION

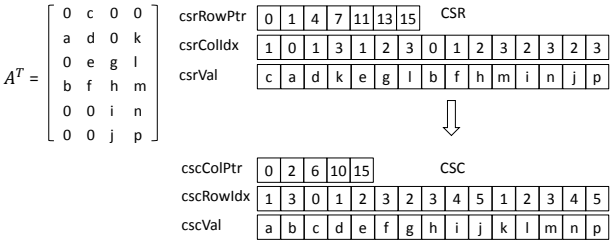
## 2.1 Preliminaries

The compressed sparse row (CSR) and compressed sparse column (CSC) representations are the most widely used storage formats for sparse matrix and graph data. The CSR format consists of three arrays: (1) csrRowPtr stores the starting and ending pointers of nonzero elements of the rows. Its length is $m + 1$, where $m$ is the number of the rows of the sparse matrix. (2) csrColIdx stores the column indices of nonzero elements. Its length is $nnz$, where $nnz$ is the number of nonzero elements in the sparse matrix. (3) csrVal array stores $nnz$ values of the nonzero elements. The number of nonzero elements in a row $i$, denoted as $nnz_i$, can be computed as csrRowPtr$[i + 1]$ − csrRowPtr$[i]$. The column indices of nonzero elements in this row can be computed as csrColIdx[csrRowPtr$[i]$], ... csrColIdx[csrRowPtr$[i + 1] - 1]$. Similar to the CSR format, the CSC format also uses three arrays, cscColPtr, cscRowIdx, and cscVal, for storing the starting and ending pointers of nonzero elements of the columns, the row indices of nonzero elements, and the corresponding values, respectively.

Matrix transposition transforms the $m \times n$ matrix $A$ to the $n \times m$ matrix $A^T$. For a matrix, the transposition is to transform the row-major storage to the column-major storage. Therefore, the CSC representation of a matrix is actually equivalent to the CSR representation of its transpose. Figure 1 illustrates the two formats for sparse matrices $A$ of size $4 \times 6$ and its transpose $A^T$ of size $6 \times 4$. We can see the equivalency in the CSR and CSC formats from both sides (Figures 1a and 1b). For brevity, in this paper, we use the statement "from CSR to CSC" to denote the transposition operation.




(a) $A$ in the CSR and CSC formats.




(b) $A^T$ in the CSR and CSC formats.

Figure 1: Transposition from $A$ to $A^T$

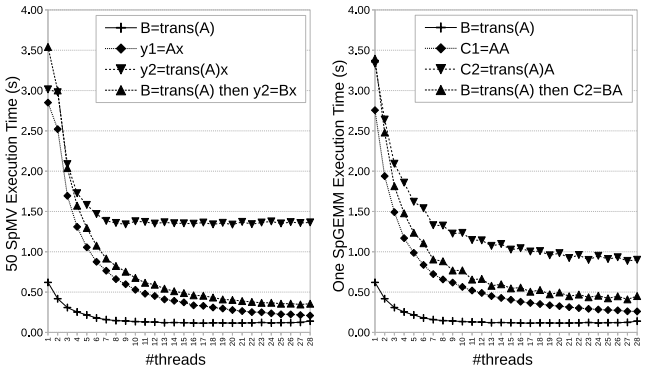
Because graphs can be represented by adjacency matrices and vice versa [16], they can share the same low-level algorithms [11]. Therefore, in this paper, we use the word "transposition" to denote *sparse matrix transposition* or *graph transposition*, *graph conversion*, or *graph reversion*.

## 2.2 Motivation

Many higher-level graph and linear algebra algorithms use sparse matrix transposition as a building block in their preprocessing and processing stages. While considering a scenario consuming a fixed graph or sparse matrix, the overhead of preprocessing (including matrix transposition) may be amortized within subsequent tens or hundreds of iterations. Some iterative linear system solvers, such as biconjugate gradient (BiCG) [20] and standard quasi-minimal residual (QMR) [22], computing sparse matrix-vector multiplication (SpMV), both on $A$ and on $A^T$, are examples of this scenario. Moreover, Buluç and Gilbert [10] utilized transposition as a part of preprocessing for fast sparse matrix-matrix multiplication (SpGEMM).

However, some other algorithms and applications do not have an iteration phase or have to process a changed sparse matrix in each iteration. For example, finding a graph's strongly connected components (SCCs) [44] depends on its sparsity structure, thus merely runs once for an input. Some recently proposed fast SCC algorithms, such as FW-BW-Trim methods [27, 37], show good scalability, but require both original graph $A$ and its transpose $A^T$ as input. In this case, the transposition operation can be a bottleneck of detecting SCCs, if it scales not so well.

Another example is the simultaneous localization and mapping (SLAM) problem [32], which is one of the most important approaches to enable an autonomous robot to explore, map, and navigate in previously unknown environments. The method acquires and analyzes a new information matrix (sparse) in each step of a whole robot trajectory, which is commonly a long process that includes a large number of steps. In an efficient SLAM implementation proposed by Dellaert and Kaess [17], the authors pointed out that com-




web-Google SpMV web-Google SpGEMM

Figure 2: Performance and scalability of SpMV and SpGEMM in the Intel MKL Sparse BLAS, running on a dual-socket 2×14-core Intel Haswell Xeon platform for matrix *web-Google*. To simulate real scenarios, SpGEMM is executed once, while SpMV is iterated 50 times.

puting $A^TA$ (where $A$ is the information matrix constructed in each step) is about *six* times more expensive than QR factorization, thus dominating the overhead of SLAM. When SpGEMM scales well by running the latest fast methods [9, 38], computing $A^TA$ will limit the overall efficiency. (Note that SLAM demands as fast as possible, even real-time, data processing.)

To further demonstrate the motivation of this work, we benchmark the most used level 2 and level 3 sparse BLAS routines, SpMV and SpGEMM, in both non-transpose and transpose patterns and see how transposition influences the performance. For the SpMV operation, we evaluate three patterns: (1) ordinary SpMV $y1 \leftarrow Ax$, (2) SpMV_T with implicit transposition $y2 \leftarrow A^Tx$, and (3) SpMV_T with an explicit transposition stage i.e., $B \leftarrow A^T$ then $y2 \leftarrow Bx$. We also run the same three patterns for the SpGEMM operations.

Figure 2 shows the latest Intel MKL sparse BLAS performance of matrix *web-Google*, downloadable from the University of Florida Sparse Matrix Collection [16]. Note that the SpMV operations are iterated 50 times to simulate a real scenario in sparse iterative solvers such as BiCG. We can see that ordinary SpMV and SpGEMM operations scale well on the used 2×14-core Intel Xeon E5-2695 v3. However, the transposition operation itself does not show good scalability. Similarly, the implicit transposition versions of SpMV and SpGEMM do not scale well. For example, the implicit SpMV_T operation stops scaling when more than eight cores are used. However, if the matrix is explicitly transposed in advance, both SpMV_T and SpGEMM_T scale as well as the ordinary versions.

On the other hand, we can see that when more cores are utilized, the cost of the transposition operation is not negligible. Taking the SpGEMM operation as an example, a serial transposition uses 620 ms when a serial SpGEMM uses 2756 ms. However, when all 28 cores are utilized, the transposition still needs 139 ms but the SpGEMM requires only 260 ms. Consequently, if the transposition operation can scale better, the performance of SpMV_T and SpGEMM_T can be naturally improved.

# 3. EXISTING METHODS

## 3.1 Serial Transposition

Serial sparse matrix transposition is straightforward to implement. For a matrix $A$ of size $m \times n$, the method uses the cscColPtr array to count the number of nonzero elements in each column. After that, the algorithm uses exclusive scan (i.e., prefix sum) on the histogram to set the starting and ending pointers in cscColPtr. Then, all nonzero elements are traversed again and moved to their final positions calculated by the column offsets in cscColPtr and the current relative positions in *curr*. Algorithm 1 illustrates the serial implementation.

