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

Ruifeng Zhang*
North Carolina State University
Raleigh, NC, USA
rzhang38@ncsu.edu

Jou-An Chen†
Qualcomm
San Diego, CA, USA
carolc830708@gmail.com

Hsin-Hsuan Sung†
Qualcomm
San Diego, CA, USA
hsinhsua@qti.qualcomm.com

Ang Li
Pacific Northwest National Lab
and University of Washington
WA, USA
ang.li@pnnl.gov

Xipeng Shen
North Carolina State University
Raleigh, NC, USA
xshen5@ncsu.edu

Abstract

Recent GPUs incorporate Sparse Tensor Cores (SPTC) to accelerate computations on matrices that satisfy structured N:M sparsity; software stacks further extend support to generalized V:N:M patterns. Although graphs in graph neural networks (GNNs) are typically sparse, their sparsity is irregular and rarely conforms to these patterns. This paper introduces a lossless graph reordering algorithm that reshapes irregular graph data into the required sparse patterns, allowing GNN workloads to exploit SPTC. The transformation preserves model accuracy and maintains the symmetry of graph adjacency matrices, ensuring compatibility with symmetry-based graph algorithms. On the SuiteSparse collection, our method removes 93.77%–98.51% of vector-level N:M violations and increases the fraction of conforming graphs from 5%–7% to 79.3%–91.3%. On NVIDIA A100 GPUs, it accelerates SpMM by up to 50.7× (geometric-mean speedups 3.44 × –7.25×) over cuSPARSE, and speeds up key GNN graph operations on real graphs by as much as 6.7× (2.6× on average).

Keywords: GNN, Sparsity, GPU, Graph Reorder

1 Introduction

Graph Neural Networks (GNNs) [20, 28, 40, 64] have been widely used on graph-related problems because they capture and propagate rich relational information among nodes and edges. They already underpin a wide spectrum of applications in social network analysis [24], large-scale recommendation [45], and molecular property prediction [27, 30], among others. As problem sizes grow and graphs become ever larger and more irregular, however, the cost of the underlying linear-algebraic kernels quickly dominates, and overall throughput emerges as the primary system bottleneck. The present work is aimed squarely at alleviating that bottleneck.

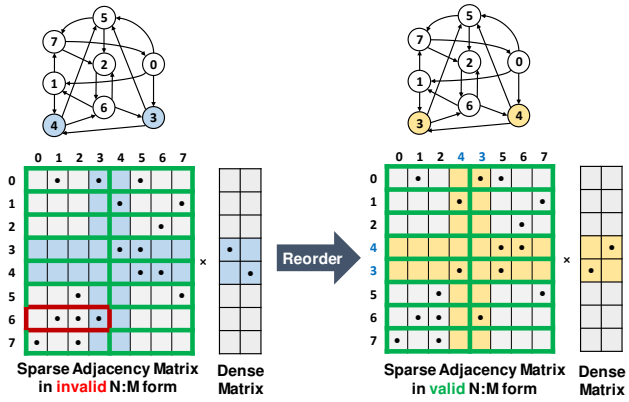


Figure 1. Graph vertex relabeling induces matched row/column permutations of the adjacency matrix, turning previously nonconforming rows (illustrated for row 6) into N:M-compliant segment vectors (2:4 shown). The corresponding rows of the dense feature matrix must be permuted consistently.

We introduce a new path that boosts GNN performance on the underlying matrix-multiplication kernels: *graph reordering*. It transforms the input into accelerator-friendly sparsity pattern for modern GPUs. Specifically, we target Sparse Tensor Cores (SPTC), which are available on recent accelerators [4, 18, 57]—notably on NVIDIA Ampere and newer generations—with analogous capabilities increasingly appearing on other AI/ML hardware (e.g., AMD/Xilinx ACAPs) [66, 69]. SPTC accelerate sparse fused multiply–accumulate instructions (`mma.sp`), the workhorse behind sparse–dense SpMM, and they are designed around *N:M* sparsity: within each length-M *segment vector*, at most N entries may be nonzeros. With the right software abstractions, these primitives extend naturally to *V:N:M* generalizations (Section 2), as demonstrated by Venom [11], which can deliver further gains beyond dense tensor-core execution.

*Corresponding author. Phone: +1-828-505-6741 Address: 405 Wolf View Dr, Raleigh, NC, 27606.
†This work was done while the authors were affiliated with North Carolina State University.

Although GNN workloads commonly feature sparse adjacency matrices, the fine-grained regularity required by patterns such as 2:4 is rarely present without additional processing. In our survey of 1,296 SuiteSparse graphs [19], only 5–7% of inputs conform out of the box (Section 5.3), leaving substantial performance untapped. This observation motivates us to reshape sparsity into SPTC-compatible forms while preserving the semantics of the original graph.

We therefore introduce **N:M sparsity–oriented graph reordering**: a family of vertex permutations that convert as many nonconforming segment vectors as possible into N:M-compliant ones by applying the same permutation to both rows and columns of the adjacency. In contrast to generic matrix reorderings [73], our transformations only renumber vertices, thereby preserving the graph's structure and semantics. For instance, swapping labels 3 and 4 swaps rows 3/4 and columns 3/4, but the underlying graph remains unchanged. This design maintains symmetry, so algorithms that critically rely on symmetric adjacency—e.g., isomorphism testing, graph partitioning, or minimum spanning tree—remain applicable without modification. The symmetry constraint, however, introduces additional combinatorial coupling across rows and columns, making the problem more challenging than unconstrained matrix permutations.

Reordering has been explored to reduce nonzero blocks or improve storage/compute locality [6–8, 10, 17, 21, 39, 63, 69, 78], but, to our knowledge, not specifically to enforce N:M (or V:N:M) sparsity for SPTC execution while preserving adjacency symmetry. Selecting high-quality permutations for arbitrary graphs and target patterns is nontrivial. Because the same graph is typically reused across many forward passes (even as node features change), the transformation can be applied offline, yet scalability is still essential. Computing a globally optimal permutation is NP-hard [7, 8, 17, 21], rendering exhaustive search infeasible. Jigsaw [73] touches on related ideas, but by treating the task as a generic matrix reordering, it sacrifices adjacency symmetry and other graph-specific properties that we explicitly preserve.

Our approach is a two-stage iterative method paired with an efficient GPU realization based on bit-level primitives. The algorithm, which we call *dual-level N:M–oriented reordering*, alternates between a *tile-level* pass and a *vector-level* pass. At the tile level (where a tile aggregates multiple segment vectors), we encode each segment as a bit string and apply a Hamming-distance–based position encoding [29, 68] to reorder rows and columns jointly; at the vector level, we employ a greedy scheme to satisfy per-vector constraints efficiently [5, 47]. These stages are composed in an iterative loop, apply broadly across graphs and patterns (N:M and V:N:M), and run in $O(n \log n)$ time.

To operationalize these ideas, we build *SOGRE* (Sparsity-Oriented Graph Reordering), a CUDA library that stores adjacency vectors as compact bit strings and implements the core routines using fast bit operations and intra-warp intrinsics. SOGRE integrates with widely used GNN software stacks, including PyG [26] and DGL [59], and can be slotted in as a preprocessing stage (or periodically for evolving graphs).

Because our transformation only relabels vertices, model accuracy is preserved. On 1,356 adjacency matrices from SuiteSparse, the method eliminates 93.77–98.51% of N:M violations at the vector level and boosts the fraction of conforming graphs from 5–7% to 79.3–91.3%. On NVIDIA A100 GPUs, the resulting SpMM achieves speedups of up to 50.7× (with 3.44×–7.25× on average), and end-to-end GNN operations on 12 real graphs realize up to 6.7× (2.6× on average) over cuSPARSE. The reordering itself averages 0.13–43.84 s, which is practical for offline use when graphs are reused across many inferences. Meanwhile, we propose *global schedules* to make the algorithm adaptive to different time constraints.

