Fast Sparse GPU Kernels for Accelerated Training of Graph Neural Networks

Ruibo Fan[†], Wei Wang[*], Xiaowen Chu[†*]
[†]The Hong Kong University of Science and Technology (Guangzhou)
[*]The Hong Kong University of Science and Technology
[†]rfan404@connect.hkust-gz.edu.cn, [*]weiwa@cse.ust.hk, [†*]xwchu@ust.hk

***Abstract*—Graph Neural Networks (GNNs) are gaining huge traction recently as they achieve state-of-the-art performance on various graph-related problems. GNN training typically follows the standard Message Passing Paradigm, in which SpMM and SDDMM are the two essential sparse kernels. However, existing sparse GPU kernels are inefficient and may suffer from load imbalance, dynamics in GNN computing, poor memory efficiency, and tail effect. We propose two new kernels, Hybrid-Parallel SpMM (HP-SpMM) and Hybrid-Parallel SDDMM (HP-SDDMM), that efficiently perform SpMM and SDDMM on GPUs with a unified hybrid parallel strategy of mixing nodes and edges. In view of the emerging graph-sampling training, we design the Dynamic Task Partition (DTP) method to minimize the tail effect by exposing sufficient parallelism. We further devise the Hierarchical Vectorized Memory Access scheme to achieve aligned global memory accesses and enable vectorized instructions for improved memory efficiency. We also propose to enhance data locality by reordering the graphs with the Graph Clustering method. Experiments on extensive sparse matrices collected from real GNN applications demonstrate that our kernels achieve significant performance improvements over state-of-the-art implementations. We implement our sparse kernels in popular GNN frameworks and use them to train various GNN models, including the GCN model in full-graph mode and the GraphSAINT model in graph-sampling mode. Evaluation results show that our kernels can accelerate GNN training by up to 1.72×.**

I. Introduction

Graph Neural Networks (GNNs) are gaining increasing popularity as they achieve state-of-the-art performance in many graph-related problems, such as node classification, link prediction, graph classification [1], etc. In GNNs, data are organized in a graph structure composed of nodes and edges. Each node (edge) is associated with a feature vector. Compared to traditional neural networks (NNs), GNN training is more complex because the graph and NN operations are interleaved.

GNN training usually follows the standard Message Passing Paradigm (MPP), which provides unifying support to various graph operations [2]–[4]. In MPP, each node updates its feature vector by aggregating features from its neighbors, and each edge updates its features by aggregating features from its source and destination nodes. More formally, let $G(V, E)$ be a self-looped graph[1] with a node set V and an edge set E. Let u and v be two neighboring nodes with a connecting edge $e(u, v) \in E$. Denote by $\mathbf{X}_u \in R^K$ the feature vector of node u and $\mathbf{X}_{uv} \in R^K$ the feature vector of edge $e(u, v)$, where K is the feature dimension. MPP performs the following node-wise and edge-wise operations:

$$\textbf{node-wise} : \mathbf{H}_v = \bigoplus_{u \in \mathcal{N}(v)} f_n \left(\mathbf{X}_u, \mathbf{X}_{uv} \right), \tag{1}$$

$$\textbf{edge-wise} : \mathbf{H}_{uv} = f_e \left(\mathbf{X}_u, \mathbf{X}_v, \mathbf{X}_{uv} \right), \tag{2}$$

where $\mathcal{N}(v)$ is the neighboring set of node v, operator $\bigoplus$ is the aggregation or reduction operation that generates a new feature vector of v, which varies in different GNN models (e.g., $\max$, $\min$, sum) [5], and f_n and f_e are the functions for node message calculation and edge feature update, respectively.

