Accelerating GNNs on GPU Sparse Tensor Cores through N:M Sparsity-Oriented Graph Reordering

Jou-An Chen[1], Hsin-Hsuan Sung[1], Ruifeng Zhang[1], Ang Li[2], Xipeng Shen[1]
[1]North Carolina State University
[2]Pacific Northwest National Laboratory and University of Washington
{jchen73,hsung2,xshen5,rzhang38}@ncsu.edu,ang.li@pnnl.gov

Abstract

Recent GPUs have introduced Sparse Tensor Cores (SPTC) to accelerate computations on sparse matrices meeting the N:M sparse patterns. Software tools expand the support to more general V:N:M patterns. Graphs in Graph Neural Networks (GNNs) are typically sparse, but the sparsity is often irregular, not conforming to the required V:N:M sparse patterns. This paper proposes a novel graph reordering algorithm to transform irregular graph data into the required sparse patterns for GNNs to benefit from SPTC. The optimization is lossless, maintaining the accuracy of GNN. It at the same time keeps the symmetry of the adjacency matrices of the graphs so that the same matrices can remain compatible with many symmetry-based graph algorithms. The optimization successfully removes 98-100% violations of the N:M sparse patterns at the vector level and increases the portion of conforming graphs in the SuiteSparse collection from 5-9% to 88.7-93.5%. On A100 GPUs, the optimization accelerates Sparse Matrix Matrix (SpMM) by up to 43X (a geomean speedup of 2.3X – 7.5X) over cuSPARSE and speeds up the key graph operations in GNNs on real graphs by as much as 8.6X (3.5X on average).

CCS Concepts: **• Computing methodologies** → *Massively parallel algorithms*; **Neural networks**; **• Computer systems organization** → Single instruction, multiple data.

Keywords: GNN, Sparsity, GPU, Graph Reorder

ACM Reference Format:
Jou-An Chen[1], Hsin-Hsuan Sung[1], Ruifeng Zhang[1], Ang Li[2], Xipeng Shen[1]. 2025. Accelerating GNNs on GPU Sparse Tensor Cores through N:M Sparsity-Oriented Graph Reordering. In *The 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '25), March 1–5, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3710848.3710881

1 Introduction

Graph Neural Networks (GNNs) [18, 24, 34, 52] have emerged as a vital and versatile tool in addressing a wide array of graph-related problems. Their capacity to model and understand complex relationships within graphs has led to their increasing importance in numerous fields, including social networks [21], recommendation systems [37], molecular chemistry [23, 26], and more. However, the utility of GNNs comes hand in hand with the pressing concern of computational efficiency, particularly as graph sizes grow and the computation becomes more resource-intensive. The pursuit of accelerating GNN speed continues to be a critical endeavor.

PPoPP '25, March 1–5, 2025, Las Vegas, NV, USA
2025. ACM ISBN 979-8-4007-1443-6/25/03
https://doi.org/10.1145/3710848.3710881

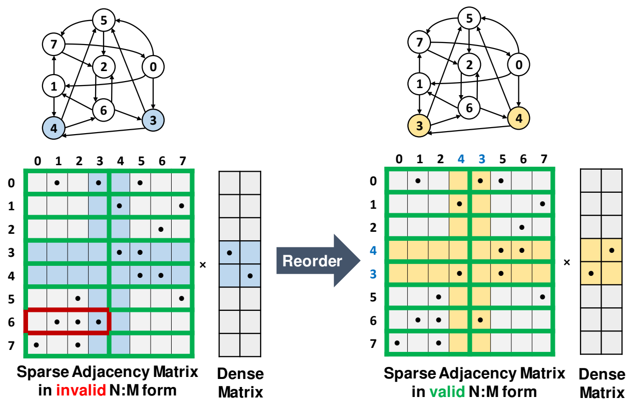

Sparse Adjacency Matrix in invalid N:M form
Dense Matrix
Reorder
Sparse Adjacency Matrix in valid N:M form
Dense Matrix

Figure 1. Graph vertex reordering leads to swapping of some rows (and columns) in the adjacency matrix, transforming row 6 into N:M-conforming (2:4 in this illustration) vectors. (Note that the corresponding rows in the dense matrix to multiply with must be swapped too.)

This paper explores a new approach to accelerating GNNs, reordering graphs to make GNNs better take advantage of the Sparse Tensor Cores (SPTC) on modern GPUs.

SPTC is a hardware feature prevalence in recent GPUs [4, 16, 46]. They are in all recent NVIDIA GPUs (Ampere and later), and similar hardware is getting into AI/ML hardware accelerators of other vendors (e.g., AMD/Xilinx ACAPs) [53, 56]. It provides efficient support for sparse fused multiplication-accumulation (FMA) instructions, or the `mma.sp` instruction, the key operation in sparse-matrix-multiply-dense-matrix (SPMM). The support is tailored explicitly for N:M sparsity. N:M sparsity refers to a pattern in a matrix where every M consecutive elements (called *segment vectors*) contain at most N non-zeros. Through hardware support, SPTC dynamically packs the non-zeros together to enable efficient FMA. Recent work [11] shows that a generalized pattern V:N:M (explained in Section 2) can be supported if one combines the hardware support with a software abstraction, producing even more speedups for SPMM over executions on dense tensor cores.

Despite the enormous potential of SPTC, most GNNs cannot take full advantage of it. It is not because GNNs have no sparse matrices: The key data structure in GNNs, the *adjacency matrix* of the input graph, is usually sparse. The reason is that most sparse matrices in GNNs do not conform to the required fine-grained sparsity patterns (e.g., 2:4 by default). Among 1356 graphs surveyed in the SuiteSparse collection [17], for instance, only 5-9% conform to the sparsity patterns (details in Section 5).

This paper proposes a novel solution, named *N:M sparsity-oriented graph reordering*. The basic idea is to transform as many non-N:M segment vectors within a matrix into N:M conforming vectors by graph reordering, that is, by swapping some rows (as well as responding columns) in the adjacency matrix of a graph. Unlike typical matrix reordering [60], the swapping in our method does not change the graph semantics but only the order of graph vertices. They are materialized by renumbering the graph vertices based on the fact that rows and columns in an adjacency matrix represent the vertices of the graph. As illustrated in Figure 1, if node 4 is renumbered to 3 and 3 to 4, the corresponding adjacency matrix would have rows 3 and 4 swapped and columns 3 and 4 swapped. The graph stays the same except for the numbering of vertices. The adjacency matrix of the graph can hence stay symmetric, keeping the matrix usable by the many graph algorithms that rely on its symmetry, such as algorithms designed for graph isomorphism, partitioning, minimum spanning tree, and so on. The constraint meanwhile makes the reordering algorithm more challenging to design than typical matrix reordering.

Prior work has used graph reordering to reduce non-zero blocks in format storage [6–8, 10, 15, 19, 33, 51, 56, 64]. But to the best of our knowledge, no prior work has studied graph reordering for N:M sparsity.

The critical challenge is how to find effective reordering for a given graph. The solution must work for various N:M patterns, including the generalized V:N:M variants. It does not have to be real-time, for a graph is often used many times (node features may change), and the one-time reordering is an offline preprocessing step. But it must still be efficient, as graphs can be large. The problem of finding the optimal reordering is NP-hard [7, 8, 15, 19], rendering enumeration impractical due to the vast combinatorial space of possible permutations. To the best of our knowledge, no prior work has undertaken this challenge. The only prior attempt is Jigsaw [60], but it applies only to the basic N:M patterns, and as a typical matrix reordering rather than graph reordering, it deprives the symmetry from the adjacency matrix.

Our solution consists of an iterative two-level graph reordering algorithm and an efficient bit intrinsic-based implementation. The algorithm, coined *dual-level N:M-sparsity oriented reordering*, consists of two reordering stages at its core. They, respectively, address the violations of tile-level patterns (a *tile* is a group of segment vectors) and vector-level patterns. The former converts a segment vector into a binary string and employs Hamming-distance position encoding [25, 55] to reorder columns and rows, while the latter uses a greedy reordering strategy to efficiently make each segment vector conform to the vector-level patterns [5, 38]. The two steps go hand-in-hand, forming the core of an iterative process. The proposed algorithm is general, working for various graphs, N:M and V:N:M patterns and GNNs. It efficiently and effectively fixes the sparsity pattern violations with a $n\log(n)$ computational complexity.