2 Terms and Background

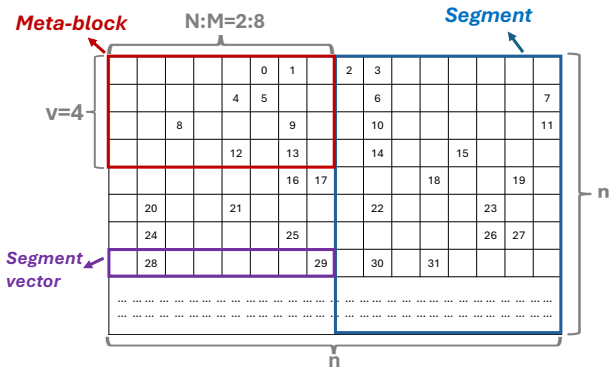
This section provides the background needed for understanding the rest of the paper. (For the broader background on SPTC hardware and VENOM, please see references [4, 11, 18, 47].)

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

GNN Background GNN [20, 28, 40, 64] 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.

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

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).



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.

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 $\frac{\text{initial } \omega - \text{final } \omega}{\text{initial } \omega}$, where ω represents the count of segment vectors that violate the required sparse pattern. The best improvement rate is 1.0, indicating that no segment vectors violate the required pattern after reordering.

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 ϕ, V:N:M pattern
Output: New vertex order

```
while V:N:M violations remain and max. iteration is not yet reached do
    ϕ = Stage-1.reorder(A, V, N, M, ϕ)
    ϕ = Stage-2.reorder(A, N, M, ϕ)
```

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* [68] 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 [68] 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 follows. Figure 3 uses an example to illustrate each of the steps.

Algorithm 2: Stage-1 Algorithm for Increasing Vertical Conformity

Input: Graph adjacency matrix A with vertex order ϕ, V:N:M pattern
Output: New vertex order ϕ'

```
iter ← 0
MBscore ← GetMbScore(A, V, N, M)
while MBscore > 0 and iter ≤ MAXITER do
    Initialize 2D matrix Vec to store the segment vector encodings per row (per vertex)
    for i ← 0 to n − 1 do
        for s ← 0 to ⌈n/M⌉ − 1 do
            bstr ← binary string representation of A[i][s]
            Vec[i][s] ← Hamming position code of bstr
            if bstr is invalid N : M pattern then
                Vec[i][s] ← −Vec[i][s]
    Sort rows in Vec
    Get the sorted order of Vec as ϕ′
    Reorder A with ϕ′; ϕ ← ϕ′ iter ← iter + 1
    MBscore ← GetMbScore(A, V, N, M)
```

(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.

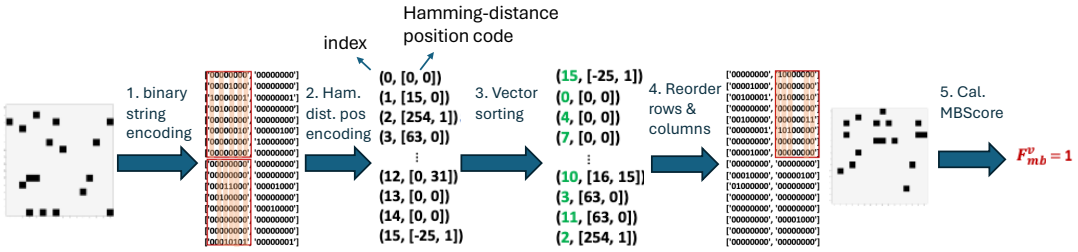


Figure 3. One iteration of the Stage-I algorithm (Alg. 2) for reducing vertical constraint violations. The example targets a V:N:M format as 8:2:8. It reduces the number of meta-blocks violating the vertical constraints (orange-covered blocks) from 2 to 1.

(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, it matters what order to follow when working with the segments in an adjacency matrix. An intuitive 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*. In this intuitive strategy, 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. Besides this intuitive strategy, we investigate several other strategies in choosing the *primary segment* and the *target segment*, discussed in Section 4.4.

Algorithm 3 outlines the algorithm that uses the intuitive strategy. 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]

[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.

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 $O(n)$.

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

Input: Graph adjacency matrix A with vertex order ϕ, N:M pattern
Output: New vertex order ϕ'

```
1  I ← GetPScoreList(A, N, M) // A priority list
2  Exclude valid segments from I
3  while |I| ≥ 1 and iter ≤ MAXITER do
4      S ← ∅
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) ← I.pop()
10             for (segID_targ, pscore_targ)=I.next() do
11                 swap_cands ← GetCandidates(segID_prim, segID_targ)
12                 Vertex pair (u, v) ← swap_cands.freshtop()
13                 S ← S ∪ {(u, v)}
14                 pscore_prim ← pscore_prim − u.gain
15                 pscore_targ ← 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 φ' with φ
21     for (u, v) ∈ S do
22         Swap u and v in φ'
23     Reorder A with φ'; φ ← φ'
24     iter ← iter + 1
25     I ← GetPScoreList(A, N, M) // update I
26     Exclude valid segments from I
```

4.4 Segment Selecting Strategies

One of the important steps in the stage-2 reordering is to select the *primary segment* and the *target segment*. There can be various *segment selecting strategies* to do that. The previous section has describes an intuitive strategy, that is, choosing the worst of all remaining segments (i.e., the one that has the largest PScore) as the primary segment, and the worst of all yet-to-check segments as the target segment. But in addition to that strategy, we also investigated three other strategies. We list all the four strategies as follows, with the previously described intuitive strategy as the first one:

1. Primary = worst (largest PScore); targets = worst in yet-to-check segments.
2. Primary = worst; targets = best (with the smallest positive PScore) in yet-to-check segments.
3. Primary = best; targets = best in yet-to-check segments.
4. Primary = best; targets = worst in yet-to-check segments.

The *worst* segments contain a relatively large number of conflicts and are hard to reorder; the *best* segments contain few conflicts and are easy to reorder.

The intuition behind the *default strategy* is to address the bottleneck first. We choose as the *primary* segment the one with the most conflicts and as the *target* the second-worst among the remaining segments. If the worst segment is postponed, the instance remains unsolved even after all other segments are successfully reordered. Swapping between two hard-to-reorder segments tends to reduce the dominant conflicts early, smoothing the problem for later steps.

Strategy 2 keeps the same primary but switches the target to the *best* segment. This is also intuitive: nodes from low-conflict segments typically have smaller degree and are less constrained, making it easier to clean up conflicts in the primary via exchanges with them.

Strategies 3 and *4* invert the emphasis (an *easy-first* approach) by selecting a low-conflict segment as primary. This can be beneficial for larger inputs: many conflicts are eliminated quickly, shrinking the candidate set and reducing subsequent computation. *Strategy 3* pairs the primary with the second-best target, rapidly resolving easy segments but risking a stubborn residual composed only of hard segments, which becomes harder once the easy parts are exhausted. *Strategy 4* mitigates this by using the worst segment as the target, aiming to clear easy segments while simultaneously chipping away at the hard core.

Recall that the overall reordering algorithm is iterative (as shown in Section 4.1) and both stage-1 and stage-2 may need to run a number of times. It is possibly beneficial for the different rounds of stage-2 to use different segment selecting strategies. We call the overall scheme that dictates what strategy to use in which rounds *global schedule*. We investigate a set of global schedules as explained next.

4.5 Global Schedule

We have explored four global schedules, as shown as follows.

4.5.1 S1: Single Best. The naive solution is only picking one strategy from stage-2 reordering, and perform *Stage-1* - *Stage-2* loop until successful reorder or maximum step is reached. The strategies are ranked with first improve rate then reorder duration for each problem, and the overall best

strategy for all formats. In our experiment, Strategy-0 tends to be the overall best strategy.