**Algorithm 1:** Serial Sparse Transposition

```
Function csr2csc_serial(m, n, nnz, csrRowPtr, csrColIdx, csrVal,
cscColPtr, cscRowIdx, cscVal)
    // construct an array of size n to record current
       available position in each column
1   *curr = new int[n]();
2   for i ←0; i < m; i++ do
3       for j ←csrRowPtr[i]; j <csrRowPtr[i+1]; j++ do
4           cscColPtr[csrColIdx[j] + 1] + +;
5   // prefix_sum
6   for i ←1; i < n + 1; i++ do
7       cscColPtr[i]+=cscColPtr[i − 1];
8   for i ←0; i < m; i++ do
9       for j ←csrRowPtr[i]; j <csrRowPtr[i+1]; j++ do
10          loc = cscColPtr[csrColIdx[j]] + curr[csrColIdx[j]] + +;
11          cscRowIdx[loc] = i;
12          cscVal[loc] = csrVal[j];
13  delete[] curr;
14  return;
```

## 3.2 Atomic-based Transposition

The serial algorithm can be parallelized with multithread programming models that support atomic operations on modern processors. In Algorithm 2, we show the implementation when applying OpenMP directives. In contrast to the serial algorithm, we allocate the array *dloc* of size *nnz* to record the relative position for each nonzero element in corresponding column. We can see that multiple threads may update the same relative position stored in *dloc*. Therefore, in the first *for* loop (that counts the number of nonzero elements in each column), we use the atomic operation `fetch_and_add` for updating *dloc*. As a result, all threads obtain conflict-free relative positions and can safely run the rest calculations.

Even though atomic operations have been used in some graph algorithms [6] to support parallelism, there still remains several potential performance bottlenecks. The first one is that atomic operations are inherently serial when multiple threads try to update the same memory address. In this case, the atomic operations actually degrade overall performance. Second, the atomic operation `fetch_and_add` cannot guarantee the relative order of nonzero elements in each column. For some algorithms that strictly require the order in each column [15], a compensation using the key-value sort [41] to reorder cscRowIdx and cscVal in each column is necessary. This leads to additional overhead.

## 3.3 Sorting-based Transposition

A sorting-based serial transposition method was first proposed by Gustavson [26]. This approach sorts column indices (i.e., the array csrColIdx in the CSR format) to gather all nonzero entries with the same column index to contigu-

**Algorithm 2:** Atomic-based Parallel Sparse Transposition

```
Function csr2csc_atomic(m, n, nnz, csrRowPtr, csrColIdx, csrVal,
cscColPtr, cscRowIdx, cscVal)
    // construct an array of size nnz to record the relative
       position of a nonzero element in corresponding column
1   *dloc = new int[nnz]();
2   #pragma omp parallel for schedule(dynamic)
3   for i ←0; i < m; i++ do
4       for j ←csrRowPtr[i]; j <csrRowPtr[i+1]; j++ do
5           dloc[j] =
            fetch_and_add(&(cscColPtr[csrColIdx[j] + 1]), 1);
6   prefix_sum(cscColPtr, n + 1);
7   #pragma omp parallel for schedule(dynamic)
8   for i ←0; i < m; i++ do
9       for j ←csrRowPtr[i]; j <csrRowPtr[i+1]; j++ do
10          loc = cscColPtr[csrColIdx[j]] + dloc[j];
11          cscRowIdx[loc] = i;
12          cscVal[loc] = csrVal[j];
13  // sort cscRowIdx,cscVal in each column
14  #pragma omp parallel for schedule(dynamic)
15  for i ←0; i < n; i++ do
16      begin=cscColPtr[i];
17      end =cscColPtr[i + 1];
18      sort_key_value(being, end, cscRowIdx, cscVal);
19  delete[] dloc;
20  return;
```

ous places in the ascending order. At the same time, each nonzero entry records a position in which it will be moved to. Then the algorithm permutes the nonzero entries (i.e., their row indices and values) to the corresponding new positions. Finally, the contiguous and possibly replicated column indices are used to generate cscColPtr as part of the CSC format. If a stable sorting algorithm (i.e., method maintains the relative order of nonzero entries in $A$) is used, entries in each column of $A^T$ will be naturally ordered.

Parallel implementation of the sorting-based transposition is possible to achieve better performance than the atomic-based method since it avoids atomic transactions (which may be degraded to serial writes on csrRowPtr). Furthermore, this method may have good scalability since it works in the nonzero entry space of size $O(nnz)$, which is not related with the number of parallel threads. If a good parallel sorting method is utilized, the sorting-based transposition is expected to be efficient.

However, because of the $O(nnz \log nnz)$ complexity, the sorting-based method may encounter significant performance degradation when the total number of nonzero elements increases. Under such a circumstance, the additional memory transactions that are needed in the sorting algorithm may kill the performance.

# 4. PROPOSED TRANSPOSITION METHODS

## 4.1 Performance Considerations

To realize an efficient transposition operation, we have to consider three aspects. First, the algorithm is required to be sparsity independent. That is to say, the nonzero entries should be evenly divided to threads. Thus transposing irregular data, such as power-law graphs, will not be affected by load imbalance (e.g., some of their very long rows are assigned to one single core and short ones to the others).

Second, because the atomic operations may serialize operations expected to be parallel, they should be avoided despite their simplicity. If required, some auxiliary arrays can be allocated to eliminate race conditions. Also, the relative order of nonzero entries should be guaranteed naturally to avoid an extra stage of sorting indices inside a row/column.

Third, the work complexity should be lower than $O(nnz \log nnz)$ of the sorting-based method. Consider the serial method has linear complexity $O(m+n+nnz)$, it may be possible to design parallel methods to achieve somewhere closer to it.

## 4.2 ScanTrans

We first introduce the scan-based sparse matrix transposition called ScanTrans. The basic idea of this algorithm is to partition nonzero elements evenly among threads, count the numbers of column indices as well as the relative position of each nonzero element in corresponding column by each thread, and finally get the absolute offset of a nonzero element in output cscRowIdx and cscVal after two rounds of scan. We construct two auxiliary arrays here: a two-dimensional array `inter` of size $(nthreads + 1) * n$, and the one-dimensional array `intra` of size $nnz$. Each row $i$, $i > 0$, in `inter` stores the number of column indices observed by the thread $i - 1$. Each element in `intra` is used to store the relative offset in the column of the corresponding nonzero element by the thread. After obtaining such histograms, the algorithm applies the column-wise scan, which is called *vertical_scan* in Figure 3, on the `inter`, followed by the prefix sum on the last row of the `inter`. After that, the values in the last row of `inter` are the starting and ending pointers of row indices of the transposed matrix (cscColPtr). Finally, the algorithm calculates the absolute offset of each nonzero element in cscRowIdx and cscVal by adding the corresponding values in cscColPtr, `inter`, and `intra`.

Figure 3 shows the process of ScanTrans on the matrix $A$ shown in Figure 1. In the initial stage, the algorithm allocates memory for the `inter` and `intra` arrays and generates the row indices for nonzero elements in csrRowIdx. In the histogram step, each thread gets column indices from csrColIdx in its own partitions and sets the values of `inter` and `intra`. For example, the nonzero element $h$, whose column index is 3 and row index is 2, is the second nonzero element in column 3 of the partition proceeded by thread 1, where $f$ is the first element in column 3. Thus, the corresponding element in `intra` is set to 1 by thread 1. Because there are two nonzero elements, i.e., $f$ and $h$, in column 3 of this partition, the corresponding element 15 of `inter`, i.e., $(tid+1) * n + colIdx = (1+1) * 6 + 3 = 15$, is set to 2. With the vertical scan on the two-dimensional array `inter`, the algorithm gets the total numbers of nonzero elements in each column, "1,3,3,4,2,2" as shown in the last row of `inter`. The vertical scan also generates the numbers of nonzero elements in each column proceeded by previous threads. For example, after the vertical scan, column 3 of `inter` is "0,1,3,3,4", which means before thread 0, no element in column 3 is proceeded; before thread 1, there is one element in column 3 is proceeded; and so on. In the third step, the algorithm applies the prefix sum on the last row of `inter` and gets the start and end pointers of row indices in CSC (cscColPtr). After that, the algorithm can calculate the absolute offset in cscRowIdx and cscVal for any nonzero element. For example, for $h$ whose column index is 3 and position in csrColIdx is 7, the absolute offset in cscRowIdx and cscVal is $7 + 1 + 1 = 9$ calculated by the following equation:

$$off = \text{cscColPtr}[colIdx] + \texttt{inter}[tid * n + colIdx] + \texttt{intra}[pos].$$

Finally, its value and row index are written back to the position 9 of cscRowIdx and cscVal, respectively, which are

tagged by the gray color in the figure.