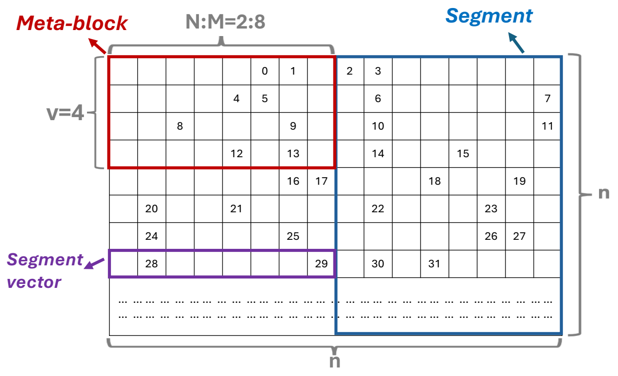
To deliver the full benefits of the algorithm, we develop the algorithm as an efficient library on GPUs, named *SOGRE* (Sparsity-Oriented Graph Reordering). SOGRE uses bit strings to represent the vectors in adjacency matrices and employs efficient bit-operations and intra-warp intrinsics on GPUs to materialize the core routines in the algorithm. The library can be integrated into existing GNN programming frameworks (e.g., PYG [22], DGL [48]).

As a lossless optimization, the reordering does not cause any accuracy loss. Experiments on 1356 adjacency matrices in SuiteSparse Matrix Collection show that the reordering algorithm can eliminate 98-100% violations of the N:M sparse patterns at the vector level and increase the proportion of conforming graphs in SuiteSparse collection from 5-9% to 88.7-93.5%. On A100 GPUs, the optimization accelerates Sparse Matrix Matrix (SpMM) by up to 43X (2.3X–7.5X on average) and speeds up the critical graph operations in GNNs on 12 real graphs by as much as 8.6X (3.5X on average) over cuSPARSE. The reordering takes 0.05–30s to run on average, offering an effective method for offline preprocessing of graphs that will be reused repeatedly across many inferences.

2 Terms and Background

This section provides the background needed for understanding the rest of the paper. (For the broader background on SPTC hardware and VENOM, please see references [4, 11, 16, 38].)

Terms and sparse patterns Figure 2 lists the frequently used terms in this paper, along with an illustration. The terms 'segment', 'meta-block', and 'segment-vector' refer to three levels of components in an adjacency matrix; see the illustration in Figure 2. The two notations, N:M and V:N:M, refer to two levels of sparse patterns. N:M is about the patterns in an M-element vector, where there are up to N non-zeros. V:N:M is about the patterns in a V-by-M tile, where every row is an N:M vector and, at the same time, at most k of its columns contain non-zeros. The value of k is determined by the SPTC hardware, 4 by default. N:M represents the patterns natively supported by SPTC hardware and V:N:M patterns are generalized forms introduced in the VENOM



Term	Description
n	Number of vertices in a graph
N:M	A sparse pattern with up to N non-zeros in M consecutive elements
V:N:M	A sparse pattern of a V-by-M tile, in which, (i) at most k columns contain non-zeros, and (ii) each row is a N:M vector. [k is a constant determined by the SPTC hardware; 4 by default]
Segment	An n-by-M submatrix in an adjacency matrix of a graph
Meta-block	A V-by-M tile in an adjacency matrix
Segment vector	A M-element row vector in an adjacency matrix

Figure 2. Terms frequently used in this paper.

work [11]. The VENOM authors show that such patterns can benefit from SPTC hardware when combined with a software abstraction.

GNN Background GNN [18, 24, 34, 52] captures contextual information of vertices and propagates it through graphs. It is a neural network performed on graph data. In a graph used in GNN, nodes represent entities in a problem domain (e.g., a user in a social network), each carrying a feature vector. Edges between nodes indicate their relationship, quantified with edge weights.

$$h_v^{l+1} = \mathrm{ReLU}((\mathrm{SUM}_{u \to v}(e_{uv} \odot h_u^l)) \otimes W^l) \qquad (1)$$

A computing layer in GNN consists of both graph operations and neural operations. A simple example is shown in Equation 1, which computes the hidden features of center node v on layer $l+1$. The input for layer l is the hidden features h^l; $u \to v$ means that there is an edge from node u to node v; e_{uv} is the value on the edge (zero means no edge between two nodes).

There are typically two phases in a layer's computation. In the *aggregation* phase, each vertex gathers information from its neighboring vertices within the graph. This phase often involves weighted summation or other aggregation functions. The implementation of this phase is usually based on matrix-matrix multiplication, which multiplies the graph's adjacency matrix with the nodes' feature matrix or hidden weight matrices. The second phase is the *update* phase, which performs fully-connected layer operations with activation functions, such as the ReLU function in Equation 1, to transform the aggregated info at each node and produce the hidden features of the nodes for the next layer h^{l+1} to process. The aggregation phase is what SpMM acceleration focuses on.

3 Problem Statement

For clarity, this section gives a formal definition of the reordering problem tackled in this work.

SPTC-oriented graph reordering problem (SGRP): Let A represent an $n \times n$ adjacency matrix of a graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a set of n vertices with ordering $\phi : (1, 2, ..., n)$, and V:N:M (which is N:M when V=1) be the sparsity structure constraints of the SPTC of interest; the SGRP problem is to find a permutation of the vertices in $\mathcal{V}$ that transforms ϕ into ϕ' such that the new form of the adjacency matrix meets the N:M (or V:N:M) constraints of the SPTC as much as possible.

It is worth noting that tiling is often used in SpMM, where the matrix is regarded as a collection of tiles; each tile is multiplied by the relevant sections in the other matrix, and the results are then put together to get the final result. So for a given tile size, the goal of SGRP becomes to minimize the number of tiles violating the V:N:M constraints.

We introduce a metric, **improvement rate**, to measure the effectiveness of reordering. It is calculated as $\frac{\text{final } \omega - \text{initial } \omega}{\text{initial } \omega}$, where ω represents the count of segment vectors that violate the required sparse pattern.

Problem Complexity. At the first glance, the search space for finding the optimal permutation ϕ is $n!$. However, there are redundancies in these permutations. For instance, a tile can remain a qualified N:M pattern when the ordering variance is within the tile boundary, reducing the total search space to $\frac{n!}{(M!)^{\frac{n}{M}}}$. But even with that, it is still too large to find optimal solutions in a polynomial time.

4 Graph Reordering Algorithm

4.1 Overview

The design of the reordering algorithm is based on the following principles:

(1) The algorithm should be *general*, able to work for various N:M and V:N:M patterns, graphs, and GNNs.

(2) The algorithm should be *beneficial*, giving significant improvement rates for various adjacency matrices and producing substantial speedups for GNNs.

(3) The algorithm should be *efficient*. The intended use of the algorithm is offline use, preprocessing a graph to prepare it for its repeated uses. Although the preprocessing time is not critical, the algorithm with a lower computational complexity could make it more friendly to adopt.

Based on these principles, we design a two-level N:M sparsity-oriented algorithm, outlined in Algorithm 1. This algorithm provides a unified treatment to N:M and V:N:M patterns by considering N:M as a special case of V:N:M when V=1. Its core consists of two stages of reordering, respectively, corresponding to the two constraints of V:N:M patterns. Recall from Section 2, V:N:M has the following two constraints:

(i) In each V-by-M meta-block, at most k columns have non-zero values. We call it **vertical constraint**.

(ii) In each row vector in each meta-block, there are at most N nonzero values. We call it **horizontal constraint**.

The first stage tries to reorder to minimize the violations of the *vertical constraint*, and the second stage for the *horizontal constraint*. Because the two stages influence each other's results, the algorithm repeats the two stages until no further progress can be made or the maximal iterations are reached, as outlined in Algorithm 1.

Both stages are designed to be efficient and effective, with computational complexities linear to n or $n\log(n)$ (recall n is the number of vertices in the graph). The algorithm makes no special assumptions on the sparse patterns, graphs, or matrices, making it generally applicable. We next explain each of the two stages.

Algorithm 1: The top-level pseudo-code of the proposed reordering algorithm

```
Input: Graph adjacency matrix $A$ with vertex order $\phi$,
       V:N:M pattern
Output: New vertex order
1 while V:N:M violations remain and max. iteration is not yet reached do
2     $\phi$ = Stage-1.reorder(A, V, N, M, $\phi$)
3     $\phi$ = Stage-2.reorder(A, N, M, $\phi$)
```

4.2 Stage-1 Reordering

Our design of stage-1 reordering hinges on two insights. The first is that it is likely to reduce violations of the vertical constraint if a reordering makes the rows in a meta-block more similar in terms of their nonzero positions. The second insight is that *Hamming-distance order* [55] of binary strings shares a similar objective as the first insight states, and hence using it could help solve our reordering problem. The first insight is easy to understand; we explain *Hamming-distance order* and the second insight next.

Hamming-distance Order *Hamming distance* is the count of differing digits between two binary strings. For instance, the hamming distance between 0011 and 0111 is one because they differ at only one position, the second digit from the left. The *cumulative Hamming distance* of a sequence of binary strings is the sum of the Hamming distances between every two adjacent strings. For example, the cumulative Hamming distance of sequence {00, 01, 10, 11} is 1+2+1=4 because the Hamming distance of the first pair of strings is 1, the second pair is 2, and the third pair is 1.