4.5.2 S2: All-Strategy Traversal. Run full *Stage-1 - Stage-2* reordering for four times—once per strategy—stopping early if any strategy achieves a 100% improvement rate, which means the matrix has been successfully reordered. If none reaches 100%, pick the best outcome by improvement rate, and break tie by runtime. This schedule maximizes success rate but is up to ~4× more expensive. The empirically best try order is

$$1 \rightarrow 3 \rightarrow 2 \rightarrow 4.$$

4.5.3 S3: Probe then Commit. We first *probe* the Stage-2 strategy by running a single round for each of the four candidates, then select the one with the highest improvement rate (breaking ties by reordering time). We subsequently execute the full *Stage-1* $\rightarrow$ *Stage-2* loop using the chosen strategy. This approach limits the upfront cost to one Stage-2 round per candidate before committing; however, probing all strategies still adds overhead—even on easy instances—and the one-round estimate can be inaccurate in some cases.

4.5.4 S4: Adaptive Cycling. Rather than committing to a single strategy per matrix, we *cycle* through strategies and switch whenever a round yields no improvement (i.e., the improvement rate does not increase). Heuristically, Strategy 1 often performs best by addressing difficult regions early, so we start with it by default. Strategy 2 has a similar focus but targets an easier region, which complements Strategy 1 when some hard regions cannot be reordered immediately. Strategy 4 approximately mirrors Strategy 2 and is placed last to avoid back-to-back inverses. Guided by this heuristic and corroborated by empirical results, we adopt the order

$$1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \circlearrowright .$$

4.5.5 S5: Learned Policy. We train an XGBoost [14] classifier on matrix features to predict the best Stage-2 strategy. At inference, we execute *Stage-1* followed by *Stage-2* using the predicted strategy—removing probing overhead at the expense of relying on prediction accuracy.

We consider two labeling schemes. *(i) Naïve.* The model predicts the Stage-2 strategy that yields the highest improvement rate on a given matrix; ties are broken by shorter reordering time. We refer to the resulting schedule as *learned policy (naive). (ii) Expanded labels.* We augment the label space with two meta-classes: `all` (every strategy succeeds) and `none` (no strategy succeeds), allowing the model to leverage the full dataset instead of forcing arbitrary choices. At inference, `all` maps to the strategy with the smallest reordering overhead, while `none` defaults to *Adaptive Cycling* (AC), which aggregates multiple strategies and offers a higher chance of success. We call this schedule *learned policy (expand).*

For both models, the input features include matrix dimensions (m, n), global sparsity, and summary statistics (mean and standard deviation) of nonzero counts computed over rows, columns, diagonals, and invalid-segment vectors across all segments. These capture reordering difficulty, distributional heterogeneity, and salient structural patterns. We tuned several tree configurations and selected 30 trees with maximum depth 8 and learning rate 0.01. For evaluation, we use standard 5-fold cross-validation and report out-of-fold predictions over the entire dataset.

4.6 Zero-tile Skipping

We observe that all-zero MetaBlocks are common in our sparse matrices, and that reordering often increases their prevalence—especially for larger, sparser inputs. This arises because Stage-1 reordering groups similar rows, so vertices with few edges are likely clustered together, forming empty blocks. The original VENOM [11] does not exploit this property. We therefore introduce an *active-block list*: after reordering, we mark MetaBlocks that contain at least one nonzero and pass their indices to the VENOM kernel, which then skips computation on empty MetaBlocks and avoids redundant work.

It's worth noting that this is a lightweight patch to VENOM: we pass a compact binary bitmap of active blocks directly to the kernel, without changing the underlying storage (i.e., we do not unload all-zero blocks from the stored sparse matrix) or its global→shared memory access pattern. A more comprehensive design could (i) remove memory redundancy by omitting storage for all-zero blocks and (ii) leverage the active-block list to guide thread-block scheduling so that empty regions are skipped while maintaining coalesced memory access and load balance (e.g., dynamically reassigning work to mitigate imbalance). However, these kernel-level optimizations are beyond the scope of this work.