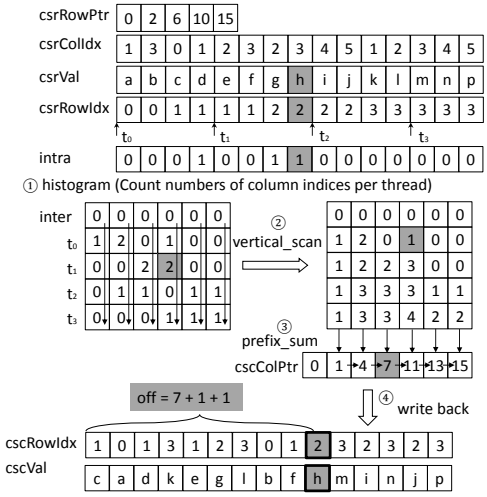



Figure 3: ScanTrans

Compared to the atomic-based sparse matrix transposition, ScanTrans avoids the usage of atomic operations, and also keeps the order of nonzero elements in each column (thus no complementary sort stage is needed). However, this method cannot avoid the random memory access. After getting the absolute offset of a nonzero element, the method copies data from csrRowIdx and csrVal to cscRowIdx and cscVal. Even though the data read on csrRowIdx and csrVal is contiguous in each thread, having the spatial locality, the data write on cscRowIdx and cscVal is totally random. It is interesting to check whether there is improved performance if we change the sequential read, random write pattern to the random read, sequential write pattern. As a result, we implement an alternative of ScanTrans: different with Algorithm 3, instead of calculating the absolute offset in cscRowIdx and cscVal, each thread calculates the absolute position for a nonzero element in csrRowIdx and csrVal reversely. Then, the data access pattern is changed to the random read on nonzero elements in CSR, and sequential write on data in CSC.

## 4.3 MergeTrans

In order to mitigate the random memory access in ScanTrans and better leverage the shared LLC (Last Level Cache) in multi- and many-core processors, we design another parallel sparse matrix transposition, called MergeTrans, as shown in Figure 4. The MergeTrans method consists of two stages, the transposition stage and the merge stage. In the transposition stage, this method partitions the nonzero elements into multiple blocks (the block number is configurable), and transposes a block of nonzero elements from CSR to CSC by using the serial algorithm 3.1 or the single threaded sorting-based algorithm. For example, if we use the sorting-based method to transpose nonzero elements in the partition of thread 0, the method (1) sorts the column indices "1,3,0,1" to "0,1,1,3"; (2) accordingly moves their values from "a,b,c,d" to "c,a,d,b" in cscVal; (3) sets the row indices of nonzero elements to corresponding positions in cscRowIdx as "1,0,1,0"; and (4) based on the sorted column indices "0,1,1,3", generates the cscColPtr "0,1,3,3,4,4,4" (one elements in the column 0, two elements in the column 1, and one elements in the column 3). If the size of a block is small enough, the random memory access can be mitigated because the data is expected to reside in the cache.

**Algorithm 3:** ScanTrans: Scan-based Parallel Sparse Matrix Transposition

```
Function ScanTrans(m, n, nnz, csrRowPtr, csrColIdx, csrVal, cscColPtr,
cscRowIdx, cscVal)
    // construct auxiliary data arrays
1   *intra = new int[nnz]();
2   *inter = new int[(nthreads + 1) * n]();
3   *csrRowIdx = new int[nnz]();
4   #pragma omp parallel for schedule(dynamic)
5   for i ←0; i < m; i++ do
6       for j ←csrRowPtr[i]; j <csrRowPtr[i+1]; j++ do
7           csrRowIdx[j] = i;
8   #pragma omp parallel
9   // partition nnz evenly on threads, get start in csrColIdx
        and len for each thread
10  for i ←0; i < len; i++ do
11      intra[start + i]=inter[(tid + 1) * n + csrColIdx[start +
        i]] + +;
12  // vertical scan
13  #pragma omp parallel for schedule(dynamic)
14  for i ←0; i < n; i++ do
15      for j ←1; j < nthread + 1; j++ do
16          inter[i + n * j]+ = inter[i + n * (j − 1)];
17  #pragma omp parallel for schedule(dynamic)
18  for i ←0; i < n; i++ do
19      cscColPtr[i + 1]=inter[n * nthread + i];
20  prefix_sum(cscColPtr, n + 1);
21  #pragma omp parallel
22  for i ←0; i < len; i++ do
23      loc = cscColPtr[csrColIdx[start + i]] + inter[tid * n +
        csrColIdx[start + i]] + intra[start + i];
24      cscRowIdx[loc] = csrRowIdx[start + i];
25      cscVal[loc] = csrVal[start + i];
26  // free intra, inter, csrRowIdx
27  return;
```

In the merge stage, the algorithm merges generated multiple CSC iteratively in parallel, until there is only one left. Two merge functions `merge_mcsc_sthread` and `merge_pcsc_mthread` are designed to merge multiple CSC by a single thread and merge a pair of two CSC by multiple threads, respectively, as shown in Algorithm 4. In Figure 4, we set the number of blocks equal to the number of threads to simplify the figure. After getting multiple CSC, this method uses two rounds of `merge_pcsc_mthread` to merge four CSC. The merge algorithm directly adds cscColPtr of two CSC to get the merged cscColPtr, and then moves nonzero elements in each column of cscRowIdx and cscVal to the column of the merged CSC with an interleaved manner. For example, in step 3, we use all four threads to merge two CSC, denoted as (t0, t1) and (t2, t3). The algorithm first gets the merged cscColPtr "0,1,4,7,11,13,15" by adding "0,1,3,5,8,8,8" and "0,0,1,2,3,5,7" correspondingly. The second and third elements "1,4" in the merged cscColPtr means there are three elements at the column 1. Two of them come from the CSC (t0, t1) because the corresponding elements in cscColPtr of CSC (t0, t1) are "1,3", while one element comes from the CSC (t2, t3) because of "0,1" in the second and third positions of cscColPtr. Thus, the algorithm copies corresponding nonzero elements *a,d* from (t0, t1) and *k* from (t2, t3) as well as their row indices. In this figure, we use two colors to tag data from two CSC; and it shows the interleaved data access pattern. Furthermore, because the algorithm can move successive data, e.g., *a,d*, to contiguous positions of the merged CSC, such sequential read and write pattern can increase

the data locality.

In our implementation, several optimizations have been applied on. Multiple successive columns may be checked to enable the block data copy. For example, after "5,8,8,8" in cscColPtr of the CSC (t0, t1) is checked, we move all nonzero elements pointed by "2,3,5,7" in the CSC (t2, t3) instead of checking them column by column. Other optimizations, including dynamic binding of thread to task, will be discussed in Section 5.3.

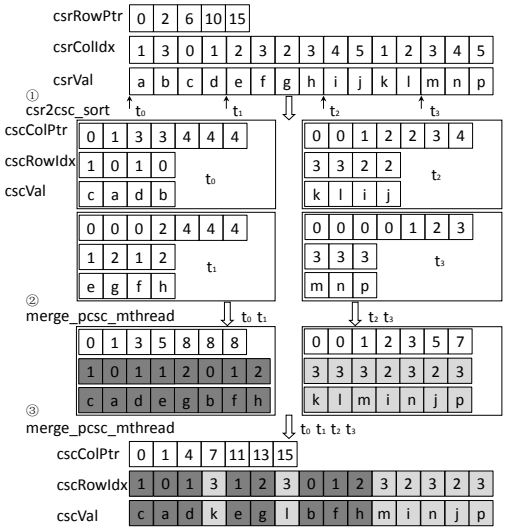



Figure 4: MergeTrans

Algorithm 4 illustrates the details of MergeTrans. The algorithm needs to construct auxiliary buffers to hold intermediate CSC. The total memory usage is $2*(nblocks*(n+1)+nnz)$ for cscColPtrA, cscRowIdxA, and cscValA, where 'A' means Auxiliary, and '2' is for the double buffering mechanism because the algorithm is not the in-place transposition [43, 13]. Compared to other algorithms, MergeTrans has better memory access patterns both in the transposition and merge stage.