The *Hamming-distance order* of a sequence of binary strings is the order that has the smallest cumulative Hamming distance. For instance, the Hamming-distance order of all 2-digit binary strings is {00,01,11,10}, whose cumulative Hamming distance is 3. Previous work [55] has proved that there is a unique Hamming-distance order for all k-digit binary strings ($k \in N^+$).

Hamming Position Encoding A key idea in our stage-I algorithm is to use *Hamming position code* to encode every segment vector in a sparse matrix and then sort them numerically to help identify a good order for reducing the violations of the vertical constraint. We explain it as follows.

The **Hamming position code** of a k-digit binary string is defined as its rank in the *Hamming-distance order* of all k-digit binary strings. For instance, the Hamming position code of a 2-digit binary string, 11, is 2 because it is entry 2 (0-based) in the Hamming-distance order of 2-digit binary strings {00, 01, 11, 10}.

In our encoding, we give special treatment to specific segment vectors. If a vector violates the *horizontal constraint*, its position code is negated. For instance, the original adjacency matrix's bottom row in Figure 3 contains three nonzeros in its first segment vector, breaching the 2:8 constraint. Thus, the encoding result of that segment vector is -25. This technique aids subsequent sorting steps in preventing these vectors from contaminating other well-formed meta-blocks.

Stage-I Algorithm (Alg. 2) Alg. 2 outlines the Stage-I algorithm. The core of the algorithm contains five steps as

Algorithm 2: Stage-1 Algorithm for Increasing Vertical Conformity

```
Input: Graph adjacency matrix $A$ with vertex order $\phi$,
       V:N:M pattern
Output: New vertex order $\phi'$
1  $iter \leftarrow 0$
2  $MBscore \leftarrow$ GetMbScore$(A, V, N, M)$
3  while $MBscore > 0$ and $iter \leq MAXITER$ do
4      Initialize 2D matrix $Vec$ to store the segment vector
         encodings per row (per vertex)
5      for $i \leftarrow 0$ to $n - 1$ do
6          for $s \leftarrow 0$ to $\lceil \frac{n}{M} \rceil - 1$ do
7              $bstr \leftarrow$ binary string representation of
                 $A[i][s]$
8              $Vec[i][s] \leftarrow$ Hamming position code of
                 $bstr$
9              if $bstr$ is invalid $N : M$ pattern then
10                 $Vec[i][s] \leftarrow -Vec[i][s]$
11     Sort rows in $Vec$
12     Get the sorted order of $Vec$ as $\phi'$
13     Reorder $A$ with $\phi'$; $\phi \leftarrow \phi'$ $iter \leftarrow iter + 1$
14     $MBscore \leftarrow$ GetMbScore$(A, V, N, M)$
```

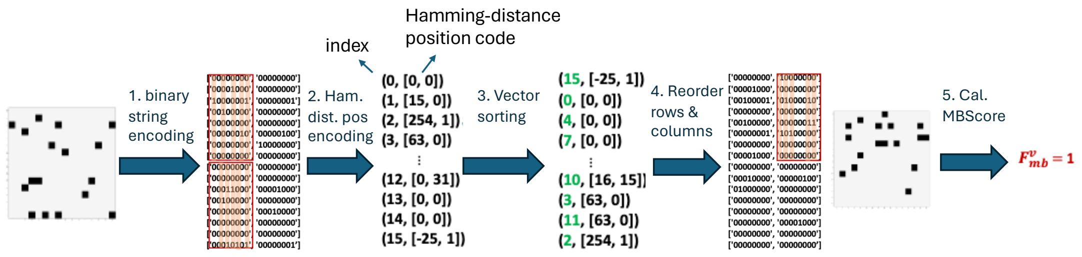


Figure 3. One iteration of the Stage-I algorithm (Alg. 2) for reducing vertical constraint violations. The example targets a V:N:M format as 8:2:8. It reduces the number of meta-blocks violating the vertical constraints (orange-covered blocks) from 2 to 1.

follows. Figure 3 uses an example to illustrate each of the steps.

(i) Binary string encoding: It represents every segment vector in an adjacency matrix as a binary string.

(ii) Hamming-distance position encoding: It encodes each binary string with its Hamming-distance position code, which transforms every row in the adjacency matrix into an integer vector.

(iii) Vector sorting: It sorts these integer vectors numerically to get a new order. The sorting result will have rows with similar Hamming position codes sitting close to each other. Because similar Hamming position codes entail few differences in the positions of nonzeros in the rows, the new order is likely to increase the conformity of the meta blocks regarding the V:N:M constraints.

(iv) Reordering: It swaps the adjacency matrix's rows (and columns) as per the new order.

(v) Assessment: It counts the number of non-conforming meta-blocks (called *MBScore*, denoted as $F_{MB}(\phi)$), and repeats steps (i-v) as needed.

Complexity Analysis. The theoretical computational complexity of the algorithm is $O(n\log(n))$ (in terms of the number of row sorting), given that the maximal number of iterations of the outermost loop is a constant. In practice, the sorting is much faster because many segment vectors are zero vectors and are left out of the sorting operation.

4.3 Stage-2 Reordering

After the Stage-1 algorithm reduces the vertical constraint violations at the Meta-block level, Stage-2 tries to reorder the graph to reduce the horizontal constraint violations at the segment vector level, that is, reduce the number of segment vectors in the adjacency matrix that violates the N:M pattern.

The challenge comes from the vast space of possible orders of the vertices in a graph. Finding the optimal is NP-hard. The question is how to find a good order without taking too much time. Our design takes the following strategy:

- Focus on a pair of segments each time. Recall from Section 2 that a segment is an n-by-M submatrix within an adjacency matrix. It consists of n segment vectors and its M columns correspond to M graph vertices. So, if we focus on two segments, each is about M vertices. We have only $2M$ vertices to consider, a much smaller scope.
- For a pair of segments, we try to identify the best pair between their vertices. Here, the best pair is a pair of vertices that, if we swap them, we get the largest reduction of the PScore of the adjacency matrix, where, **PScore** is defined as the number of horizontal constraint violations (P for pattern). As each segment has M vertices, there are only M^2 pairs to consider, which can be enumerated quickly to identify the best pair.
- After swapping the vertices in the best pair, we continue to explore other swapping opportunities.

Here, the order we follow when working with the segments in an adjacency matrix matters. Our strategy is to pick the worst segment (i.e., having the largest PScore) and keep working with it until all its columns have been used for swapping or its PScore cannot be lowered further, then move to the next worst segment, and so on. We call it our *primary segment*. To form a pair for the aforementioned vertex swapping, we need another segment, which we call *target segment*. We start with the worst of the remaining segments as our target segment and then move on to other segments to find more beneficial swapping opportunities with the primary segment.

Algorithm 3 outlines the algorithm. There are several details worth noting. (i) First, we exclude healthy segments (i.e., having zero PScore) from consideration, which can prevent polluting those segments and also help with the algorithm efficiency as it reduces the problem space. (ii) Second, when there is only one unhealthy segment left to examine, we choose the sparsest segment to form a swapping pair with it (line 6 in Algorithm 3). As that segment has the least nonzero elements, using it can maximize the chance of fixing the unhealthy segment while keeping itself healthy. That is the only time when a healthy segment may be used. (iii) Third, after a segment finishes serving as a primary segment, it will no longer be considered in any swapping, reflected by line 9 in Algorithm 3. This helps avoid polluting it after its treatment, and ensures the progress of the algorithm. (iv)

Fourth, instead of swapping them immediately after identifying a swapping pair, our algorithm records the swapping pair without swapping. It waits until all the swap pairs are recorded and then makes the swaps together (lines 21-23 in Algorithm 3). It helps with the execution efficiency. (v) The *freshtop()* function in line 13 chooses a swap pair that gives the largest gains (i.e., the largest reduction of the pscore of the primary and target segment), and none of its elements has been added into the swap records (S). It ensures the progress of the algorithm: The inner-most loop must exit after M iterations.[1]

Complexity Analysis. Let ω be the total number of segments that contain unhealthy segment vectors. For each primary segment, the number of iterations of the inner-most loop (line 10 in Algorithm 3) is at most M as every iteration adds one pair into S and there are at most M pairs needed to be added as the length of a segment is M. Both M and $MAXITER$ are constants. Therefore, the complexity of the algorithm is $O(\omega)$, no larger than $O(n)$.