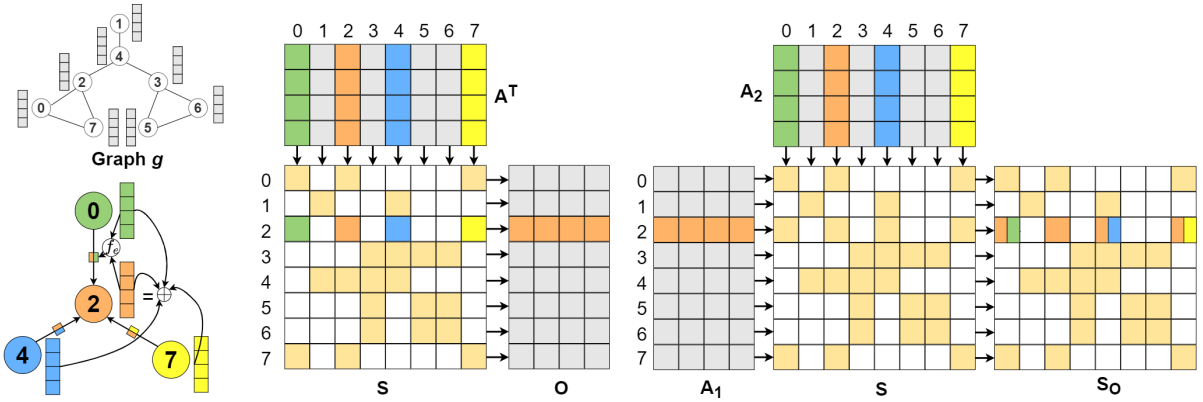
As shown in Fig. 1, MPP in the entire graph can be viewed as a general multiplication between the sparse adjacent matrices and the dense feature matrices. If $\bigoplus$ and f_e are sum, Eqs. (1) and (2) represent Sparse Matrix-Matrix Multiplication (SpMM) and Sampled Dense-Dense Matrix Multiplication (SDDMM), respectively (see Fig. 1). GNN frameworks [2], [6] implement the widely used graph convolution (GCN) layer with an SpMM operation followed by a fully-connected layer.

GNN frameworks support two training modes, *full-graph* (full-batch) and *graph-sampling* (mini-batch) training. In the full-graph mode, the graph remains unchanged during multiple training iterations. Compared with full-graph, the graph-sampling mode is dynamic since it uses the newly sampled graphs as the input in each training iteration. As GNN models are getting larger and more complex, accelerating GNN training by GPUs becomes increasingly important. One effective approach is to devise efficient core sparse GPU kernels, including SpMM and SDDMM, for accelerated training [3], [7]. However, existing sparse kernels [3], [7]–[9] are not well optimized for GNNs and may result in the following performance issues.

Load imbalance and poor data locality. In real-world applications, the degree of nodes in a graph varies dramatically. Several existing sparse kernels use a *node-parallel* execution strategy, which assigns the tasks of a node to a parallel execution unit in GPU [3], [7]–[10]. As the node degree has a large variance in a graph, node-parallel execution results in severe load imbalance. When the graph is highly irregular, the data locality can also be compromised [11].

Unable to handle the dynamics in GNN computing. Some works utilize preprocessing to improve the perfor-

Corresponding author: Xiaowen Chu
[1]For consistency with the matrices used for illustration.



(a) Message Passing Paradigm (b) SpMM (c) SDDMM

Fig. 1: Message Passing Paradigm and sparse matrix multiplications on graph g. (a) MPP of node 2 and edge $e(0,2)$. Node 2 updates its feature vector (orange) by taking the reduction result of its neighbors' features. Edge $e(0,2)$ updates its feature vector by taking function f_e with inputs of the features of the two connecting nodes. (b) An illustration of SpMM $O = SA$. SpMM is performed on the adjacent matrix S of graph g and the feature matrix A. (c) An illustration of SDDMM $S_O = (A_1 A_2) \odot S$. SDDMM is performed on the adjacent matrix S of graph g and the feature matrices A_1 and A_2. Row 2 of S is highlighted to illustrate the relationship between the MPP of node 2 and SpMM/SDDMM.