4.7 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` [47] and `SPATHA (VENOM)` [11], all have a limit at the level of about 45K-by-45K matrices. Empirically (Section 5.3), 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]);
```

4.8 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 5.3 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 25. 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. We use Spatha, an SpMM library that takes advantage of SPTCs, as the backend kernel to execute SpMM on the reordered graphs. Spatha was developed in a prior VENOM work [11] for DNNs. It is the only available library that supports V:N:M patterns. While it's original implementation doesn't include zero-tile skipping, we incorporate this patch by default. Section 5.5 compares the benefits of reordering with and without zero-tile skipping.

The results are the implementation of our method with "S2: All-Strategy Traversal" and "S4: Adaptive Cycling" schedule (details are in Section 5.4.4). Section 5.4 gives the comparisons among the various schedules.

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

		#V	#E	Sparsity	#Graphs
Small	*Avg*	428	4,981	96.13%	345
	Med	432	2,196	98.23%	
Medium	*Avg*	2,911	46,722	99.35%	632
	Med	2,048	20,705	99.60%	
Large	*Avg*	14,367	328,141	99.81%	319
	Med	13,252	156,089	99.91%	

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 1396 adjacency

Table 2. GNN graph dataset.

Dataset	#V	#E	#Features	#Classes
Cora [67]	2,708	10,556	1,433	7
Citeseer [67]	3,327	9,104	3,703	6
Facebook [67]	4,039	88,234	1,283	193
Computers [55]	13,752	491,722	767	10
CS [55]	18,333	163,788	6,805	15
CoraFull [9]	19,793	126,842	8,710	70
Amazon-ratings [51]	24,492	93,050	300	5
Physics [55]	34,493	495,924	8,415	5
ogbn-proteins [32]	132,534	39,561,252	128	2
ogbn-products [32]	2,449,029	61,859,140	100	47
ogbn-arxiv [32]	169,343	1,166,243	128	40
ogbn-papers100M [32]	111,059,956	1,615,685,872	128	172

matrices[2] of the real-world graphs included in the SuiteSparse Matrix Collection [19]. 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)* [40] (2) *GraphSAGE (SAGE)* [28] (3) *Chebyshev Spectral Graph Convolutional Neural Networks (Cheb)* [20] (4) *Simplifying Graph Convolutional Networks (SGC)* [64].

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) [26] and Deep Graph Library (DGL) [59]. Both libraries are well optimized for GPUs and have supported a broad range of existing research works [20, 28, 40, 64]. 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 the two frameworks by replacing their SpMM kernels with Spatha SpMM library from VENOM. 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 4.5X for the GCN model and 2.9-9.4X at maximum for SGC. These translate into an end-to-end speedup of 1.1-3.8X for GCN and 2.2-6.7X 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

[2]Compared with the conference version of this work [13], we (i) include a larger share of *large* matrices−−which are typically harder to reorder under a fixed time budget, and (ii) remove near-duplicate instances (pairs with identical dimensions and $> 99\%$ nonzero-pattern similarity) to reduce bias and make the dataset more representative.

Table 3. Normalized speedup of four GNN models compared to the default PyG [26] and DGL [59] (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. Blue numbers in brackets indicate results from Table 1. Results in the blue brackets show the speedup without applying zero-tile skipping.

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	1.44[1.41]	1.13[1.12]	2.34[2.13]	1.16[1.15]	2.45[2.33]	1.56[1.53]	2.87[2.63]	2.15[2.03]	1.28[1.18]	1.12[1.08]	2.06[2.01]	1.28[1.27]	4.09[3.80]	1.35[1.34]	2.26[2.06]	1.77[1.67]
Citeseer	32:2:8	4.49[2.11]	1.17[1.11]	4.53[2.84]	1.20[1.16]	6.35[3.16]	1.94[1.65]	5.15[3.40]	1.87[1.69]	2.70[1.71]	1.14[1.09]	4.78[2.92]	1.44[1.34]	9.12[4.49]	1.56[1.46]	4.93[2.94]	4.21[2.71]
Facebook	1:2:4	3.60[3.26]	1.69[1.64]	6.50[6.01]	2.27[2.23]	9.32[8.56]	4.37[4.22]	9.37[8.64]	6.74[6.37]	3.92[3.27]	1.71[1.63]	6.87[6.01]	2.51[2.42]	4.82[4.41]	3.14[2.98]	7.05[6.39]	5.07[4.74]
Computers	1:2:4	3.34[2.72]	3.26[2.99]	3.95[3.83]	2.91[2.85]	6.30[5.31]	4.52[4.02]	6.10[5.30]	5.81[5.15]	4.32[3.30]	3.19[2.65]	4.58[4.12]	3.10[2.91]	3.74[2.90]	3.64[3.06]	5.14[4.13]	4.94[4.05]
CS	16:2:16	1.76[1.73]	1.11[1.11]	3.39[3.12]	1.85[1.79]	5.33[4.36]	2.85[2.60]	4.39[4.38]	4.22[4.21]	2.57[1.85]	1.17[1.12]	2.99[2.78]	1.60[1.56]	2.77[2.12]	1.99[1.70]	3.67[2.69]	3.50[2.61]
CoraFull	32:2:16	1.57[1.44]	1.06[1.05]	2.13[2.05]	1.42[1.40]	3.04[2.91]	2.00[1.96]	3.03[2.95]	2.80[2.73]	1.82[1.53]	1.08[1.06]	2.05[1.96]	1.32[1.30]	2.07[1.96]	1.42[1.39]	1.92[1.88]	1.82[1.79]
Amazon-ratings	1:2:32	3.89[2.01]	3.78[1.99]	7.81[3.42]	2.56[1.98]	7.39[4.24]	2.82[2.20]	8.22[4.30]	6.54[3.85]	3.17[1.30]	3.11[1.43]	4.00[1.99]	1.71[1.38]	4.43[2.00]	1.56[1.30]	3.58[1.80]	3.01[1.70]
Physics	16:2:16	1.86[1.56]	1.19[1.14]	3.16[3.00]	2.36[2.28]	4.48[4.46]	3.22[3.21]	5.77[4.76]	5.50[4.68]	2.19[1.80]	1.18[1.14]	2.60[2.23]	1.73[1.61]	2.38[2.20]	1.78[1.70]	2.49[2.20]	2.45[2.18]

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, 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.

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.

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%)

Table 6. OGBN [32] 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

Baselines. We compare our work against the following libraries:

- **cuSPARSE**: The standard NVIDIA library for sparse linear algebra.
- **Block-SpMM** [46]: NVIDIA's block-wise SpMM kernels leveraging block sparse format and Tensor Cores. We use block size of 32×32 for the experiments.
- **Hidayetoğlu et al. (SpDNN)** [31]: This library focuses on efficient sparse deep neural network inference. For brevity, we refer to this baseline as *SpDNN* in our evaluation.
- **nmSPARSE** [42]: This library is also optimized for N:M-sparse weight structures, but it utilizes Tensor Cores instead of Sparse Tensor Cores.

Speedups. Figure 4 shows normalized speedups versus the aforementioned baselines as we vary the dense multiplier width $N \in \{128, 512, 2048\}$, matching typical embedding sizes. We apply zero-tile skipping on medium and large matrices, and all matrices are reordered to the best possible format. That is for the target $V : N : M$ formats, we try to reorder them to the largest M then the largest V. Our approach achieves significant speedups compared to cuSPARSE across a broad range of matrix sizes, particularly in medium and large graphs where sparsity and structure afford greater reordering leverage. Only a small fraction ($\approx 4.5\%$) of matrices exhibit a slowdown after reordering; these are typically extremely sparse (density $< 0.01\%$) and occur within large matrices. In such cases, processing exclusively non-zero elements on standard CUDA cores can outperform our Sparse Tensor Core (SPTC) approach, which may still encounter zero elements due to packing requirements. Furthermore, performance gains generally scale with the column count of the dense multiplier (N), as larger N values typically yield better speedups. For small N values, such as $N = 128$, the L1 cache hit rate of SpMM kernel is higher, making cache utilization the primary bottleneck and narrowing the performance gap between libraries using different computation cores. While Block-SpMM, SpDNN, and nmSPARSE perform similarly to our approach at this specific setting, our speedup remains more significant than those existing solutions across the vast majority of input scenarios. Note that our zero-tile skipping is somewhat like Block-SpMM, which achieves speedup by skipping whole tiles, but our approach utilizes Sparse Tensor Cores and is