**Algorithm 4:** MergeTrans: Bottom-up Transposition and Merge

```
   Function MergeTrans(m, n, nnz, csrRowPtr, csrColIdx, csrVal,
   cscColPtr, cscRowIdx, cscVal)
       // partition nnz evenly to blocks
1      // allocate cscRowIdxA, cscValA with size 2*nnz
2      // allocate cscColPtrA with size 2*nblocks*(n+1)
3      // use serial/sort transposing blocks in parallel
4      while nblock!=1 do
5          #pragma omp parallel
6          if nblocks ⩾ 2 * nthread then
7              merge_mcsc_sthread(...);
8          else
9              merge_pcsc_mthread(...);
10         return;
   // merge multi csc by single thread
11 Function merge_mcsc_sthread(nblocks, n, colPtrIn, rowIdxIn, valIn,
   colPtrOut, rowIdxOut, valOut)
12     for i ←0; i < n + 1; i++ do
13         for j ←0; j <nblocks; j++ do
14             colPtrOut[i]+ = colPtrIn[i + (n + 1) * j];
15     for i ←0; i < n + 1; i++ do
16         // sin/sout are start positions for input/output
17         // cin/cout are current positions for input/output
18         // set sin, sout, cin, cout to 0
19         sout = colPtrOut[i];
20         for j ←0; j <nblocks; j++ do
21             sin = colPtrIn[i + (n + 1) * j];
22             cin+ = colPtrIn[j * n];
23             for k ←0; k < colPtrIn[i + 1 + (n + 1) * j] − sin; k++
               do
24                 rowIdxOut[sout+cout+k] = rowIdxIn[sin+cin+k];
25                 valOut[sout + cout + k] = valIn[sin + cin + k];
26             cout+ = colPtrIn[i + 1 + (n + 1) * j] − sin;
27     return;
28 // merge a pair of csc by multi threads
29 // partition columns evenly on threads
30 // each thread works on begin to end columns
31 Function merge_pcsc_mthread(begin, end, colPtrA, rowIdxA, valA,
   colPtrB, rowIdxB, valB, colPtrC, rowIdxC, valC)
32     for i ←begin; i <= end; i++ do
33         colPtrC[i] = colPtrA[i] + colPtrB[i];
34     for i ←begin; i < end; i++ do
35         sa = colPtrA[i]; la = colPtrA[i + 1] − sa;
36         // similarly get sb, lb, sc, lc
37         for j ←0; j < la; j++ do
38             rowIdxC[sc + j] = rowIdxA[sa + j];
39             valC[sc + j] = valA[sa + j];
40         sc+ = la;
41         for k ←0; k < lb; k++ do
42             rowIdxC[sc + k] = rowIdxB[sb + k];
43             valC[sc + k] = valB[sb + k];
44     return;
```

# 5. PARALLEL IMPLEMENTATION DETAILS

## 5.1 Parallel Prefix Sum

Prefix sum is an important building blocks used by multiple transposition methods, e.g., serial, atomic-based, and ScanTrans. However, due to the data dependency in successive elements, as shown in Line 7 of Algorithm 1, the compiler directives cannot optimize such a loop efficiently. As a result, we implement a parallel scan by following the scan-scan-add strategy [45] and vectorizing it with ISA intrinsics.

Manually programming with ISA intrinsics is non-trivial, and the code may not be portable across ISAs. For example, implementing the same functionality on Intel Haswell and Intel Xeon Phi requires 256-bit and 512-bit SIMD intrinsics, respectively. Thus, we implement a framework to automatically generate the vector codes for the prefix sum function on different ISAs. The basic idea is similar to the previous research [36, 28, 29]. We implement a template function for the prefix sum, and define the data reordering and computation patterns as building blocks of the template. The framework takes the data patterns as the input, searches corresponding ISA intrinsics, e.g., load, store, add, permutation, shuffle, etc., and selects intrinsics that can implement desired patterns to construct the building blocks. In current Intel architectures, a vector register consists multiple lanes, each of which has multiple elements, e.g., a 256-bit wide register of Intel Haswell has two lanes and each lane has four 32-bit elements. We use the intra-lane and inter-lane intrinsics to implement the desired pattern.

Figure 5 shows the data reordering and computation patterns of the prefix sum on vector registers. On the vector registers whose width is $w$, the in-register prefix sum needs $logw$ steps and each step $i$ consists of four stages: (1) intra-lane shuffle that rotates $2^i$ positions to left in each lane, (2) inter-lane permute that shifts lanes to left, (3) blend two vectors that will mask $2^i$ elements of the first vector, and (4) add two vectors. When $w$ is 8, there are 3 steps to implement the in-register prefix sum. For the step 0, after loading 8 elements into the register $v_0$, the stage 1 will rotate $2^0$ po-

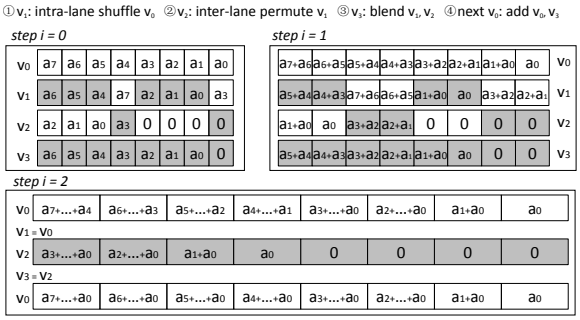



Figure 5: Prefix Sum: in-register data permutation and computation patterns (on registers with 8 elements)

sition to left and set the output into the register $v_1$. The stage 2 will shift lanes in $v_1$ to left and set the output into the register $v_2$. The stage 3 will blend $v_1$ and $v_2$ by masking $2^0$ position in each lane of $v_1$. In this case, the stage 3 will select $a_6$, $a_5$, and $a_4$ from $v_1$ and $a_3$ from $v_2$. The output is set into the register $v_3$. The method will add $v_0$ and $v_3$ to $v_0$ in the stage 4, and enter the stage 1 of next step. Note that, in the step 2, we can skip the stage 1 and 3, because the stage 1 in the step 2 that rotates $2^2$ positions to left in each lane will make $v_1$ to be equal to $v_0$, and the stage 3 in the step 2 that masks $2^2$ positions in each lane of $v_2$ will make $v_3$ to be equal to $v_2$. In such cases, for this step, we directly comment out the corresponding lines of the generated code and change the names of register variables in the following lines. The patterns are same on vector registers whose width is 16 (Intel Xeon Phi), but the number of steps is changed from 3 to 4. Once we get the patterns, we set the patterns into the framework to search ISA intrinsics and generate the vector codes on the targeted ISA.

## 5.2 Parallel Sort

We implement the sorting-based transposition algorithm following a sort-merge manner: (1) halve data into two segments until the segment meets the predefined threshold, (2) sort each segment, and (3) merge segments with a multi-way merge manner until one segment is left. We implement each stage in parallel with multiple threads. When sorting a segment in one thread, we load data into vector registers and use the bitonic sort to sort data, because the vectorized bitonic sort has shown good performance and scalability on multi- and many-core processors in previous research [41, 30, 28]. In the merge stage, when the number of segments is larger than the number of threads, we use multiple threads to merge one pair of sorted segments. In this case, we use the mergepath method [24] to partition two segments, making each thread to merge same sized data and the merged data in each thread to be ordered among threads.

## 5.3 Dynamic Scheduling

In multiple transposition methods, e.g., the sorting-based method and MergeTrans method, we need to schedule threads for data chunks. The dynamic schedule directives of OpenMP are used to optimize the *for* loops in the atomic-based method and ScanTrans method, but they are not efficient to schedule threads for blocked tasks, e.g., the sort and merge tasks in the sorting-based method, and the transposition tasks and merge tasks in MergeTrans. For such cases, we use OpenMP tasking mechanism introduced in OpenMP 3.0 to dynamically schedule threads to execute tasks. The tasking of the OpenMP runtime system has supported the dynamic task binding, even for the recursive algorithms. Better load balance is expected by using this mechanism.