mance of sparse kernels [8]–[11]. [8], [10], [11] notice the load imbalance issue. They use preprocessing (i.e., sorting or binary search) and the special Compressed Sparse Row (CSR) format with additional arrays to achieve load balance. These works have the following shortcomings when applied to accelerating GNN computing. ❶ The dynamic property of GNNs is ignored. The preprocessing overhead can only be amortized in many rounds of execution. However, GNN inference and graph-sampling training are dynamic, and each round of execution is based on different graphs. Thus, it's hard to use these preprocess-based techniques to accelerate the graph-sampling training [7]. ❷ These works adopt the same granularity for different inputs and ignore the study on task partition granularity which is crucial for performance. Sparse matrices are all divided into tiles whose sizes are equal to the number of launched threads in the CUDA block in [8]. [11] separates long rows into tiles but left the tile size unstudied.

Inefficiency when loading sparse data. Keeping memory access aligned and coalesced is crucial to maximizing the global memory throughput. We evaluate and analyze the SpMM kernel in cuSPARSE [12] on multiple graph datasets using the Nsight Compute [13] and find that there are misaligned and uncoalesced memory accesses. Besides, it is difficult to enable vectorized memory access when fetching sparse data.

Tail effect and insufficient parallelism. The tail effect of GPU kernels occurs when the number of thread blocks is not well configured, which results in low GPU utilization [14]. The tail effect is insignificant when large sparse matrices are used (e.g., sparse kernels for scientific computing). However, GNN training uses small sparse matrices, causing a salient tail effect that significantly slows down the execution of GNN algorithms, especially when graph-sampling is used. This problem has received little attention in the literature.

To address the above issues, we design two new kernels, HP-SpMM and HP-SDDMM, that perform SpMM and SDDMM with a unified hybrid parallel strategy of mixing nodes and edges[2]. We make the following contributions:

- We devise a unified hybrid parallel strategy for SpMM and SDDMM. Our strategy assigns each GPU execution unit precisely the same amount of work, thereby achieving a balanced load.
- We develop two new techniques, Dynamic Task Partition (DTP) and Hierarchical Vectorized Memory Access (HVMA). DTP adaptively adjusts the task allocation to minimize the tail effect. HVMA, on the other hand, enables aligned, coalesced, and vectorized memory access.
- We propose Graph Clustering based Reordering (GCR), an effective approach based on the Louvain method that improves data locality by judiciously grouping nodes into communities.
- We implement our optimization approaches and evaluate their performance on extensive sparse matrices collected from the full-graph and graph-sampling training of various GNN models. On two platforms, significant performance improvement is achieved over available state-of-the-art sparse kernels. We also embed our sparse kernels into DGL and PyG frameworks to train popular GNN models, including GCN in full-graph mode and GraphSAINT [15] in graph-sampling mode and achieve up to 1.72× end-to-end training speedup.

II. Background and Related Work

In this section, we provide the background information. The notations used in this paper are listed in Table I.

[2]https://github.com/fan1997/HP-SpMM-SDDMM.git

TABLE I: Notations

Notation	Description
S	sparse input matrix of dimension $M \times N$
A	dense input matrix for SpMM of dimension $N \times K$
O	dense output matrix for SpMM of dimension $M \times K$
A_1	dense input matrix for SDDMM of dimension $M \times K$
A_2	dense input matrix for SDDMM of dimension $K \times N$
S_O	sparse output matrix for SDDMM of dimension $M \times N$
M	number of rows in S; number of destination nodes in graph
N	number of columns in S; number of source nodes in graph
K	number of columns in A, A_1, A_2 and O; the dimension of the feature vector
NNZ	number of non-zero elements in S; number of edges in graph

Training methods of GNN models. Current GNN frameworks support two classes of training algorithms: full-batch (*full-graph*) training and mini-batch (*graph-sampling*) training. Full-batch training loads the entire graph into GPUs for training. Limited by the GPU memory capacity, full-graph training cannot handle massive graphs. Graph-sampling training addresses this problem. It samples a number of subgraphs from the original graph to construct a mini-batch and use it as the input in each iteration. Graph-sampling training is widely supported in popular GNN frameworks. SpMM and SDDMM are the two basic operations in the two training modes. In the full-graph mode, both operations work with the adjacent matrix of the original full graph, whereas in the graph-sampling mode, they work with the adjacent matrices of the generated subgraphs. Each training iteration's input graphs are different, making the graph-sampling mode more dynamic than full-graph training.