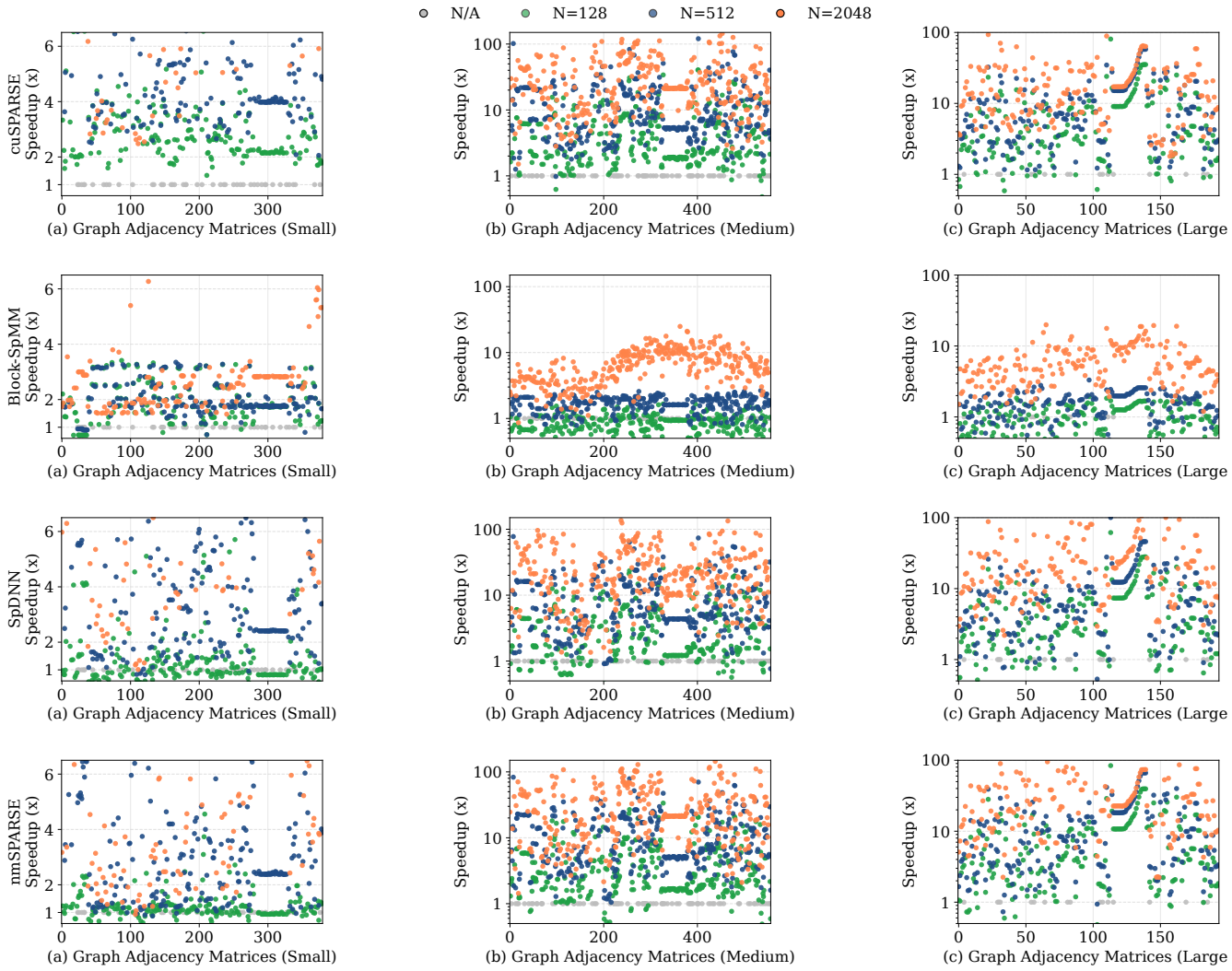


Figure 4. Speedup of SpMM over cuSPARSE after reordering matrices to their best V:N:M formats. The x-axis enumerates SuiteSparse graphs; N is the column count of the dense multiplier; N/A means the matrix cannot be reordered into any compliant formats so speedup is not applicable. (a) Small. (b) Medium. (c) Large.

more efficient in most cases. Since the reordering process is conducted offline, users can simply skip this step for the specific hyper-sparse graphs that do not benefit from the transformation.

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 93.77–98.51% for different graph sizes, and 95.54% on average, and increases the proportion of conforming matrices from 5.21% to 87.11% for small graphs, 6.54% to 79.28% for medium graphs, and 5.93% to 91.26% for large graphs.

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

Table 7. 1:2:4 reordering quality on SuiteSparse. "#inv segvec" is the number of invalid segment vectors in 1:2:4 (i.e., $F_p(\phi)$). "Iter." is the number of iterations until convergence. (*Med: Median*)

		Init. #inv segvec	Finl. #inv segvec	Imprv. rate	Iter.	Reorder time (s)
Small	*Avg*	61.22	5.72	96.02%	73.20	0.13
	Med	34.00	0.00	100.00%	15.00	0.01
Medium	*Avg*	554.60	58.64	93.77%	589.39	8.02
	Med	336.00	0.00	100.00%	85.00	1.88
Large	*Avg*	1,783.93	94.43	98.51%	1,195.42	43.84
	Med	1,760.00	0.00	100.00%	263.00	33.45

Table 8. Reordering success rate (%) on SuiteSparse across different V constraints. (For 2:4 format, results are identical for any V.)

Size	Format	V=1	V=4	V=8	V=16	V=32
	V:2:4	87.11	–	–	–	–
Small	V:2:8	73.17	29.03	9.44	5.40	1.35
	V:2:16	31.91	12.41	4.70	0.58	0.33
	V:2:4	79.28	–	–	–	–
Medium	V:2:8	70.03	29.98	16.70	15.00	11.61
	V:2:16	48.23	22.46	12.30	8.24	4.44
	V:2:4	91.26	–	–	–	–
Large	V:2:8	85.86	36.62	25.59	22.94	15.89
	V:2:16	65.14	26.71	18.11	12.08	5.18

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 [76, 77, 80], 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.

5.4 Comparisons of Reordering Strategies and Schedules

This section reports in-depth comparison among the various strategies for selecting columns/rows for reordering and the various global schedules.

5.4.1 Comparisons of Various Strategies. As discussed in Section 4.4, we have explored four different strategies in selecting the segments in a matrix for column/row reordering in stage-2. This section reports the comparisons among them.

Table 9 reports the average success rates for each individual strategy and, for comparison, the rate achieved when selecting the best strategy per matrix (*Best-of-all*). *Strategy-1* performs best overall: for example, on medium-sized matrices targeting the 1:2:8 format, it attains a 65.56% success rate—nearly twice that of the next best (*strategy-2*) (34.13%).

Table 9. Average Reorder success rate (%) on SuiteSparse. Comparison of stage-2 strategies (S_i stands for Strategy i) across different input sizes and formats.

Size	Format	S1	S2	S3	S4	Best
	1:2:4	**81.12**	74.38	76.85	77.08	86.52
Small	1:2:8	**64.94**	55.51	53.93	60.90	72.58
	4:2:8	**15.96**	15.73	15.06	14.38	27.19
	1:2:4	**70.41**	65.03	65.35	67.72	78.64
Medium	1:2:8	**65.56**	34.13	47.30	46.35	70.63
	4:2:8	**16.51**	15.26	15.26	14.79	29.57
	1:2:4	**85.89**	81.82	85.27	84.95	89.97
Large	1:2:8	**76.79**	63.04	69.34	67.62	83.38
	4:2:8	20.07	20.00	**22.09**	20.90	35.22

Table 10. Reordering results of different global schedules on different reorder formats. The details about the schedules are in Section 4.5.

Schedule	Metric(Avg)	1:2:4	1:2:8	4:2:8
	Imprv. rate (%)	93.73	91.87	81.20
Single best (strategy-1)	Success (%)	72.85	67.89	16.75
	Time (s)	16.37	14.64	16.81
	Imprv. rate (%)	94.40	**93.74**	**89.36**
All-Strategy traversal	Success (%)	**83.88**	**74.37**	**30.45**
	Time (s)	27.98	28.85	57.40
	Imprv. rate (%)	93.96	93.20	83.32
Probe then commit	Success (%)	80.01	68.40	22.25
	Time (s)	28.21	20.06	18.22
	Imprv. rate (%)	**94.56**	93.28	89.02
Adaptive cycling	Success (%)	81.30	72.75	26.88
	Time (s)	21.82	17.18	25.65
	Imprv. rate (%)	93.96	92.78	84.27
Learned policy (naive)	Success (%)	76.07	63.90	21.30
	Time (s)	**14.97**	**11.92**	**15.82**
	Imprv. rate (%)	94.29	92.80	88.01
Learned policy (expand)	Success (%)	79.15	68.68	26.93
	Time (s)	16.45	13.94	17.69

Even so, there is still some substantial potential if we do not limit to a single strategy for all matrices. Averaged over all sizes and formats, *best-of-all* improves the success rate by roughly 25% over strategy-1 (~ 1.25×). In the hardest setting—4:2:8 with large matrices, it increases the success rate from 22.09% to 35.22% (a 59% improvement).

These results suggest that a global schedule that adaptively selects the right strategy can be beneficial, especially for challenging cases, while a single strong strategy (strategy-1) may suffice for easier cases.

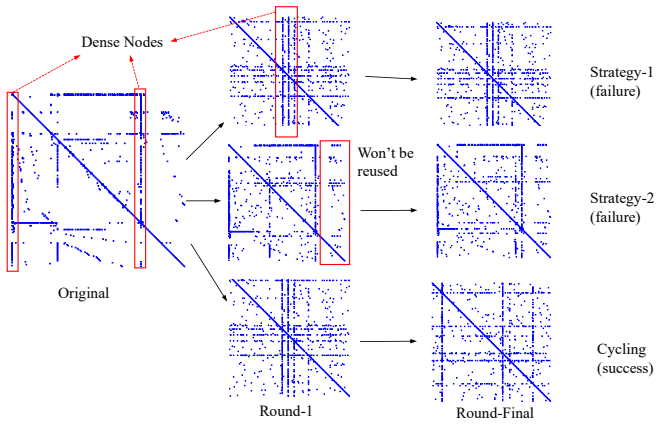


Figure 5. Different stage-2 strategies v.s. *adaptive cycling* schedule results on fs_183_1.

5.4.2 Comparisons of Global Schedules. The various schedules described in Section 4.5 offer different ways to leverage the various segment selection strategies. Table 10 reports their performance.

The time of the runtime of learned policies includes both feature extraction and model inference times. Unless noted otherwise, we set `MaxIter` to 50 and enforce an 800s time cap. The algorithm stops early if reordering is successful, or a round doesn't make any node swapping (the *adaptive cycling* schedule stops if all strategies don't give any improvement).

Single-best serves as the baseline. As discussed earlier, Strategy 1 is the best performing one and is hence used in this baseline schedule.

All-strategy traversal attains the highest improvement rate but at substantial time cost, roughly 1.7×–3.4× slower than the Single-best. The overhead grows as the target format becomes harder, since more strategies must be attempted to find a feasible path. For the 4:2:8 format, the average completion time reaches ~57s, which may be unsuitable for some resource-constrained/time-sensitive settings.

Probe-then-commit (PtC) reduces this overhead by running one round per candidate to choose a strategy, then committing to that choice. For easier formats such as 1:2:4 (few rounds), the benefit is modest; for harder cases like 1:2:8 and 4:2:8, PtC substantially reduces the time compared to the all-strategy traversal schedule, but the loss of quality is also significant. This indicates that the first-round winner may not be the global optimal.

adaptive cycling (AC) trims time further by switching strategies whenever a round fails to improve, and it doesn't have an explicit probing phase. This is most helpful on simpler formats where a dedicated probe is expensive. In practice, AC consistently outperforms PtC, and we observe inputs that AC solves even when every single strategy and the all-strategy traversal schedule fails. Figure 5 illustrates such a case with a small matrix: several *dense nodes* have far more connections than others have. Strategy 1 prioritizes segments that both contain many invalid vectors, naturally shuffling nodes near dense regions; dense nodes cluster after the first round, subsequent swaps keep targeting the same segments, and the progress stalls. Strategy 2 swaps between the worst and best segments (with invalid vectors), initially cleaning sparse regions (i.e. making them compatible to the desired format through node swapping) and excluding them from later swaps; however, the early exclusion of those regions reduces room for columns/rows swapping for other segments in the matrix, and reordering also fails. AC avoids these pitfalls: it begins with Strategy 1, detects stagnation, switches to Strategy 2, and successfully completes the reordering.