4.4 Application to Large Graphs

The algorithm is intended for offline use, preprocessing a graph to prepare it for its repeated uses. But still the low computational complexity of the proposed algorithm will make it more friendly to adopt. It is worth noting that in practice, sampling is often used for GNNs on large graphs, where a small subgraph is sampled each time from the large graph for processing. The size of the subgraph is usually based on the limit of the underlying libraries. Current sparse libraries that use SPTC, such as NVIDIA official `cuSparseLT` [38] and `SPATHA (VENOM)` [11], all have a limit at the level of about 45K-by-45K matrices. Empirically (Section 5), the reordering algorithm completes within half a minute for sampled graphs of that size. In addition, the algorithm is compatible with parallel or distributed GNNs where each node conducts the multiplications involving a portion of the adjacency matrix; the reordering algorithm can be independently applied to each portion of the matrix; the results are reordered back before accumulation with the results from other nodes.

Listing 1. Bit-string encoding

```
1 int id = binarySearchInd(bsrcolind, idy, bsrrowptr[
     idx/M], bsrrowptr[idx/M+1]);
2 register unsigned val = 0x00000000;
3 if (id != -1)
4     for (int i=0; i<M; i++)
5         val = (val << 1)|(bsrval[id*M*M+(
            laneid%M)*M+i]);
```

[1]Notice that that function doesn't require the gain to be positive: We tried a design that enforces that condition but found it no more effective in practice but make the algorithm run much slower.

Algorithm 3: Stage-2 Algorithm for Increasing N:M Conformity

```
Input: Graph adjacency matrix $A$ with vertex order $\phi$, N:M pattern
Output: New vertex order $\phi'$
1  $I \leftarrow$ GetPScoreList$(A, N, M)$ // A priority list
2  Exclude valid segments from $I$
3  while $|I| \geq 1$ and $iter \leq MAXITER$ do
4      $S \leftarrow \emptyset$
5      if $|I| = 1$ then
6          Make beneficial swaps with the sparsest segment
7      else
8          while $|I| > 1$ do
9              $(segID_{prim}, pscore_{prim}) \leftarrow I.pop()$
10             for $(segID_{targ}, pscore_{targ})$=I.next() do
11                 $swap\_cands \leftarrow$ GetCandidates$(segID_{prim}, segID_{targ})$
12                 Vertex pair $(u, v) \leftarrow swap\_cands.freshtop()$
13                 $S \leftarrow S \cup \{(u, v)\}$
14                 $pscore_{prim} \leftarrow pscore_{prim} - u.gain$
15                 $pscore_{targ} \leftarrow pscore_{targ} - v.gain$
16                 if $pscore_{targ} == 0$ then
17                     remove $segID_{targ}$ from $I$
18                 if $pscore_{prim} == 0$ or all vertics of $segID_{prim}$ have been added into $S$ then
19                     break
20     Initialize $\phi'$ with $\phi$
21     for $(u, v) \in S$ do
22         Swap $u$ and $v$ in $\phi'$
23     Reorder $A$ with $\phi'$; $\phi \leftarrow \phi'$
24     $iter \leftarrow iter + 1$
25     $I \leftarrow$ GetPScoreList$(A, N, M)$ // update I
26     Exclude valid segments from $I$
```

4.5 Library and Integration

We implement the graph reordering algorithm as a library for easy adoption. The implementation is in CUDA, so it can benefit from GPUs. The implementation carefully takes advantage of the low-level intrinsics (e.g., intra-warp shuffling, voting) for high efficiency. As values in an adjacency matrix are binary, we use bit intrinsics in CUDA for frequently invoked subroutines.

Listing 1 shows an example. This routine converts an adjacency matrix into a binary string. The adjacency matrix is stored in a Block Sparse Row (BSR) format [2]. In that format, the matrix is viewed as a collection of M-by-M blocks (called *bsr block*). It builds indexing structures (`bsrcolind` and `bsrrowptr`) (like those in CSR format) to help index the positions of the bsr blocks in the adjacency matrix. By doing that, it only needs to store (in the `bsrval` array) the values of the bsr blocks that contain any nonzeros. In Lst 1, line 1 lets each GPU thread locate, based on (idx, idy), the segment vector that it needs to do the binary string encoding. Line 5 uses bit shifting to complete the binary string encoding for one number in the segment vector. Its enclosing loop at line 4 makes the thread accomplish the encoding of the whole segment vector.

More subroutines are in the supplementary material. (For GPU intrinsics, see documentation [1].)

Integration with Existing GNN Frameworks Our optimization is complementary to other GNN optimizations and can be used together. Our optimization results in matrix formats that can benefit from the efficient SpMM kernels on SPTC. Those kernels can serve as a drop-in replacement of the SpMM kernels in existing frameworks and the existing optimizations in the frameworks can take effect as usual. The SPTC SpMM kernels we use is based on Spatha [11]; the used PTX mma instruction is mma.sp.sync with the default m16n8k32, and the default index representation [3] required by SPTC is used. During SpMM execution, the kernels put the matrices into the SPTC required form on the fly and then uses the mma instructions to do the computations. The time spent on data movement is negligible (over three orders of magnitude less) compared to both the reordering and SpMM execution times. Section 5 reports the benefits of integrating the optimization with two existing GNN frameworks, PyG and DGL.

5 Evaluation

We evaluate the efficacy of the reordering algorithm. The reordering happens in an offline preprocessing stage, applying our reordering algorithms to the graphs first to make them conform to the constraints and then measure the GNN performance. It determines the best V:N:M by trying 1:2:M forms, with M initialized to 4 and progressively doubled until the graph can no longer be reordered to conform to the form. Then, it fixes M and determines the best V by trying V:2:M with V going from 1 to 32 in a similar manner (N must be 2 due to the hardware constraint). In our experiments, we set the maximum number of iterations for the vertical and fine-grained reordering loops to 10. For most matrices, these loops converge within six iterations or fewer. As the reordering is an offline process and the results can benefit all future use of the graphs, the reordering time is not counted in the GNN running time.

Table 1. SuiteSparse graph collection. (*Med*: *Median*)

		#V	#E	Avg Degree	Max Degree	Diameter	#Graphs
Small	*Avg*	426	4.97k	12.5	60.7	12.5	444
	Med	430	2.19k	7.6	15	6	
Medium	*Avg*	3.6k	93.2k	22.5	405.1	42.1	724
	Med	2.6k	24.1k	9.7	61	8	
Large	*Avg*	22.6k	878k	36.1	1041.6	75.9	188
	Med	20.5k	229k	13.8	98.5	10	

Table 2. GNN graph dataset.

Dataset	#V	#E	#Features	#Classes
Cora [54]	2,708	10,556	1,433	7
Citeseer [54]	3,327	9,104	3,703	6
Facebook [54]	4,039	88,234	1,283	193
Computers [44]	13,752	491,722	767	10
CS [44]	18,333	163,788	6,805	15
CoraFull [9]	19,793	126,842	8,710	70
Amazon-ratings [41]	24,492	93,050	300	5
Physics [44]	34,493	495,924	8,415	5
ogbn-proteins [27]	132,534	39,561,252	128	2
ogbn-products [27]	2,449,029	61,859,140	100	47
ogbn-arxiv [27]	169,343	1,166,243	128	40
ogbn-papers100M [27]	111,059,956	1,615,685,872	128	172

Datasets: In evaluating the performance benefits to GNNs, we use 12 graph datasets, widely adopted in the GNN field, for our evaluation. The characteristics of these datasets, including the number of vertices, edges, features, and classes for node classification, are listed in Table 2. In addition, for a more comprehensive evaluation of the performance benefits brought to SpMM by our reordering algorithm, we conducted a performance comparison of SpMM on the 1356 adjacency matrices of the real-world graphs included in the SuiteSparse Matrix Collection [17]. The adjacency matrices in the collection are organized into three classes, *small*, *medium*, and *large*, as shown in Table 1.

GNN Models: We use four commonly used GNN models: (1) *Graph Convolutional Neural Networks (GCN)* [34] (2) *GraphSAGE (SAGE)* [24] (3) *Chebyshev Spectral Graph Convolutional Neural Networks (Cheb)* [18] (4) *Simplifying Graph Convolutional Networks (SGC)* [52].

Platforms: The performance evaluation of executing reordered graphs was primarily carried out on a node with NVIDIA A100 GPUs with 40GB memory (CUDA v11.7) and 4th Generation AMD EPYC CPUs.

5.1 GNN Performance Comparison

In this subsection, we assess the performance of GNN (per-layer and end-to-end). To our best knowledge, even though many prior studies have explored graph reordering, none is for and hence applicable to the V:N:M configurations. As the first algorithm to that end, we evaluate it by assessing the speeds of GNNs and the SPMM kernels before and after the reordering. Our evaluation focuses on the forward pass of node classification.