Sparse matrix representations in GNN frameworks. In GNNs, the sparse matrix refers to the adjacent matrix of the graph. Sparse matrices can be stored in several formats in the current GNN frameworks. As shown in Fig. 2(b), the CSR format represents a graph with three arrays: $RowOffset$, $ColInd$, and $Value$. The $Value$ array contains all the non-zero elements in the sparse matrix; $RowOffset[i]$ contains the $Value$ index of the first element of the i^{th} row; $ColInd[i]$ contains the column index of element i. Compared to CSR, the COO format is simpler. As shown in Fig. 2(c), it represents a graph in three arrays: $RowInd$, $ColInd$, and $Value$. $RowInd[i]$ and $ColInd[i]$ contain the row and column index of the i^{th} non-zero element in the sparse matrix, respectively. As shown in Fig. 2(d), GNN frameworks also support a special hybrid CSR/COO format which decodes the CSR's compressed row index array into a complete one. We note that current GNN frameworks store the sampled subgraphs in hybrid CSR/COO format [2], [6].

To store the sparse matrix S, the CSR format requires $M+1+2\times NNZ$ elements, while COO and hybrid CSR/COO require $3 \times NNZ$. It is feasible to alleviate load imbalance based on the CSR format, but preprocessing (e.g., sorting and binary search) makes these works hard to be utilized for GNN computing [7]. [16] points out that the COO and hybrid CSR/COO help achieve more direct load balancing but has a major disadvantage that prevents it from being widely adopted; that is, it requires more memory than CSR. We implement kernels based on the hybrid CSR/COO format due to the following three observations. ❶ The sampled graphs are directly stored in this format in GNN frameworks; thus, no format conversion is needed. ❷ Most of the memory required for GNN training comes from each layer's feature matrix ($M\times K$), and the sparse matrix only needs to be stored once. This largely masks the disadvantages of this format (i.e., it requires larger storage space compared to CSR). ❸ Compared with CSR-based load-balanced methods (with preprocessing), we are able to develop preprocessing-free kernels that are suitable for GNN computations with high real-time requirements (the dynamic graph-sampling and inference mode).

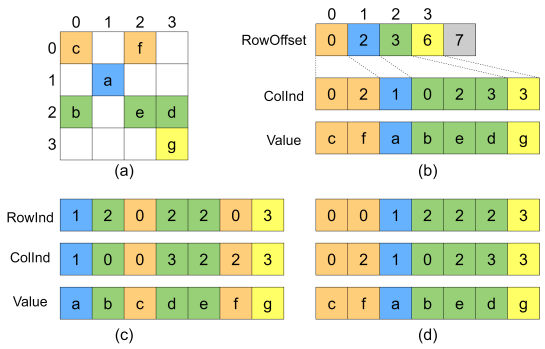


Fig. 2: Different representations of a sparse matrix. (a) The original sparse matrix. (b) The CSR representation. (c) The COO representation. (d) The hybrid CSR/COO representation decodes the CSR's compressed row index array into a complete one.

SpMM and SDDMM. SpMM implements the following computation: $O = SA$, where S, A, and O denote the sparse matrix, dense matrix, and output matrix, respectively. SDDMM computes $S_O = (A_1A_2) \odot S$, where A_1 and A_2 are two dense matrices, S is the input sparse matrix, S_O is the output sparse matrix, and $\odot$ is the Hadamard product. *NNZ* is the number of non-zero elements in S. Fig. 1 shows a conceptual view of SpMM and SDDMM. Algo. 1 and Algo. 2 describe the computations of sequential SpMM and SDDMM based on the hybrid CSR/COO format.