Naïvely learned policy offers only some slight gains over single-best. When multiple strategies succeed on an input, the model often defaults to Strategy 1. Nevertheless, the naïve model achieves the *lowest average time* by executing exactly one strategy and occasionally choosing better than the default——making it preferable to single-best in some practical settings.

Expanded learned policy augments labels with `all`/`none` and can sometimes outperform the all-strategy traversal schedule by sometimes falling back to AC. Overall its performance is close to AC, suggesting that predicting the optimal schedule from static, pre-reordering features is less effective than the iterative adapting schedule. We hypothesize that as reordering progresses, matrix structure changes and the best choice becomes context-dependent; re-extracting features and re-predicting each round would be costly.

Regarding time cost, it is essential to analyze *both* successful and failed cases separately (Figure 6), because failed reordering can be far more expensive than the successful ones: Successful runs usually terminate early, while the failed trials often happen on large, intrinsically hard cases. PtC spends extra time probing but terminates earlier than the all-strategy traversal schedule; the all-strategy traversal schedule performs well on successful cases but exhibits a long tail in the failed cases. AC lies between them, often faster to succeed than the traversal schedule, but with a longer time in the failed cases than PtC.

5.4.3 Quality of Different Schedules on Hard Problems. Hard-to-reorder cases more likely incur long reordering times and failed reordering results. They are hence worthy to be further examined to see how well each global schedule works on them and how much time they may save on them.

Reordering Hard Problems under Time Constraints. We define *hard problems* as those with a large amount of initial constraint violations. Concretely, we select the top 1/3 of matrices by the number of initial segment vectors that violate the required sparse pattern. We prefer this criterion over raw size/sparsity because very large but sparser matrices and smaller but denser matrices can be equally hard. Under this setting, we obtain a subset of ~500 SuiteSparse

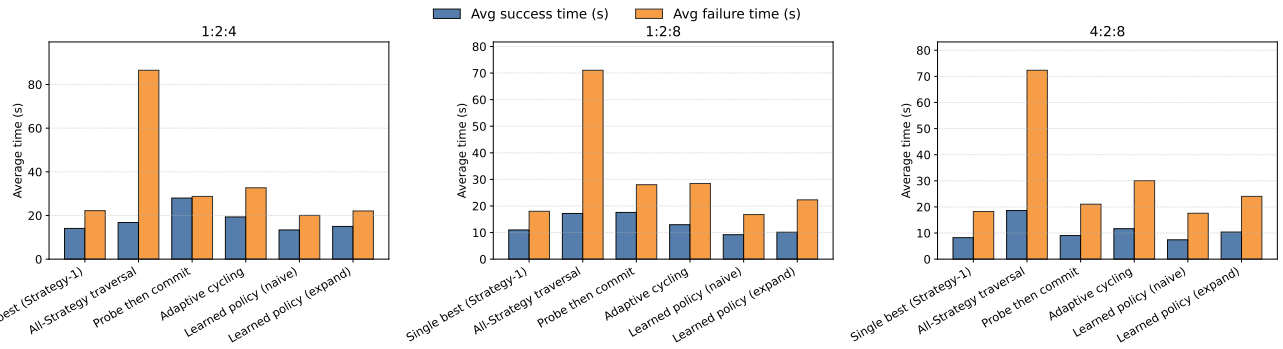


Figure 6. Average time cost of schedules on the successfully reordered subset and failure subset. Failure cases typically cost significantly more time.

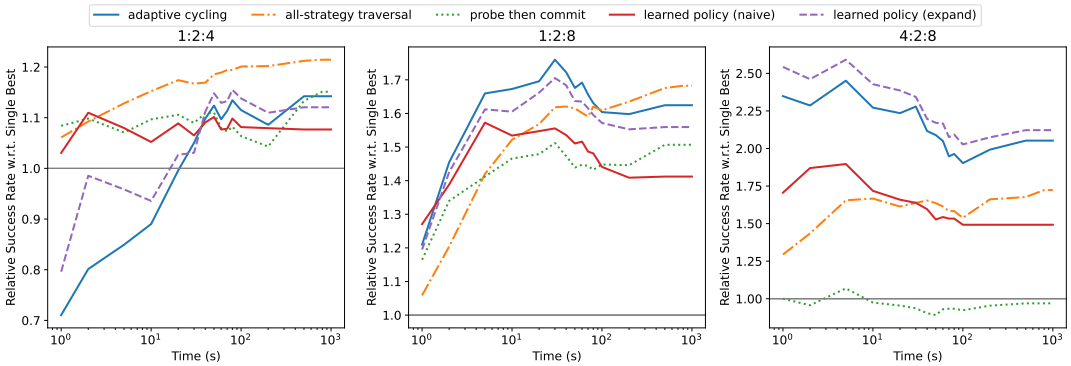


Figure 7. Relative success ratio of global schedules under different time constraint.

matrices, predominantly from the medium and large groups. The average count of segment vectors violating 1:2:4 format, for instance, is 662 per matrix in the whole dataset, and 1732 in the hard problem set.

Time constraints are also common in practice (e.g., dynamic GNN workloads [25, 53]), and from a compute-efficiency perspective it is essential to scrutinize the cost profile on hard inputs. Figure 7 reports the relative success rate of each global schedule against the single best static strategy as a function of a wall-clock budget (X-axis). When the time budget is tight and the target format is relatively easy, the single best static strategy often suffices: It achieves decent success without the overhead that *adaptive cycling* may pay if its first round yields no improvement. That said, *all-strategy traversal* can still be beneficial because it begins with the empirically best strategy and applies early stopping; it does not cost more time on instances that succeed early, while it can turn some failures into successes.

On the 1:2:8 format, all multi-strategy schedules outperform the single strategy markedly; *adaptive cycling* is strongest under budgets below ~100 s, while traversal requires more time to show a larger advantage. For the more challenging 4:2:8 format, the *learned policy* performs best on average: it maps many cases to *adaptive cycling* but also selects alternative strategies when they provide a time advantage. Overall, on these large/hard problems, using an appropriate schedule yields **1.1×–2.5×** higher success than the single strategy, with *all-strategy traversal* favored for easier formats and ample time, and *adaptive cycling* or *learned policy (expand)* preferred for complex formats or tighter budgets (see Table 10).

Cases Beyond Single Strategy. Another informative slice of hardness is the subset that the single best strategy cannot solve. We select the subset that strategy-1 cannot solve even for the basic 1:2:4 format, which contains about 30% of matrices.

Figure 8 analyzes this subset. On this subset, alternative schedules not only improve success but also potentially reduce time. Because the single strategy fails, it typically runs until the maximum iteration/time cap, which is highly inefficient and contributes disproportionately to the total runtime, especially on easier formats which could be easily solved by another strategy. Importantly, both traversal and *adaptive cycling* start from Strategy-1; thus, their behavior when the initial choice is wrong is crucial. While *all-strategy traversal* can achieve the highest success on some formats, it often pays a large cost by executing multiple (up to 4) strategies before stopping. In contrast, *adaptive cycling*——although it starts with Strategy-1——quickly switches when progress stalls (see the illustrative case in Figure 5). *Probe-then-commit* avoids

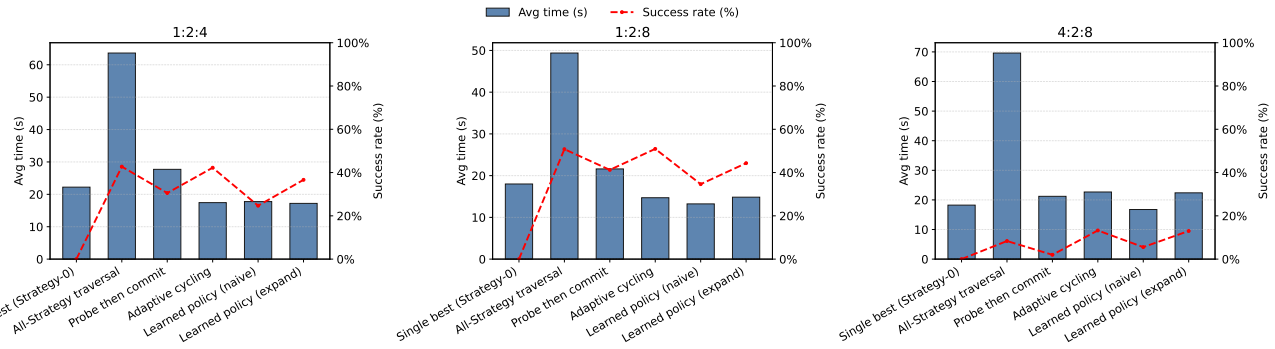


Figure 8. Performance of schedules on problems that strategy-1 cannot solve. The red dotted line indicates the success rate of schedules while the blue bar indicates the average time cost.

long tails by only probing in the first round, but because it could stick to the wrong probing result, it doesn't achieve a better success rate compared to traversal and cycling, and the probing is in general less efficient than cycling. The *learned policies* naturally evades worst-case trails by making flexible first-round selections. The naive model should work as a better alternative than probing, but it cannot achieve a better success rate. However, it is the most time-efficient. The model with expanded label, on the other hand, is on par with *adaptive cycling* in general, which corresponds to the previous discussions. In scenarios where long latency outliers are unacceptable, traversal should be avoided in favor of these dynamic schedules.

5.4.4 Making Selections. We consider two operating regimes for schedule selection. The first prioritizes *success* (runtime is secondary), which fits mostly static graphs that are not updated frequently. The second prioritizes *time* (runtime is constrained), which fits dynamic settings such as temporal GNNs [25, 53] where the graph evolves, or other resource-constrained systems that cannot tolerate long runtimes. As discussed earlier, schedule effectiveness varies markedly with problem size and format, so we partition instances into two splits: (i) *easy/small*, including simple target formats such as 1:2:4 or 1:2:8 and, empirically, with ≤2,000 total conflicts to resolve; and (ii) *hard/large*, otherwise. We use the 2,000-conflict threshold consistently unless noted.