Because the column indices in each row of CSR are already in order, we only need to merge rows to get the sorted column index array. However, we did not observe better performance of the merging-based method in our evaluation. The major reason is because the numbers of nonzero elements in different rows are usually not equal, this method has the imbalance problem even with the OpenMP tasking. As a result, we only show the performance numbers of the sorting-based method in Section 6.

# 6. EXPERIMENTS

## 6.1 Experimental Setup

In this paper, we evaluate five parallel transposition methods: (1) the sparse BLAS method `mkl_sparse_convert_csr` with the inspector-executor pattern from the latest Intel MKL 11.3, (2) the atomic-based method from the graph analysis package [27], (3) the sorting-based method optimized by using the parallel sort with SIMD [28], (4) the proposed ScanTrans method described in Section 4.2, and (5) the proposed MergeTrans method described in Section 4.3. We benchmark the above five methods on Intel Haswell (HSW) and Knights Corner (KNC), respectively. The HSW node is a dual-socket Intel Xeon E5-2695 v3 multi-core platform (2×14-core Haswell, 2.3 GHz, 128 GB ECC DDR4, 2×68.3 GB/s), and the KNC node has an Intel Xeon Phi 5110P many-core processor (60 cores, 1.05 GHz, 8 GB GDDR5, 320 GB/s). All methods are compiled using Intel compiler *icpc* 16.1. On HSW, we use the compiler option *-xCORE-AVX2* to enable AVX2. On KNC, we run the experiments using the *native* mode and compile the codes with *-mmic*. All codes in our evaluations are optimized in the level of *-O3*.

To better cover various real-world scenarios, we transpose a given matrix stored in three data types: symbolic (i.e., no value thus only sparsity structure is transposed), 32-bit single precision and 64-bit double precision. But note that the Intel MKL parallel transposition routine does not support symbolic transposition thus this group of numbers are leaved blank.

## 6.2 Benchmark Suite

The University of Florida Sparse Matrix Collection [16] now includes near 3000 matrices from a variety of research fields, but most of them are symmetric. To benchmark our work described in this paper, we choose 21 unsymmetric matrices from a broad of applications such as computational fluid dynamics, communication network, economics, and DNA electrophoresis. We also add the matrix *Dense* to the suite to better understand performance behavior of transposition algorithms, even though it is not downloadable from the Collection and is actually symmetric. Table 1 shows the 22 matrices.

## 6.3 Transposition Performance

Figure 6 shows transposition performance, i.e., archived bandwidth in GB/s, of the above five methods on the multi-core CPU platform running for the 22 matrices in the three

Table 1: The list of selected matrices

| Name | #Rows/Cols | #Nonzeros | Kind |
|---|---|---|---|
| ASIC_680k | 682,862 | 3,871,773 | Circuit simulation |
| cage14 | 1,505,785 | 27,130,349 | Weighted directed graph |
| circuit5M | 5,558,326 | 59,524,291 | Circuit simulation |
| Dense | 2,000 | 4,000,000 | Dense matrix in CSR format |
| Economics | 206,500 | 1,273,389 | Economic |
| eu-2005 | 862,664 | 19,235,140 | Directed graph |
| flickr | 820,878 | 9,837,214 | Directed graph |
| FullChip | 2,987,012 | 26,621,990 | Circuit simulation |
| language | 399,130 | 1,216,334 | Weighted directed graph |
| memchip | 2,707,524 | 14,810,202 | Circuit simulation |
| para-4 | 153,226 | 5,326,228 | Semiconductor device |
| rajat21 | 411,676 | 1,893,370 | Circuit simulation |
| rajat29 | 643,994 | 4,866,270 | Circuit simulation |
| sme3Dc | 42,930 | 3,148,656 | Structural problem |
| Stanford_Berkeley | 683,446 | 7,583,376 | Directed graph |
| stomach | 213,360 | 3,021,648 | 2D/3D problem |
| torso1 | 116,158 | 8,516,500 | 2D/3D problem |
| transient | 178,866 | 961,790 | Circuit simulation |
| venkat01 | 62,424 | 1,717,792 | Computational fluid dynamics |
| webbase-1M | 1,000,005 | 3,105,536 | Weighted directed graph |
| web-Google | 916,428 | 5,105,039 | Directed graph |
| wiki-Talk | 2,394,385 | 5,021,410 | Directed graph |

data types. Figure 7 plots the same groups of performance numbers on the Intel Xeon Phi device.

We can see that the performance of atomic-based method highly depends on matrix sparsity structure. It can be very fast if the input matrix is relatively regular and has less write conflicts on the same memory addresses. Taking the matrix *memchip* as an example, the atomic-based method achieves the best performance and obtained 17.2% and 33.9% performance improvement over the ScanTrans and the MergeTrans methods (which are the second fastest) on CPU and Xeon Phi, respectively. But for irregular inputs (e.g., matrix *FullChip* in a power-law shape), the atomic-based method gives much lower performance (up to 16.6x as slow as the MergeTrans method on the Xeon Phi).

In contrast, the sorting-based method gives relatively stable but modest efficiency. Because of its $O(nnz \log nnz)$ complexity, the method achieves stable bandwidth throughput. For example, on the CPU platform, single precision transposition using the method almost always obtains 1 GB/s bandwidth utilization, and double precision version almost always stands on 1.5 GB/s. Moreover, we can see that although this approach never achieves the best performance in the benchmark suite, it is on average faster than the atomic-based method.

On the CPU platform, the ScanTrans method is the clear winner. It outperforms all the other methods on all matrices except *memchip*. Compared to multi-threaded Intel MKL transposition, the ScanTrans method can achieve up to 5.6-fold and 6.2-fold speedup on *circuit5M* and *wiki-Talk* for single precision and double precision, respectively. Compared to the atomic-based method, the better performance of ScanTrans comes from two aspects: (1) ScanTrans does not need the atomic operations; (2) ScanTrans does not require additional segmented sort. Compared to the sorting-based method, ScanTrans has the time complexity $O((nnz/p) + (n/p) + n)$, where $p$ is the number of parallel threads; and the sorting-based method with $p$ threads has the time complexity $O(nnz/p \log nnz/p)$ for sort and $O(nnz/p \log p)$ for merge, which is larger than that of ScanTrans when $nnz$ is larger than $n$.

Because the cache miss penalty on Intel Xeon Phi is much higher than that on multi-core CPU [39], the ScanTrans method has obvious performance degradation. Under such a circumstance, the MergeTrans method that mitigating the cache miss has become the overall winner. Compared to the counterpart from Intel MKL, the MergeTrans can achieve up to 11.7-fold and 9.9-fold speedups for the single precision and double precision both on *wiki-Talk*. Compared to its performance on Intel Xeon Phi, the MergeTrans performs relatively poor on Intel Haswell, as shown in Figure 6. This is because the MergeTrans uses multiple iterations to move nonzero elements from two input CSC to the merged one, and more memory accesses are needed compared to the ScanTrans. On Intel Haswell, the benefit of eliminating irregular memory access patterns and avoiding accompanying cache miss penalties in the MergeTrans is offset by extra memory access operations. This can also explain the observation on Intel Xeon Phi: when the matrices become dense, the ScanTrans will retake the leadership from the MergeTrans, as shown in *sme3Dc*, *torso1*, and *dense* of Figure 7.

Note that in our experiments, we tuned all five methods with different values of parameters, e.g., the thread affinity and the number of threads, and showed their best performance numbers in the figures. On Xeon Phi, the best performance could be achieved for all methods when we set the thread affinity type to *balanced* and the number of threads to 60. On Haswell, the thread affinity type *compact* could provide better performance than *scatter*, while the numbers of threads achieving the peak performance were diverse. We observed that the ScanTrans method could achieve the peak performance when using 16 - 22 cores, while the Intel MKL transposition stopped scaling after 8 - 14 cores. For example, for *torso1* with single precision, the Intel MKL method scaled up to 14 cores, and the ScanTrans method could obtain 2.5-fold bandwidth when using 22 cores. Although the scalability of parallel transposition for sparse data structures is improved by the ScanTrans method, it is still an issue on multi-core processors and further optimizations are needed.