We use two widely used GNN frameworks: PyTorch Geometric (PYG) [22] and Deep Graph Library (DGL) [48]. Both libraries are well optimized for GPUs and have supported a broad range of existing research works [18, 24, 34, 52]. It is worth noting that our reordering method should be seen as a modular utility orthogonal to all existing GNN frameworks, as reordering can be applied offline as a preprocessing step. Consequently, any GNN framework that relies on GPU-based SpMM can essentially benefit from our approach.

The *default* SpMM kernels in PYG and DGL do not take advantage of SPTCs on GPU. We create *revised* versions of

Table 3. Normalized speedup of four GNN models compared to the default PyG [22] and DGL [48] (default-original). "Best V:N:M" is the best format the graph can reach through the proposed reordering algorithm. "LYR" refers to average speedup on aggregation; "ALL" refers to end-to-end speedup.

Dataset	Best V:N:M	PYG								DGL							
		GCN		SAGE		Cheb		SGC		GCN		SAGE		Cheb		SGC	
		LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL
Cora	1:2:4	1.41	1.12	2.13	1.15	2.33	1.53	2.63	2.03	1.18	1.08	2.01	1.27	3.80	1.34	2.06	1.67
Citeseer	32:2:8	2.11	1.11	2.84	1.16	3.16	1.65	3.40	1.69	1.71	1.09	2.92	1.34	4.49	1.46	2.94	2.71
Facebook	1:2:4	3.26	1.64	6.01	2.23	8.56	4.22	8.64	6.37	3.27	1.63	6.01	2.42	4.41	2.98	6.39	4.74
Computers	1:2:4	2.72	2.99	3.83	2.85	5.31	4.02	5.30	5.15	3.30	2.65	4.12	2.91	2.90	3.06	4.13	4.05
CS	16:2:16	1.73	1.11	3.12	1.79	4.36	2.60	4.38	4.21	1.85	1.12	2.78	1.56	2.12	1.70	2.69	2.61
CoraFull	32:2:16	1.44	1.05	2.05	1.40	2.91	1.96	2.95	2.73	1.53	1.06	1.96	1.30	1.96	1.39	1.88	1.79
Amazon-ratings	1:2:32	2.01	1.99	3.42	1.98	4.24	2.20	4.30	3.85	1.30	1.43	1.99	1.38	2.00	1.30	1.80	1.70
Physics	16:2:16	1.56	1.14	3.00	2.28	4.46	3.21	4.76	4.68	1.80	1.14	2.23	1.61	2.20	1.70	2.20	2.18

the two frameworks by replacing their SpMM kernels with Spatha SpMM, a SpMM library that takes advantage of SPTCs. Spatha was developed in a prior VENOM work [11] for DNNs. It is the only available library that supports V:N:M patterns. It, however, cannot directly apply to GNNs because most graphs, including those in our experiments, do not conform to the pattern constraints of V:N:M—a problem addressed by our reordering algorithm. We compare the performance in four settings.

- **Default-original:** the default PYG and DGL frameworks (without SPTC support) on the original matrices.
- **Default-reordered:** the default PYG and DGL frameworks (without SPTC support) on our reordered matrices. Because our reordering is for leveraging SPTC while this setting does not use SPTC, we do not expect this setting to have performance improvement over the *default-original*.
- **Revised-pruned:** the revised PYG and DGL frameworks (with SPTC support) on pruned matrices. The pruning is based on magnitude of values. For each V:N:M meta-block, it zeros a minimum number of elements with the least magnitude such that the meta-block conforms to the SPTC required V:N:M sparse pattern. This method can turn matrices into SPTC-required forms and hence is expected to generate similar speedups over the *default-original* as our method does, but because its pruning introduces errors, it is expected to affect the prediction accuracy of the GNNs.
- **Revised-reordered (our solution):** the revised PYG and DGL frameworks (with SPTC support) on our reordered matrices. This is our solution. We expect that by making the matrices able to take advantage of SPTC, this method shall bring significant speedups over *default-original*. Reordering is a lossless transformation: It renumbers the vertices in a graph and involves no approximation at all. So it does not compromise the accuracy of GNN.

Compare to *default-original*: Table 3 reports the speedups of our solution over the *default-original* version of PyG and DGL. PYG uses the Torchsparse-based CSR-SpMM for SpMM. As shown in the left part of Table 3, the average layer-wise speedup of our approach over PYG is from 1.4 to 3.3X for the GCN model and 2.6-8.6X at maximum for SGC. These translate into an end-to-end speedup of 1.1-3X for GCN and 2-6.3X for SGC. GraphSAGE (SAGE in Table 3) and Cheb show speedups in the between. The differences in the GNN composition and hence the execution order of computations cause the differences. For instance, GCN aggregates after its linear layer, and GraphSAGE aggregates before two linear layers, which makes GraphSAGE exhibit more speedups over GCN as our optimizations are for linear layers. The performance difference between the Torchsparse-based CSR-SpMM and the SPTC-based SpMM becomes even more prominent when the multiplier matrix has more columns, which typically represent larger feature lengths, hidden embedding lengths, and numbers of classes. For example, SGC has more feature embedding columns than GCN does, and gains more substantial speedups through the reordering and V:N:M of SPTC.

In general, DGL is more performant than PYG (except in some cases where the vertices exceed 4,000 and H $\leq$ 512), partially because it uses CuSparse CSR SPMM kernel with the "CUSPARSE_SPMM_CSR_ALG2" algorithm for SpMM, which is overall faster than the one used in PYG. Nonetheless, there are clear speedups across the board when DGL adopts our reordering-based SpMM kernel. The trends are similar to those observed on PYG. **Compare to *default-reordered:*** Table 4 shows the speedups of *default-reordered* over *default-original*. There are no significant speedups, in both the aggregation layers and the end-to-end GNNs. The reason is that in this setting, the GNN kernels work similarly to those in *default-original*, CSR-based SpMM running on CUDA cores. The reordered matrices have the same sparsity as the original matrices have and the CUDA cores are oblivious to the V:N:M sparse patterns.

Compare to *revised-pruned:* Like our solution, the *revised-pruned* setting also yields matrices conforming to the required V:N:M sparse patterns. So it is no surprise that the revised PyG and DGL in the *revised-pruned* setting are able to generate the same degree of speedups as in our solution. The differences between their measured speedups over the *default-original* are marginal, ranging from ±0.01 to ±0.07,

Table 4. Speedups of *default-reordered* over *default-original*

Dataset	Best V:N:M	PYG								DGL							
		GCN		SAGE		Cheb		SGC		GCN		SAGE		Cheb		SGC	
		LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL	LYR	ALL
Cora	1:2:4	1.01	1.00	1.02	1.01	1.04	1.03	0.99	0.99	1.03	1.01	0.98	0.99	1.06	1.01	1.08	1.07
Citeseer	32:2:8	1.05	1.02	1.01	1.00	1.02	1.01	0.98	0.98	1.04	1.01	0.97	0.99	1.00	1.00	1.01	1.01
Facebook	1:2:4	1.02	1.02	1.02	1.01	1.02	1.01	1.02	1.02	0.94	0.97	0.99	1.01	0.98	0.99	0.98	0.99
Computers	1:2:4	0.94	0.94	0.99	1.00	1.01	1.01	1.00	1.00	0.98	0.99	0.99	1.00	1.00	1.00	1.00	1.00
CS	16:2:16	0.97	1.00	0.98	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00	1.00
CoraFull	32:2:16	1.00	1.00	0.98	0.98	0.95	0.96	0.99	0.99	0.99	1.00	0.98	1.00	0.98	1.00	0.98	0.98
Amazon-ratings	1:2:32	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.02	1.01	1.01	1.00	0.98	0.99
Physics	16:2:16	1.00	1.00	0.96	0.99	0.98	0.98	1.00	1.00	0.97	1.00	1.01	1.00	1.00	1.00	1.00	1.00

Table 5. Accuracy comparison between our solution (reorder) and revised-pruned. Reorder is lossless and causes no accuracy loss, but pruning is lossy. The numbers in brackets are the accuracy loss caused by pruning.