Success prioritized. For tasks that emphasize reordering success over time (e.g., static graphs), we recommend:

- **Easy/small (≤2,000 conflicts; 1:2:4 or 1:2:8).** Use the *all-strategy traversal* schedule.
- **Hard/large (complex formats or >2,000 conflicts).** Use *adaptive cycling*.

For the evaluation in Section 5.3, success rate is the priority so we select schedules accordingly: all-strategy traversal for easy cases and *adaptive cycling* for hard cases. Note that, combining both (i.e. all-strategy traversal and *adaptive cycling*) can further improve success rates, but the improvement is marginal, and the expense of additional time is high.

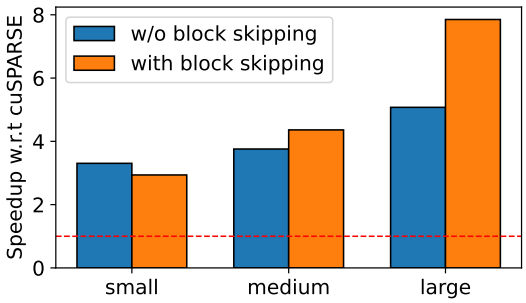


Figure 9. Average speedup of reordered matrices using Venom backend compared with cuSPARSE, with and without zero-tile skipping.

Scheduling under time constraints. When runtime is constrained, *all-strategy traversal* is often unsuitable because its failure cases can be time-consuming. Among heuristic schedules, *adaptive cycling* performs well across both easy and hard instances: on easy cases it adds little probing overhead and handles failures efficiently; on hard cases it attains high success with lower runtime than exhaustive traversal. Under very tight time budgets, however, even *adaptive cycling* may be more conservative than desired. If only one schedule must be chosen, we recommend the *expanded learned policy*, which dynamically blends single-strategy behavior with *adaptive cycling* and better meets stringent time limits than using *adaptive cycling* alone.

5.5 Effectiveness of Zero-tile skipping

As discussed in Section 4.6, we use a light-weight patch to skip all-zero metablocks to further speedup VENOM backend. Figure 9 reports the average speedup versus cuSPARSE baseline with and without block skipping. On the SuiteSparse *small*, *medium*, and *large* subsets, we skip on average 31%, 32%, and 39% of blocks, respectively; the corresponding speedups brought by the skipping are −10% (i.e., a slowdown), 9%, and 56%. The benefit grows with input size for two reasons: (i) larger matrices tend to be sparser, increasing the fraction of all-zero tiles; and (ii) larger problems drive

higher GPU utilization, so eliminating zero-tile work translates more directly into wall-clock gains. By contrast, for small inputs the GPU is underutilized, so the saved computation does not fully convert into runtime reduction.

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 [70, 71]. Since Huang and others' systematic exploration of the implications of GNN performance gaps [35], many studies have been devoted to bridging the gap [12, 34, 61, 62, 72].

SpMM on GPU. A large body of prior work has focused on improving SpMM performance on GPUs for GNN workloads. CSR-based approaches such as GE-SpMM [33] propose general-purpose GPU SpMM frameworks that improve efficiency through techniques including coalesced row caching and warp-level merging, and are widely adopted as framework-friendly baselines due to their compatibility with existing GNN systems. Another significant line of work explores accelerating SpMM by leveraging dense Tensor Cores. TC-GNN [62] enables sparse GNN aggregation on dense Tensor Cores by transforming sparse workloads into Tensor-Core-friendly representations and execution schedules, while DTC-SpMM [23] further analyzes inefficiencies in Tensor-Core-based SpMM and introduces optimizations such as more memory-efficient Tensor-Core-friendly formats, software pipelining, and sparsity-aware execution to improve utilization and reduce memory stalls. Acc-SpMM [75] advances this line of research by providing a library-level framework that combines data-affinity-aware reordering, compact sparse formats, adaptive workload partitioning, and high-throughput pipelining, achieving strong performance across a wide range of sparsity patterns and GPU architectures. Targeting NVIDIA Hopper GPUs, Voltrix [65] introduces architecture-specific optimizations including asynchronous data-loading pipelines, bit-level compression, warp-specialized persistent kernels, and bulk copy mechanisms to better overlap memory movement with Tensor Core computation. Meanwhile, Mehrabi et al. [43] propose learning sparse matrix row permutations to improve SpMM efficiency on GPUs. HR-SpMM [60] proposes a hybrid execution strategy that partitions rows based on sparsity and maps them to either CUDA cores or Tensor Cores to improve utilization under highly irregular row-length distributions.

Beyond the stand-alone SpMM kernels, several systems focus on accelerating end-to-end GNN training; HP-SpMM [22] introduces hybrid-parallel SpMM kernels that dynamically combine node- and edge-parallel execution across forward and backward passes, while MaxK-GNN [48] proposes a co-designed training framework with a novel MaxK nonlinearity, compressed feature representations, and customized sparse GPU kernels to achieve significant end-to-end speedups. Despite these advances, relatively few works directly exploit Sparse Tensor Cores for SpMM due to the strict N:M structured sparsity requirements. In contrast, this work focuses on a reordering algorithm that preserves graph characteristics while enforcing N:M sparsity, enabling efficient execution on Sparse Tensor Cores.

Table 11. Speedup of our work compared with other SpMM libraries across different input sizes and column count of the dense multiplier (N). *Avg* denotes the mean speedup over all Ns evaluated, i.e., $N \in \{128, 256, 512, 1024, 2048\}$. (For sake of space, we include only the results of two cases (N=512, N=1024), and the average of all cases in the table.)

Library	Small			Medium			Large		
	512	1024	Avg	512	1024	Avg	512	1024	Avg
HP-SpMM	3.36	4.12	3.48	3.37	5.65	4.48	3.69	7.76	5.71
MaxK	4.56	11.87	7.02	2.64	6.75	4.21	1.19	3.82	2.03
Acc-SpMM	2.43	2.82	2.41	1.55	1.71	1.52	0.79	1.15	0.97
GE-SpMM	1.99	2.35	2.04	2.72	2.05	2.10	2.34	1.39	1.84

While Tensor Core based methods gain popularity recently, a common limitation is their reliance on dense or semi-dense representations, which can increase memory footprint and bandwidth pressure by tens to hundreds of times compared to CSR, posing a critical scalability challenge for large graphs. In the meantime, few prior works directly exploit Sparse Tensor Cores for SpMM due to the strict N:M structured sparsity requirements. As shown in Table 11, our approach achieves up to 7.76× speedup in SpMM kernels; among existing methods, only Acc-SpMM outperforms our approach on large matrices, especially when the dense multiplier has a small number of columns. However, unlike Acc-SpMM, our method is orthogonal to backend kernel and library optimizations, leaving substantial room for further performance improvements.

Graph Reordering. Graph reordering has been commonly used as a preprocessing technique for graph computation to improve data locality and other purposes [7, 8, 17, 21, 36, 44, 50, 56]. 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 [49] and MiLogA [16, 54], 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 [63]. For scalability, coarser-grained reordering methods like degree-based sorting [21, 37, 41, 74] and partitioning [6, 10, 39, 69, 78] 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. [79] 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. [58] present DominoSearch for iterative N:M sparsity determination in DNN training using magnitude-based pruning. Pool et al. [52] 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. [38] develop "structure decay" for iterative N:M pruning, using a mask decay method to improve training stability. Chen et al. [15] 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 [47] 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 [73]. 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