## 6.4 Transposition in Higher Level Routines

As sparse matrix transposition is normally used as a building block for higher level problems, we construct a routine suite composed of four representative graph and linear algebra algorithms requiring transposition operation: (1) sparse matrix-transpose-matrix addition, i.e., $C = A^T + A$, (2) sparse matrix-transpose-vector multiplication, i.e., $y = A^T x$, (3) sparse matrix-transpose-matrix multiplication, i.e., $C = A^T A$, (4) strongly connected components problem, i.e., SCC($A$). We briefly introduce them below.

(1) The operation $A^T + A$ adds two sparse matrices $A$ and its transpose $A^T$. Applications of this fundamental operation include symmetrizing unsymmetric sparse matrices for graph partitioning [6] and sparsity preserving pre-ordering (such as the approximate minimum degree (AMD) ordering [2]) in prior to numerical factorizations.

(2) The operation $y = A^T x$ multiples a matrix $A^T$ and a dense vector $x$. It is commonly used in iterative methods for sparse solvers such as BiCG and QMR. Because of this, we

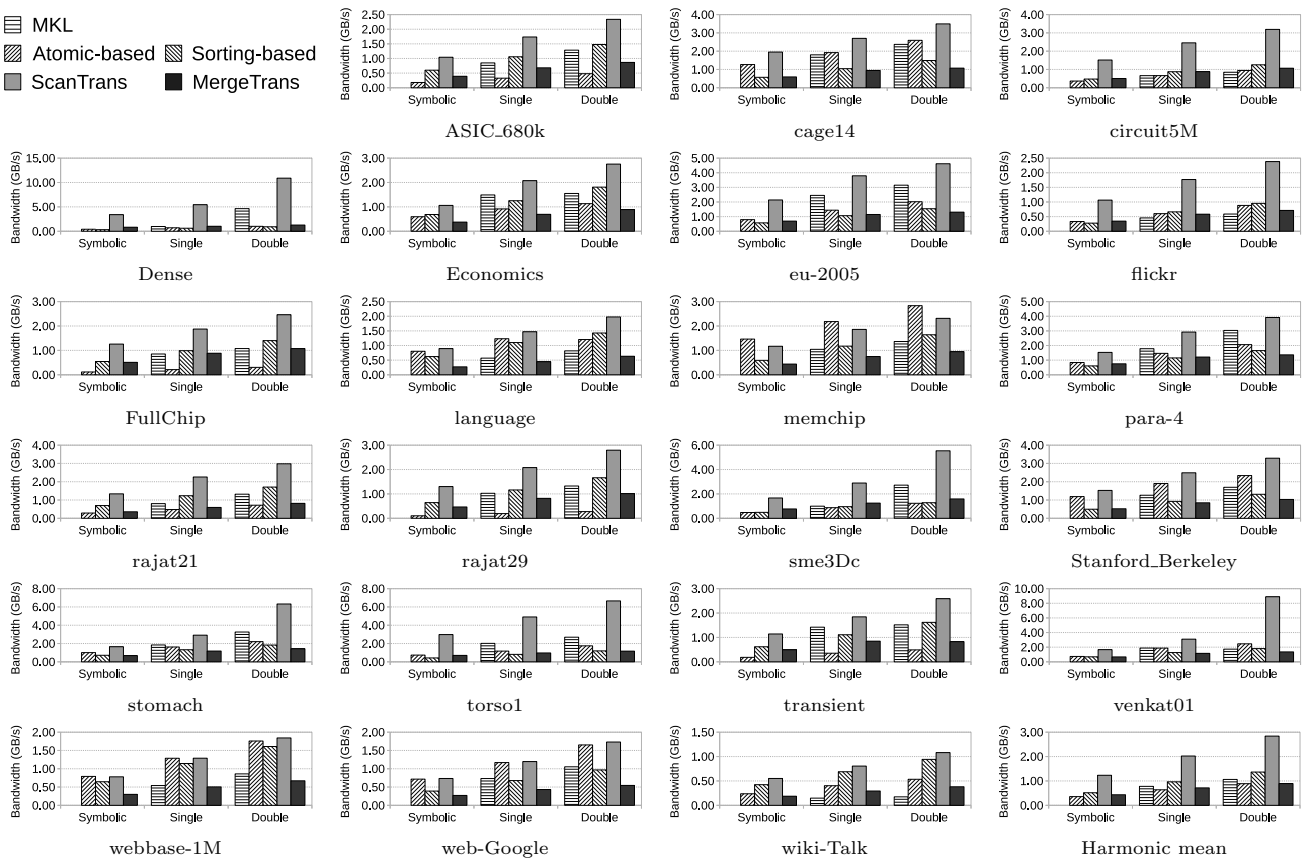



Figure 6: Bandwidth utilization of transposing the 22 matrices using five parallel algorithms on the dual-socket Intel Haswell Xeon platform.

choose a scenario requires 50 iterations and report 50 SpMV cost.

(3) The operation $A^TA$ multiplies a sparse matrix's transpose $A^T$ and itself. Both $A^TA$ arises in sparse least-squares problems [7] (e.g., the above mentioned SLAM method [17] in robotics). Moreover, finite element assembly problem can also be implemented by calculating $A^TA$ [26].

(4) The operation SCC($A$) detects strongly connected components in a given directed graph $A$. An SCC is a maximal subgraph where there exists a path between any two vertices in the subgraph. The classic sequential SCC method designed by Tarjan [44] only works on $A$. But another classic serial algorithm from Kosaraju [1] and several efficient parallel SCC methods [27, 37] require connection information from both $A$ and $A^T$.

Figures 8 and 9 demonstrate performance improvement while using the ScanTrans and MergeTrans methods proposed in this paper. For brevity, we only select four matrices (*Economics*, *stomach*,*venkat01* and *web-Google*) from our benchmark suite of 22 matrices. We compare the parallel transposition method in the MKL library with ScanTrans and MergeTrans methods on the CPU and Xeon Phi platforms, respectively. In Figure 2, we already show that the explicit methods (i.e., transpose in advance and conduct SpMV or SpGEMM computations) are much faster than the implicit methods. Therefore we only evaluate the explicit methods from the newest Intel MKL library.

We can see that all scenarios achieve better performance when using our methods. Taking matrix *venkat01* as an example, the SCC operation obtains 202% performance improvement by our ScanTrans method on the CPU platform. On average, the four matrices achieve 26.8%, 30.7%, 26.2% and 62.7% improvement on the CPU device. On the Xeon Phi, the overall performance improvement is also noticeable.

## 6.5 Discussion

**Symmetric Matrix:** Except for the matrix *Dense*, all matrices used in our evaluations are unsymmetric. We did not show the performance numbers for symmetric matrices in the figures because sparse matrix algorithms usually use different solvers to handle symmetric and unsymmetric matrices, and those for symmetric matrices may process the transposition implicitly [21, 46]. Actually, in our observation, the proposed ScanTrans and MergeTrans can also provide competitive benefits for symmetric cases. For example, for the symmetric matrices *delaunay_n21*, *hollywood-2009*, and *gupta3* from [16], ScanTrans can achieve 1.4-fold, 3.4-fold, and 4.0-fold speedups over the multi-threaded Intel MKL version on Haswell, respectively; and MergeTrans can deliver 5.4-fold, 3.6-fold, and 5.0-fold speedups on Xeon Phi.