Dataset	Sparsity	Prune ratio	GCN acc		GraphSAGE acc		ChebNet acc		SGC acc	
			reorder	prune	reorder	prune	reorder	prune	reorder	prune
Cora	0.14%	1.48%	0.8130	0.7940 (-2.34%)	0.8040	0.7770 (-3.36%)	0.7910	0.7360 (-6.95%)	0.8010	0.7840 (-2.12%)
Citeseer	0.08%	0.88%	0.6860	0.6720 (-2.04%)	0.7030	0.6750 (-3.98%)	0.6900	0.6400 (-7.25%)	0.6660	0.6610 (-0.75%)
Facebook	0.54%	4.35%	0.6922	0.6452 (-6.79%)	0.6452	0.5933 (-8.04%)	0.5167	0.4784 (-7.41%)	0.6737	0.6403 (-4.96%)
Computers	0.26%	0.09%	0.8975	0.7899 (-11.9%)	0.8964	0.8177 (-8.78%)	0.6036	0.5227 (-13.4%)	0.8895	0.8644 (-2.82%)
CS	0.05%	0.004%	0.9414	0.9343 (-0.75%)	0.9504	0.9414 (-0.95%)	0.9515	0.9436 (-0.83%)	0.9419	0.9321 (-1.04%)
CoraFull	0.03%	9.14%	0.7076	0.6788 (-4.07%)	0.6909	0.6437 (-6.83%)	0.6437	0.5707 (-11.3%)	0.7121	0.6831 (-4.07%)
Amazon-ratings	0.02%	0.72%	0.4307	0.4187 (-2.79%)	0.4350	0.4215 (-3.10%)	0.4378	0.4225 (-3.49%)	0.4060	0.3813 (-6.08%)
Physics	0.04%	0.007%	0.9694	0.9580 (-1.18%)	0.9704	0.9635 (-0.71%)	0.9707	0.9657 (-0.52%)	0.9629	0.9572 (-0.59%)

within 1% of the speedups. It is the case even for large datasets. For instance, for Facebook graph, their average layer-wise speedups on PyG vary in the range of 8.49×–8.56×, and their end-to-end speedups vary in the range of 4.19×—4.22×. The key differences are in the prediction accuracy, which is what our report focuses on in Table 5. Because reordering is a lossless transformation, our solution preserves the prediction accuracy of GNNs. But pruning is lossy. Table 5 shows the accuracy comparison. Unlike weight pruning in DNNs, graph edges carry critical information, and their removal can result in significant accuracy degradation. Some graphs have a small pruning ratio due to their small numbers of pattern violations, allowing them to maintain accuracy with a drop of less than or around 1%. A small pruning ratio however does not always lead to a small accuracy drop. For instance, in the case of 0.09% pruned *Computers*, the accuracy loss can reach as high as 13.4%, surpassing that of 9.14% pruned *CoraFull* in most GNNs. Furthermore, the accuracy drop varies depending on the network's robustness. For example, pruned graphs generally result in higher accuracy drops for ChebNet compared to SGC, but for the least pruned graphs, the trend is reversed. Reordering is a more reliable solution than pruning for achieving V:N:M sparse patterns.

5.2 Speedups on Distributed GNN on Large Graphs

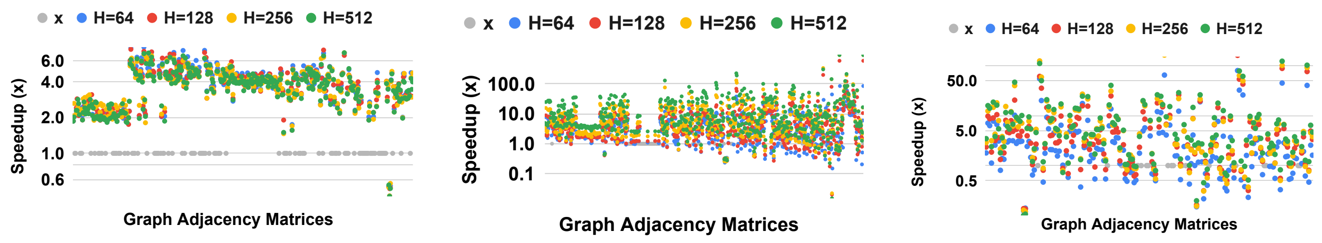
For large graphs that cannot fit into a single GPU, our experiments follow the typical practice: partitioning or sampling the large graphs into smaller subgraphs for processing. The OGBN dataset is often used to evaluate multi-GPU GNNs. For the completeness of the evaluation, we use each of the four datasets in OGBN in our experiment, even though some of them are small enough to fit into one of the A100 GPUs. Specifically, the four datasets in OGBN, *ogbn-protein*, *arxiv*, *products*, and *papers100M* are partitioned into multiple subgraphs using the PYG's `NeighborSampler`, with average vertex counts of 24604, 2514, 19833, and 7607 per sample, respectively. After the sampled subgraphs are reordered, they serve the SPTC-based GNNs in parallel on **four A100 GPUs**. Table 6 shows the per-layer and overall speedups of our approach over PYG implementation using the SGC model. Overall, our reordering with SPTC can deliver 1.16–3.23× speedups in end-to-end GNN performance. The speedups come not only from the performance benefits of SPTCs (which become usable after our reordering) over CUDA cores, but also from the software efficiency. The baseline performs SpMM on the CUDA cores using a sparse format (i.e., CSR), which causes it to suffer from irregular memory access; the baseline performance is hence far below the theoretical performance of CUDA cores. In contrast, SPTC-based computing uses a compact format, and enjoys efficient regular memory access and superior cache benefits, yielding a much higher throughput.

Table 6. OGBN [27] large graphs GNN evaluation. "LYR" refers to average speedup on aggregation; "ALL" refers to end-to-end speedup.

	ogbn-proteins	ogbn-arxiv	ogbn-products	ogbn-papers100M
LYR	1.140	6.494	1.423	2.781
ALL	1.159	3.229	1.399	2.449

5.3 SpMM Kernel Evaluation

As the key benefits come from the SpMM kernels, we give a comprehensive evaluation of the SpMM kernel performance using the 1356 adjacency matrices in SparseSuite.



(a) Small **(b)** Medium **(c)** Large

Figure 4. Speedup of SpMM over cuSPARSE brought by graph reordering matrices to their best V:N:M formats. The x-axis corresponds to individual graphs in SuiteSparse; "H" is #columns in the second matrix in SpMM.

Table 7. 1:2:4 Reordering algorithm quality on SuiteSparse. The "#inv segvec" is the number of invalid segment vectors in 1:2:4 (i.e., $F_p(\phi)$). "Iter." is the number of iterations needed for the process to conclude. This value depends on both how much invalid segment vectors are there initially and how effectively the exploration converges.The rightmost column lists the average times for the reordering. (*Med: Median*)

		Init. #inv segvec	Finl. #inv segvec	Imprv. rate	Iter.	Reorder time (s)
Small	*Avg*	510.31	0.96	99.29%	34.40	0.05
	Med	120.00	0.00	100.00%	20.00	0.01
Medium	*Avg*	12,656.56	21.43	99.94%	267.62	4.39
	Med	1,124.00	0.00	100.00%	208.50	0.63
Large	*Avg*	33,202.87	930.11	98.87%	670.70	30.55
	Med	8,423.00	0.00	100.00%	589.00	11.12

Speedups. Figure 4 illustrates the normalized speedups over cuSPARSE. We adjust the number of columns in the multiplier matrix from 64 to 512, reflecting common hidden embedding dimensions in GNNs. As can be seen, significant speedups are achieved generally across all three size categories, particularly within medium and large. This trend can be attributed to larger graphs having increasingly sparse adjacency matrices, thereby providing greater scope and potential for reordering-based SPTC executions.

Additionally, we noticed that a small subset of matrices (3.9%) experienced slowdowns following the reordering process. Further analysis revealed that these matrices are extremely sparse (mostly with density $< 0.01\%$), where the advantages of focusing solely on non-zero elements perhaps surpass the benefits of using SPTC. This is because SPTC still needs to process some zero elements if the matrices are very sparse for successful ordering and compaction. Nevertheless, since reordering is performed offline, users can decide whether their graph is unsuitable for reordering based on these considerations.

Effects on fixing pattern violations. Among the 1356 adjacency matrices, over 94% of them do not conform to any V:N:M patterns. As reported in Table 7, the reordering algorithm (for the 1:2:4 pattern) reduces the segment vectors with invalid patterns by 98.9–100% and increases the proportion of conforming matrices from 8.74% to 93.92% for small graphs, 7.32% to 87.85% for medium graphs, and 5.91% to 89.25% to large graphs.

Table 8. Reordering success rate on SuiteSparse.

	Small		Medium		Large	
	V:2:8	**V:2:16**	**V:2:8**	**V:2:16**	**V:2:8**	**V:2:16**
V=1	69.1%	49.6%	65.6%	49.1%	71.7%	61.3%
V=4	29.1%	8.2%	16.2%	10.2%	25.8%	14.6%
V=8	8.1%	3.1%	5.0%	6.2%	31.0%	13.5%
V=16	3.8%	0.4%	7.0%	4.6%	17.9%	7.3%
V=32	2.2%	2.2%	7.9%	2.2%	1.3%	1.2%

Reordering time. The rightmost column in Table 7 lists the average and median reordering times reordering takes for the 1356 adjacency matrices. The times range from 0.01s to 30.55s.