In this work, we develop a graph reordering framework that enables GNN workloads to exploit modern GPU Sparse Tensor Cores. It reshapes adjacency matrices into the V:N:M pattern by coordinating bit-level primitives in an iterative two-stage pipeline: a Hamming-distance–guided ordering followed by greedy updates. We further combine multiple strategies to broaden the search space and employ a global scheduler and select effective policies under varying constraints. Experiments show 3.44–7.25× speedups for the core SPMM kernel, yielding substantial end-to-end gains for GNN computation.

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. USDA NIFA P24-001771, NIH 1R01HD108473-01, and NSF OAC-2417850. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of UDA, NIH, or NSF. This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231.

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. doi: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. doi:10.1145/3577193.3593725

[13] Jou-An Chen, Hsin-Hsuan Sung, Ruifeng Zhang, Ang Li, and Xipeng Shen. 2025. Accelerating GNNs on GPU Sparse Tensor Cores through N: M Sparsity-Oriented Graph Reordering. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 16–28.

[14] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[15] 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.

[16] 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.

[17] 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

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

[19] 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. doi:10.1145/3322125

[20] 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.

[21] 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.

[22] Ruibo Fan, Wei Wang, and Xiaowen Chu. 2023. Fast sparse gpu kernels for accelerated training of graph neural networks. In *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 501–511.

[23] 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. doi:10.1145/3620666.3651378

[24] 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.

[25] Lanting Fang, Yulian Yang, Kai Wang, Shanshan Feng, Kaiyu Feng, Jie Gui, Shuliang Wang, and Yew-Soon Ong. 2024. SIG: Efficient Self-Interpretable Graph Neural Network for Continuous-time Dynamic Graphs. *arXiv preprint arXiv:2405.19062* (2024).

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

[27] 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.

[28] 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.

[29] 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]

[30] 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).

[31] Mert Hidayetoğlu, Carl Pearson, Vikram Sharma Mailthody, Eiman Ebrahimi, Jinjun Xiong, Rakesh Nagi, and Wen-mei Hwu. 2020. At-scale sparse deep neural network inference with efficient gpu implementation. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[32] 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.

[33] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Gespmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[34] 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.

[35] 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

[36] 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.

[37] 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.

[38] 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).

[39] 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.

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

[41] 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.

[42] Bin Lin, Ningxin Zheng, Lei Wang, Shijie Cao, Lingxiao Ma, Quanlu Zhang, Yi Zhu, Ting Cao, Jilong Xue, Yuqing Yang, et al. 2023. Efficient gpu kernels for n: M-sparse weights in deep learning. *Proceedings of Machine Learning and Systems* 5 (2023), 513–525.

[43] Atefeh Mehrabi, Donghyuk Lee, Niladrish Chatterjee, Daniel J Sorin, Benjamin C Lee, and Mike O'Connor. 2021. Learning sparse matrix row permutations for efficient spmm on gpu architectures. In *2021*

IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). IEEE, 48–58.

[44] 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. doi:10.1023/A:1011119519789

[45] 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).

[46] NVIDIA. 2021. *NVIDIA Technical Blog - Blocked ELL Format.* https://developer.nvidia.com/blog/accelerating-matrix-multiplication-with-block-sparse-format-and-nvidia-tensor-cores/

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

[48] Hongwu Peng, Xi Xie, Kaustubh Shivdikar, Md Amit Hasan, Jiahui Zhao, Shaoyi Huang, Omer Khan, David Kaeli, and Caiwen Ding. 2024. Maxk-gnn: Extremely fast gpu kernel design for accelerating graph neural networks training. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 683–698.

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

[50] 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.

[51] 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*.

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

[53] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637* (2020).

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

[55] 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

[56] 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.

[57] 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.

[58] 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.

[59] 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).

[60] Qi Wang, Yaobin Wang, Yi Luo, Rong Luo, and Pingping Tang. 2025. Hr-spmm: Adaptive row partitioning and hybrid kernel design for sparse matrix multiplication. In *Proceedings of the 39th ACM International Conference on Supercomputing*. 161–172.

[61] 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)*.

[62] 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

[63] 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.

[64] 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.

[65] Yaqi Xia, Weihu Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. 2025. Voltrix: Sparse {Matrix-Matrix} Multiplication on Tensor Cores with Asynchronous and Balanced Kernel Optimization. In *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 699–714.

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

[67] 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. doi:10.1007/s00778-023-00790-4

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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)*.

[73] 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. doi:10.1145/3673038.3673108

[74] 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.

[75] Haisha Zhao, San Li, Jiaheng Wang, Chunbao Zhou, Jue Wang, Zhikuang Xin, Shunde Li, Zhiqiang Liang, Zhijie Pan, Fang Liu, et al. 2025. Acc-SpMM: Accelerating General-purpose Sparse Matrix-Matrix

Multiplication with GPU Tensor Cores. In *Proceedings of the 30th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 326–338.

[76] 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.

[77] 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.

[78] 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.

[79] 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

[80] 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.