**Auto-Selection:** Although ScanTrans and MergeTrans illustrate best performance on Intel Haswell and Xeon Phi, respectively, it is ideal to provide an adaptive method that can automatically select a proper transposition method for

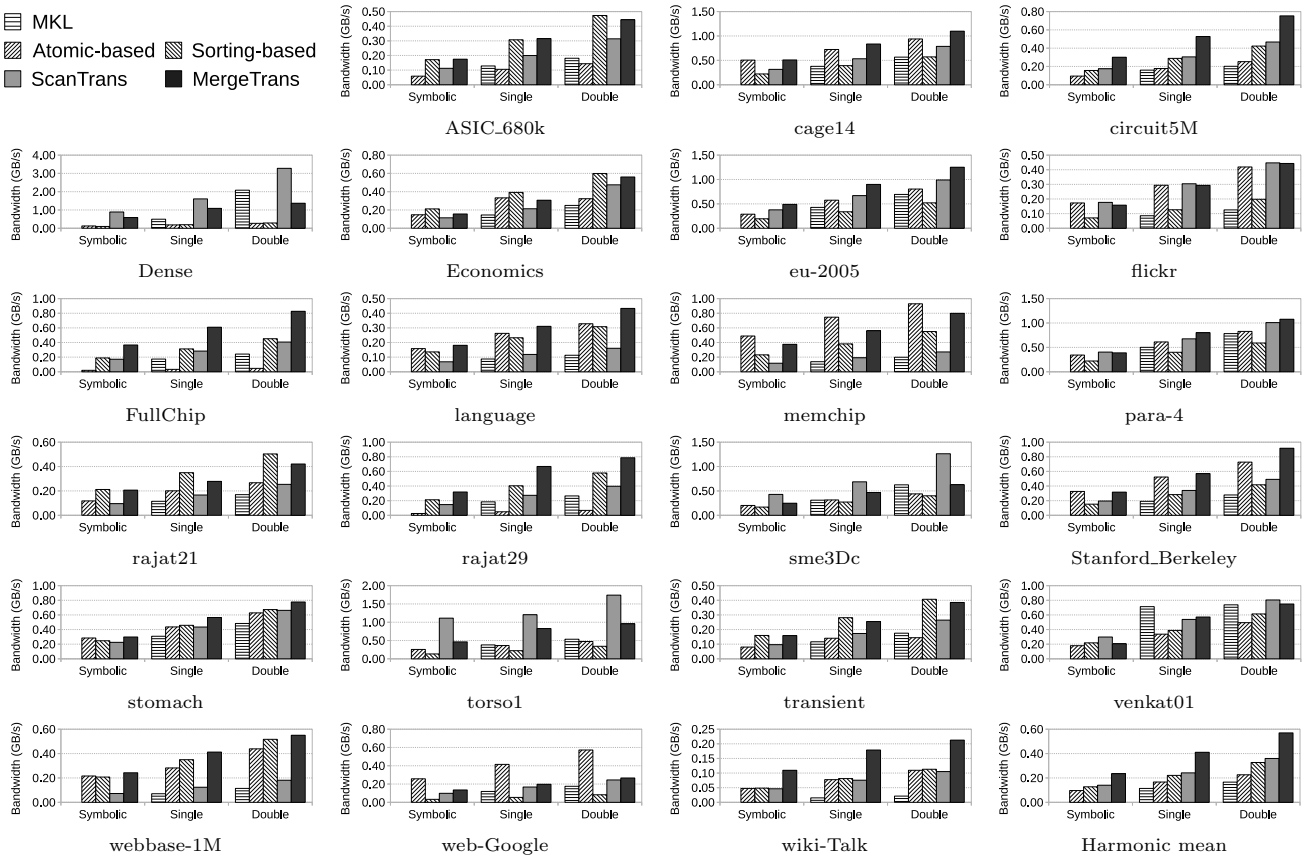



Figure 7: Bandwidth utilization of transposing the 22 matrices using five parallel algorithms on the Intel Xeon Phi processor.

more sparse matrices and hardware platforms. It is non-trivial because the performance depends on combinations of many configurable factors, including the characteristics of the input sparse matrix $A$, e.g., m, n, nnz, sparsity, deviation, etc., the targeted platforms, e.g., architecture of processors, cache and memory hierarchy, etc., the parameters in the transposition methods, e.g., predefined segment size in the sorting-based method, configurable number of blocks in MergeTrans, etc. We have observed many graph applications have put the sparse matrix transposition kernel inside *for* loops before calling the SpMV and SpMM_T because the $A$ and $A^T$ are updated within each iteration. For these applications, it is possible to switch between different transposition methods by using performance data and statistical data from earlier calls to the transposition function. We leave the research for the adaptive transposition selection mechanism in the future.

# 7. RELATED WORK

**Parallel transposition of dense matrix** [25, 13, 43] has received much attention because of its widespread use in leveraging the SIMD working pattern and accelerating dense numerical linear algebra. Dotsenko et al. [18] pointed out that explicitly transposing dense data can significantly increase the overall performance of parallel FFT. In this paper, we show that transposing sparse data structures requires more complex algorithm design and can improve many higher-level routines as well.

A serial method for **transposing sparse matrix** has been designed by Gustavson [26]. The approach uses stable sorting as the main primitive. Even though its parallel implementation scales well on modern multi-core and many-core machines when fast parallel sorting algorithm [28] is used. It does not give the best observed performance in our experiments because of inherently more memory transactions.

Because the historically high cost of transposition, several **transposition-avoid algorithms and data structures** have been implemented. Freund developed a method [21] to avoid SpMV_T in quasi-minimal residual (QMR) algorithm. Buluç et al. [8, 12] proposed a unified sparse matrix format called compressed sparse blocks (CSB) to accelerate both SpMV and SpMV_T operation in one single framework. However, it may be hard to find general approaches to redesign algorithms and data structures for avoiding transposition in a range of sparse matrix and graph applications. In contrast, the work described in this paper transposes the widely used CSR/CSC format thus can be easily adopted by a majority of existing libraries.

Transposition can be implemented by using some **parallel primitives**. Ashkiani et al. [5] recently designed a multisplit method which in theory can be used for sparse matrix transposition. In their experiments, the new method behaves very well when the number of buckets is 64 or smaller. When the number of buckets grows to 65536, the new method is actu-

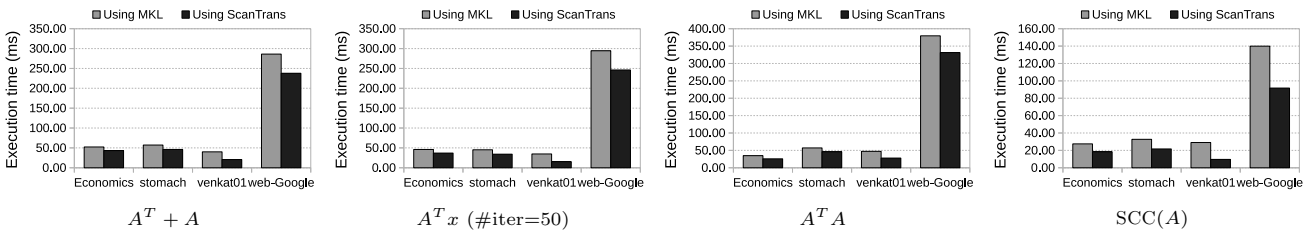



Figure 8: Execution time of the four higher level routines using the two transposition algorithms on the dual-socket Intel Haswell Xeon platform.

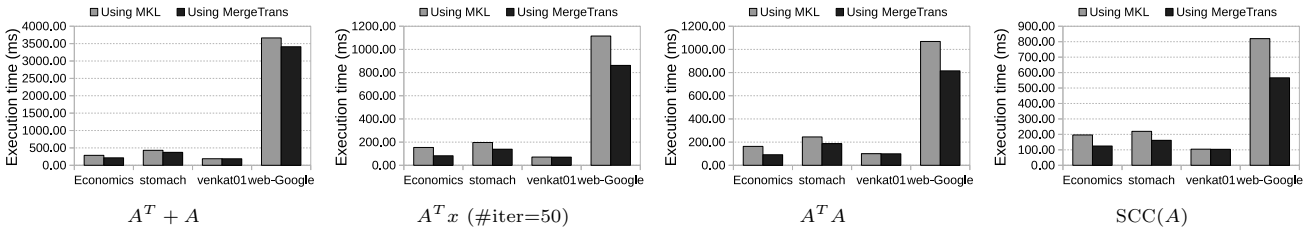



Figure 9: Execution time of the four higher level routines using the two transposition algorithms on the Intel Xeon Phi processor.

ally slower than sorting-based method. However, the number of buckets in sparse matrix transposition must be the number of rows/columns, which is up to multiple millions in our test suite. In this scenario, performance of using multisplit method decreases very fast and can be further slower than sorting-based method. In contrast, we show that the ScanTrans and MergeTrans methods proposed in this paper can be much faster than the sorting-based method. Moreover, we believe that our method can be potentially generalized to further accelerate the multisplit problem with a large amount of buckets.