Distribution of preferred V:N:M patterns. Different matrices may prefer different V:N:M patterns. It depends on the matrix density and the distributions of non-zeros. Table 8 presents the distribution of preferred V:N:M formats across the matrices. The proportions of formats with larger V values tend to be smaller than those of other formats. This is because as V increases, the size of the meta-block imposes stricter restrictions on non-zero patterns. Consequently, for many matrices, although the reordering algorithm effectively optimizes numerous segment vectors, transforming the entire matrix to fit the required patterns becomes challenging. Therefore, despite patterns with larger V values often yield more remarkable speedups, they are not the most frequently selected choices. It is possible to create some machine learning models to predict the preferred V:N:M pattern for a given matrix, akin to the predictors of the best sparse storage format for a matrix [62, 63, 66], which is out of the scope of this work. Given that the reordering algorithm is completed in only a short time, a simple approach is to try a number of common patterns and select the best one.

6 Related Work

There have been many efforts on how to make irregular computations best benefit from the massive parallelism of GPUs. The line of research was pioneered by the on-the-fly optimizations introduced by Zhang and others [57, 58]. Since

Huang and others' systematic exploration of the implications of GNN performance gaps [29], many studies have been devoted to bridging the gap [12, 28, 49, 50, 59].

Several existing efforts address the challenge of mitigating sparse workloads on tensor cores. TC-GNN [50] and DTC-SpMM [20] tackle sparse workloads by employing specialized formats and designs to execute them on dense tensor cores. The use of dense formats significantly increase memory usage, adding tens to hundreds of times more space and memory pressure—a critical issue for large graphs.

Graph Reordering. Graph reordering has been commonly used as a preprocessing technique for graph computation to improve data locality and other purposes [7, 8, 15, 19, 30, 36, 40, 45]. This is because altering the vertex order, along with their corresponding edges in representations like the adjacency matrix, typically doesn't impact correctness but can potentially optimize data layout for more efficient computation. Existing fine-grained graph reordering proposals, such as MinLA [39] and MiLogA [14, 43], tailor vertex orders specifically for social network computation. Meanwhile, GScore in GOrder confines the reordering within a sliding window, enhancing graph processing efficiency on CPUs [51]. For scalability, coarser-grained reordering methods like degree-based sorting [19, 31, 35, 61] and partitioning [6, 10, 33, 56, 64] come into play. These methods aim to balance subgraph size for workload balancing or minimize cuts for enhanced communication efficiency during scaling. Despite the demonstrated effectiveness of graph reordering in optimizing data layout and subgraph workload distribution, no prior studies like our work has explored graph reordering for V:N:M sparse patterns to unlock the potential of SPTCs on GPUs.

N:M Sparsity. Many DNN pruning and fine-tuning techniques have emerged for fitting DNNs into 2:4 sparse patterns for SPTCs. Zhou et al. [65] present a training approach to construct an N:M fine-grained structured sparse network from scratch. They also present a metric called Sparse Architecture Divergence (SAD) to track and guide the topology change of the sparse DNN during training. Sun et al. [47] present DominoSearch for iterative N:M sparsity determination in DNN training using magnitude-based pruning. Pool et al. [42] suggest rearranging CNN channels prior to pruning for N:M compliance, which can reduce the possibility of pruning significant entries that are important to accuracy. Kao et al. [32] develop "structure decay" for iterative N:M pruning, using a mask decay method to improve training stability. Chen et al. [13] present Dynamic Feature Selective Sparsity (DFSS) in Transformers, dynamically pruning the network to the N:M pattern. All of these works, including popular libraries such as cuSPARSELt [38] and VENOM [11], however, focus on transforming dense DNNs into N:M structured sparse models through lossy DNN compression such as pruning. Our work centers on the lossless transformation of adjacency matrices to best fit the constraints of V:N:M sparse patterns.

A study concurrent to our work is Jigsaw [60]. Jigsaw directly reorders the columns of adjacency matrices into 2:4 sparse forms and uses their customized SpMM kernels to harness the power of SPTC. It differs from our work in several aspects. First, because our method is graph reordering, the adjacency matrix remains symmetric after reordering, while it is not the case for the column reordering in Jigsaw. The symmetry is essential to many graph algorithms, such as Graph Isomorphism, Graph Partitioning using spectral clustering, Kruskal's Algorithm for Minimum Spanning Tree, and so on. Second, our reordering algorithm runs more efficient and hence reorders more matrices in shorter times, thanks to its design and efficient implementation on GPU. For large graphs in SparseSuite, for instance, our method can reorder 52% of the graphs within 20s each while JigSaw can do only 30%. Finally, Jigsaw supports only the 2:4 format, whereas our algorithm accommodates the more flexible V:N:M formats. We note that Jigsaw introduces additional optimizations to the sparse kernel implementation to hide memory access latency. Those kernel-level optimizations could further enhance the performance of our solution.

7 Conclusion

This paper presents a novel graph reordering technique to harvest modern GPU sparse tensor cores for GNNs. By orchestrating bit-level operations, our proposed algorithm transforms graph adjacency matrices into V:N:M formats, employing an iterative two-stage reordering process based on Hamming-distance order and greedy algorithm designs. Experiments show that the method can accelerate the essential graph computation kernel SPMM by 2.3-7.5×, boosting the GNN computation significantly.

Acknowledgment

We thank the anonymous shepherd for guiding the preparation of the final version of this paper, and the anonymous reviewers for their helpful comments. This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CNS-2312207 and the Department of Energy (DOE) under Grant No. DE-EE0009357, as well as the work supported by DOE Office of Science, Office of Advanced Scientific Computing Research, ComPort: Rigorous Testing Methods to Safeguard Software Porting, under Award Number 78284. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or DOE. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830.

References

[1] [n. d.]. CUDA Math API Documentation: Integer Intrinsics. https://docs.nvidia.com/cuda/cuda-math-api/group__CUDA__MATH__INTRINSIC__INT.html Accessed: 2023-11-30.

[2] [n. d.]. Scipy Block Compressed Row Format (BSR). https://scipy-lectures.org/advanced/scipy_sparse/storage_schemes.html#block-compressed-row-format-bsr Accessed: 2023-11-30.

[3] [n. d.]. SPTC index. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-sparse-matrix-storage

[4] 2023. Nvidia ampere architecture in-depth. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/

[5] NVIDIA Corporation Accessed: 2023-09-08. *NVIDIA CUDA Documentation — Warp Level Matrix Multiply-Accumulate Instructions*. NVIDIA Corporation. https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#warp-level-matrix-multiply-accumulate-instructions

[6] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[7] Vignesh Balaji and Brandon Lucia. 2018. When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 203–214.

[8] Reet Barik, Marco Minutoli, Mahantesh Halappanavar, Nathan R Tallent, and Ananth Kalyanaraman. 2020. Vertex reordering for real-world graphs and applications: An empirical evaluation. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 240–251.

[9] Aleksandar Bojchevski and Stephan Günnemann. 2018. Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking. In *International Conference on Learning Representations*. https://openreview.net/forum?id=r1ZdKJ-0W

[10] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. *Recent advances in graph partitioning*. Springer.

[11] Roberto L. Castro, Andrei Ivanov, Diego Andrade, Tal Ben-Nun, Basilio B. Fraguela, and Torsten Hoefler. 2023. VENOM: A Vectorized N:M Format for Unleashing the Power of Sparse Tensor Cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 72, 14 pages. https://doi.org/10.1145/3581784.3607087

[12] Jou-An Chen, Hsin-Hsuan Sung, Xipeng Shen, Sutanay Choudhury, and Ang Li. 2023. BitGNN: Unleashing the Performance Potential of Binary Graph Neural Networks on GPUs. In *Proceedings of the 37th International Conference on Supercomputing* (Orlando, FL, USA) *(ICS '23)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/3577193.3593725

[13] Zhaodong Chen, Zheng Qu, Yuying Quan, Liu Liu, Yufei Ding, and Yuan Xie. 2023. Dynamic n: M fine-grained structured sparse attention mechanism. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 369–379.

[14] Flavio Chierichetti, Ravi Kumar, Silvio Lattanzi, Michael Mitzenmacher, Alessandro Panconesi, and Prabhakar Raghavan. 2009. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 219–228.

[15] Benjamin Coleman, Santiago Segarra, Alex Smola, and Anshumali Shrivastava. 2022. Graph Reordering for Cache-Efficient Near Neighbor Search. In *Advances in Neural Information Processing Systems*, Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (Eds.). https://openreview.net/forum?id=8LeCgKb6UX

[16] NVIDIA Corporation. 2023. NVIDIA H100 Tensor Core GPU Architecture. https://resources.nvidia.com/en-us-tensor-core Accessed: 2023-11-28.

[17] Timothy A. Davis. 2019. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. 45, 4, Article 44 (Dec. 2019), 25 pages. https://doi.org/10.1145/3322125

[18] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 3844–3852.

[19] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 1–13.

[20] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2024. DTC-SpMM: Bridging the Gap in Accelerating General Sparse Matrix Multiplication with Tensor Cores *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 253–267. https://doi.org/10.1145/3620666.3651378

[21] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The world wide web conference*. 417–426.

[22] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[23] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[24] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[25] Insu Han, Rajesh Jayaram, Amin Karbasi, Vahab Mirrokni, David P. Woodruff, and Amir Zandieh. 2023. HyperAttention: Long-context Attention in Near-Linear Time. arXiv:2310.05869 [cs.LG]

[26] Hatem Helal, Jesun Firoz, Jenna Bilbrey, Mario Michael Krell, Tom Murray, Ang Li, Sotiris Xantheas, and Sutanay Choudhury. 2022. Extreme Acceleration of Graph Neural Network-based Prediction Models for Quantum Chemistry. *arXiv preprint arXiv:2211.13853* (2022).

[27] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[28] Kezhao Huang, Jidong Zhai, Liyan Zheng, Haojie Wang, Yuyang Jin, Qihao Zhang, Runqing Zhang, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2024. Nollie: Optimizing GNN with Joint Workload Partition of Graph and Operations. In *Proceedings of The European Conference on Computer Systems (EuroSys)*. Aveiro, Portugal. April 22, 2024.

[29] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and Bridging the Gaps in Current GNN Performance Optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Association for Computing Machinery, New York, NY, USA, 119–132. https://doi.org/10.1145/3437801.3441585

[30] E. J. Im, K. A. Yelick, and R. Vuduc. 2004. SPARSITY: Framework for optimizing sparse matrix-vector multiply. *International Journal of High Performance Computing Applications* 18, 1 (Feb 2004), 135–158.

[31] U Kang and Christos Faloutsos. 2011. Beyond'caveman communities': Hubs and spokes for graph compression and mining. In *2011 IEEE 11th international conference on data mining*. IEEE, 300–309.

[32] Sheng-Chun Kao, Amir Yazdanbakhsh, Suvinay Subramanian, Shivani Agrawal, Utku Evci, and Tushar Krishna. 2022. Training Recipe for N: M Structured Sparsity with Decaying Pruning Mask. *arXiv preprint arXiv:2209.07617* (2022).

[33] George Karypis and Vipin Kumar. 1998. Multilevel algorithms for multi-constraint graph partitioning. In *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE, 28–28.

[34] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

[35] Yongsub Lim, U Kang, and Christos Faloutsos. 2014. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering* 26, 12 (2014), 3077–3089.

[36] J. Mellor-Crummey, D. Whalley, and K. Kennedy. 2001. Improving Memory Hierarchy Performance for Irregular Applications Using Data and Computation Reorderings. *International Journal of Parallel Programming* 29 (2001), 217–247. https://doi.org/10.1023/A:1011119519789

[37] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[38] NVIDIA. 2023. *cuSPARSELt: A High-Performance CUDA Library for Sparse Matrix-Matrix Multiplication*. https://docs.nvidia.com/cuda/cusparselt/index.html

[39] Jordi Petit. 2003. Experiments on the minimum linear arrangement problem. *Journal of Experimental Algorithmics (JEA)* 8 (2003).

[40] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. 2005. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Comput.* 31, 8-9 (2005), 858–876.

[41] Oleg Platonov, Denis Kuznedelev, Michael Diskin, Artem Babenko, and Liudmila Prokhorenkova. 2023. A critical look at evaluation of GNNs under heterophily: Are we really making progress?. In *The Eleventh International Conference on Learning Representations*.

[42] Jeff Pool and Chong Yu. 2021. Channel permutations for n: m sparsity. *Advances in Neural Information Processing Systems* 34 (2021), 13316–13327.

[43] Ilya Safro and Boris Temkin. 2011. Multiscale approach for the network compression-friendly ordering. *Journal of Discrete Algorithms* 9, 2 (2011), 190–202.

[44] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. 2018. Pitfalls of Graph Neural Network Evaluation. *ArXiv* abs/1811.05868 (2018). https://api.semanticscholar.org/CorpusID:53303554

[45] M. M. Strout and P. D. Hovland. 2004. Metrics and models for reordering transformations. In *Proceedings of the MSP '04*. Association for Computing Machinery, New York, NY, USA, 23–34.

[46] Wei Sun, Ang Li, Tong Geng, Sander Stuijk, and Henk Corporaal. 2022. Dissecting Tensor Cores via Microbenchmarks: Latency, Throughput and Numeric Behaviors. *IEEE Transactions on Parallel and Distributed Systems* 34, 1 (2022), 246–261.

[47] Wei Sun, Aojun Zhou, Sander Stuijk, Rob Wijnhoven, Andrew O Nelson, Henk Corporaal, et al. 2021. DominoSearch: Find layer-wise fine-grained N:M sparse schemes from dense neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 20721–20732.

[48] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019).

[49] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: Accelerating Quantized GNN via GPU Tensor Core. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (PPoPP'22)*.

[50] Yuke Wang, Boyuan Feng, Zheng Wang, Guyue Huang, and Yufei Ding. 2023. TC-GNN: Bridging Sparse GNN Computation and Dense Tensor Cores on GPUs. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. USENIX Association, Boston, MA, 149–164. https://www.usenix.org/conference/atc23/presentation/wang-yuke

[51] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.

[52] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. 2019. Simplifying Graph Convolutional Networks. In *Proceedings of the 36th International Conference on Machine Learning*. PMLR, 6861–6871.

[53] Xilinx. 2023. *AMD Xilinx Versal Adaptive Compute Acceleration Platforms*. https://www.xilinx.com/products/silicon-devices/acap/versal.html Accessed: 2023-10-19.

[54] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, Sourav S. Bhowmick, and Juncheng Liu. 2023. PANE: scalable and effective attributed network embedding. *The VLDB Journal* 32, 6 (March 2023), 1237–1262. https://doi.org/10.1007/s00778-023-00790-4

[55] Amir Zandieh, Insu Han, Majid Daliri, and Amin Karbasi. 2023. KDEformer: Accelerating Transformers via Kernel Density Estimation. In *Proceedings of the 40th International Conference on Machine Learning* (Honolulu, Hawaii, USA) *(ICML'23)*. JMLR.org, Article 1701, 19 pages.

[56] Chengming Zhang, Tong Geng, Anqi Guo, Jiannan Tian, Martin Herbordt, Ang Li, and Dingwen Tao. 2022. H-gcn: A graph convolutional network accelerator on versal acap architecture. In *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 200–208.

[57] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, and Xipeng Shen. 2010. Streamlining GPU Applications On the Fly. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*. Tsukuba, Japan.

[58] Eddy Z. Zhang, Yunlian Jiang, Ziyu Guo, Kai Tian, and Xipeng Shen. 2011. On-the-Fly Elimination of Dynamic Irregularities for GPU Computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, California, USA.

[59] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2022. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective. *Proceedings of Machine Learning and Systems (MLSys)*.

[60] Kaige Zhang, Xiaoyan Liu, Hailong Yang, Tianyu Feng, Xinyu Yang, Yi Liu, Zhongzhi Luan, and Depei Qian. 2024. Jigsaw: Accelerating SpMM with Vector Sparsity on Sparse Tensor Core. In *Proceedings of the 53rd International Conference on Parallel Processing* (Gotland, Sweden) *(ICPP '24)*. Association for Computing Machinery, New York, NY, USA, 1124–1134. https://doi.org/10.1145/3673038.3673108

[61] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.

[62] Yue Zhao, Jiajia Li, Chunhua Liao, and Xipeng Shen. 2018. Bridging the gap between deep learning and sparse matrix format selection. In *Proceedings of the 23rd ACM SIGPLAN symposium on principles and practice of parallel programming*. 94–108.

[63] Yue Zhao, Weijie Zhou, Xipeng Shen, and Graham Yiu. 2018. Overhead-conscious format selection for SpMV-based applications. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 950–959.

[64] Da Zheng, Xiang Song, Chengru Yang, Dominique LaSalle, and George Karypis. 2022. Distributed hybrid CPU and GPU training for graph neural networks on billion-scale heterogeneous graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 4582–4591.

[65] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning N:M Fine-grained Structured Sparse Neural Networks From Scratch. In *International Conference on Learning Representations*. https://openreview.net/forum?id=K9bw7vqp_s

[66] Weijie Zhou, Yue Zhao, Xipeng Shen, and Wang Chen. 2019. Enabling Runtime SpMV Format Selection through an Overhead Conscious Method. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 80–93.