# 8. CONCLUSIONS

In this paper, we proposed two new methods, ScanTrans and MergeTrans, for parallel sparse matrix and graph transposition, and implemented and optimized them for modern x86-based multi- and many-core processors. By using 22 sparse matrices from diverse application domains, we evaluated the two methods with the existing parallel transposition methods, including the atomic-based method, the sorting-based method, and the routine from the Intel MKL library, on Intel Haswell multi-core CPU and Xeon Phi many-core processor. Our experimental results showed that ScanTrans has the best performance on Intel Haswell CPUs, while MergeTrans has the best performance on Intel Xeon Phi. Compared to the counterpart from the Intel MKL library, the ScanTrans method achieved up to 5.6-fold and 6.2-fold speedups on Intel Haswell for single and double precision datatype, respectively; while on Intel Xeon Phi, the MergeTrans method achieved up to 11.7-fold and 9.9-fold speedups.

Because of the observed performance improvement on higher level routines using the new transposition methods, we believe more computation kernels for sparse matrix computations or graph processing can obtain benefits from this work.

# 9. ACKNOWLEDGEMENT

This research was supported in part by the NSF BIGDATA program via IIS-1247693, the NSF XPS program via CCF-1337131, the Department of Computer Science at Virginia Tech, and the Elizabeth and James Turner Fellowship. This research was also supported in part by the EU's Horizon 2020 program under grant number 671633. Finally, we acknowledge Advanced Research Computing at Virginia Tech for access to high-performance computational resources and thank Kaiyong Zhao for helpful discussions about the simultaneous localization and mapping (SLAM) problem.

# 10. REFERENCES

[1] A. V. Aho, J. E. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[3] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan. Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 781–792. IEEE, 2014.

[4] A. Ashari, N. Sedaghati, J. Eisenlohr, and P. Sadayappan. An Efficient Two-Dimensional Blocking Strategy for Sparse Matrix-vector Multiplication on GPUs. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 273–282. ACM, 2014.

[5] S. Ashkiani, A. Davidson, U. Meyer, and J. D. Owens. GPU Multisplit. In *Proceedings of the 21st ACM*

*SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16. ACM, 2016.

[6] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner. *Graph Partitioning and Graph Clustering*. American Mathematical Soc., 2013.

[7] Å. Björck. *Numerical Methods for Least Squares Problems*. Society for Industrial and Applied Mathematics, 1996.

[8] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244. ACM, 2009.

[9] A. Buluç and J. R. Gilbert. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *Proceedings of the 37th International Conference on Parallel Processing*, ICPP '08, pages 503–510. IEEE, 2008.

[10] A. Buluç and J. R. Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing*, IPDPS '08, pages 1–11. IEEE, 2008.

[11] A. Buluç and J. R. Gilbert. The Combinatorial BLAS: Design, Implementation, and Applications. *International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[12] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-Bandwidth Multithreaded Algorithms for Sparse Matrix-Vector Multiplication. In *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '11, pages 721–733. IEEE, 2011.

[13] B. Catanzaro, A. Keller, and M. Garland. A Decomposition for In-place Matrix Transposition. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 193–206. ACM, 2014.

[14] D. Church, V. Schneider, K. Steinberg, M. Schatz, A. Quinlan, C.-S. Chin, P. Kitts, B. Aken, G. Marth, M. Hoffman, J. Herrero, M. L. Mendoza, R. Durbin, and P. Flicek. Extending Reference Assembly Models. *Genome Biology*, 16(1):13, 2015.

[15] T. A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[16] T. A. Davis and Y. Hu. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, dec 2011.

[17] F. Dellaert and M. Kaess. Square Root SAM: Simultaneous Localization and Mapping via Square Root Information Smoothing. *The International Journal of Robotics Research*, 25(12):1181–1203, 2006.

[18] Y. Dotsenko, S. S. Baghsorkhi, B. Lloyd, and N. K. Govindaraju. Auto-tuning of Fast Fourier Transform on Graphics Processors. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 257–266. ACM, 2011.

[19] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., 1986.

[20] R. Fletcher. Conjugate Gradient Methods for Indefinite Systems. In *Numerical Analysis*, Lecture Notes in Mathematics, pages 73–89. Springer Berlin Heidelberg, 1976.

[21] R. W. Freund. A Transpose-Free Quasi-Minimal Residual Algorithm for Non-Hermitian Linear Systems. *SIAM Journal on Scientific Computing*, 14(2):470–482, 1993.

[22] R. W. Freund and N. M. Nachtigal. QMR: a Quasi-Minimal Residual Method for Non-Hermitian Linear Systems. *Numerische Mathematik*, 60(1):315–339, 1991.

[23] E. Georganas, A. Buluç, J. Chapman, S. Hofmeyr, C. Aluru, R. Egan, L. Oliker, D. Rokhsar, and K. Yelick. HipMer: An Extreme-scale De Novo Genome Assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 14:1–14:11. IEEE, 2015.

[24] O. Green, R. McColl, and D. A. Bader. GPU Merge Path - A GPU Merging Algorithm. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 331–340. ACM, 2012.

[25] F. Gustavson, L. Karlsson, and B. Kågström. Parallel and Cache-Efficient In-Place Matrix Storage Format Conversion. *ACM Trans. Math. Softw.*, 38(3):17:1–17:32, 2012.

[26] F. G. Gustavson. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[27] S. Hong, N. C. Rodia, and K. Olukotun. On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 92:1–92:11. IEEE, 2013.

[28] K. Hou, H. Wang, and W.-c. Feng. ASPaS: A Framework for Automatic SIMDization of Parallel Sorting on x86-based Many-core Processors. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, pages 383–392. ACM, 2015.

[29] K. Hou, H. Wang, and W.-c. Feng. AAlign: A SIMD Framework for Pairwise Sequence Alignment on x86-based Multi- and Many-core Processors. In *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*, IPDPS '16. IEEE, 2016.

[30] H. Inoue and K. Taura. SIMD- and Cache-Friendly Algorithm for Sorting an Array of Structures. *Proceedings of the VLDB Endowment*, 8(11):1274–1285, 2015.

[31] H. Kabir, J. D. Booth, G. Aupy, A. Benoit, Y. Robert, and P. Raghavan. STS-k: A Multilevel Sparse Triangular Solution Scheme for NUMA Multicores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 55:1–55:11. IEEE, 2015.

[32] J. J. Leonard, H. F. Durrant-Whyte, and I. J. Cox. Dynamic Map Building for an Autonomous Mobile

Robot. *Int. J. Rob. Res.*, 11(4):286–298, 1992.

[33] W. Liu and B. Vinter. A Framework for General Sparse Matrix-Matrix Multiplication on GPUs and Heterogeneous Processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015.

[34] W. Liu and B. Vinter. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, pages 339–350. ACM, 2015.

[35] W. Liu and B. Vinter. Speculative Segmented Sum for Sparse Matrix-Vector Multiplication on Heterogeneous Processors. *Parallel Computing*, 49:179–193, 2015.

[36] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel. Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets. In *Proceedings of the 25th ACM International Conference on Supercomputing*, ICS '11, pages 265–274. ACM, 2011.

[37] W. McLendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger. Finding Strongly Connected Components in Distributed Graphs. *J. Parallel Distrib. Comput.*, 65(8):901–910, 2005.

[38] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. G. Vadlamudi, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey. Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms. In *High Performance Computing*, volume 9137, pages 48–57. Springer, 2015.

[39] S. Ramos and T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. In *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '13, pages 97–108. ACM, 2013.

[40] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003.

[41] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 351–362. ACM, 2010.

[42] N. Sedaghati, T. Mu, L.-N. Pouchet, S. Parthasarathy, and P. Sadayappan. Automatic Selection of Sparse Matrix Representation on GPUs. In *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15. ACM, 2015.

[43] 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 *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 207–218. ACM, 2014.

[44] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[45] S. Yan, G. Long, and Y. Zhang. StreamScan: Fast Scan Algorithms for GPUs without Global Barrier Synchronization. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 229–238. ACM, 2013.

[46] X. Yu, H. Wang, W.-c. Feng, H. Gong, and G. Cao. cuART: Fine-Grained Algebraic Reconstruction Technique for Computed Tomography Images on GPUs. In *Proceedings of the 2016 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGrid '16. IEEE, 2016.