Algorithm 1 Sequential SpMM

Input: Hybrid CSR/COO S[M][N], A[N][K]
Output: O[M][K]
1: **for** $i \leftarrow 0$ **to** NNZ **step** 1 **do**
2: $r \leftarrow$ S.RowInd[i], $c \leftarrow$ S.ColInd[i], $v \leftarrow$ S.Value[i]
3: **for** $k \leftarrow 0$ **to** K **step** 1 **do**
4: O[r][k] $+= v \times$ A[c][k]
5: **end for**
6: **end for**

There are many efforts to optimize the two key kernels, SpMM and SDDMM. Nvidia cuSPARSE library [12] provides

Algorithm 2 Sequential SDDMM

Input: Hybrid CSR/COO S[M][N], A_1[M][K], A_2[K][N]
Output: S_O[M][N]

```
for i ← to NNZ step 1 do
    r ← S.RowInd[i], c ← S.ColInd[i], v ← S.Value[i]
    for k ← to K step 1 do
        S_O[i] += A_1[r][k] × A_2[k][c]
    end for
    S_O[i] ← S_O[i] × v
end for
```

high-performance SpMM and SDDMM kernels (not open-source). It supports three storage formats (CSR, COO, and Blocked-Ellpack) for SpMM while CSR for SDDMM. It designs several algorithms with different performances for SpMM and leaves the choice to the users. Yang *et al.* [8] introduce *row-split* and *merge-path* algorithms to SpMM (in GraphBLAST) based on the CSR format to hide global memory latency. Huang *et al.* [7] propose *GE-SpMM*, which targets at GNNs and optimizes the memory efficiency of SpMM by reusing sparse data.

Some works implement these two kernels based on special input formats not currently supported by GNN frameworks. Hong *et al.* [9] propose *ASpT*, which uses an adaptive tiling technique to reorder and partition sparse matrices into dense (CSR) and sparse parts (DCSR). Shared memory can be utilized with the dense part. Shi *et al.* [17] introduce grouped COO format to improve data reuse. We implement our kernels with the hybrid CSR/COO format, which is widely used in GNN frameworks.

Some efforts require preprocessing and additional arrays to alleviate load imbalance. *Merge-path* in [8] utilizes binary search as preprocessing to partition the original sparse matrices and use an additional array to store the row indices of each partition. Huang *et al.* [11] introduce neighbor grouping to partition the long rows in the sparse matrix into several short tiles. Gale *et al.* [10] propose *Sputnik*, which introduces 1-dimensional tiling and reverse offset memory alignment for SpMM and SDDMM targeted at matrices with less sparsity in sparse deep learning. *Sputnik* sorts the rows in the sparse matrix by length to alleviate load imbalance. An additional array is used to store the sorted result. The preprocessing time can be several or dozens of times the actual SpMM and SDDMM execution time, which is difficult to be amortized in dynamic GNN computing [7]. Our kernels can achieve load balancing without preprocessing, which makes them suitable for GNN computing. Wang [18] proposes *SparseRT* targeted at sparse DNN inference. Load balancing is achieved by partitioning the sparse weight matrix into 2D tiles. Load balancing and autotuning are performed at compile time as the sparse weight matrices are known at that time. However, sparse matrices are not known at compile time due to the dynamics in GNN computing, making these compilation techniques hard to utilize.

Several works [19], [20] focus on optimizing distributed-memory SpMM by reducing processor-to-processor communication. Separately, introducing Tensor Cores to low-precision SpMM and SDDMM has received attention recently [21]. *FusedMM* fuses these two kernels to accelerate GNNs [22].

Load imbalance is a significant performance issue in parallel computing. Different from SpMM, two sparse input matrices are multiplied to output a sparse matrix in Sparse General Matrix-Matrix Multiplication (SpGEMM). Several works group the rows of a similar amount of computations together to balance the load [23]–[25]. Triangle Counting counts intersections of source and destination nodes' neighbors for every edge in the graph. [26] [27] introduce Logarithmic Radix Binning (LRB) that groups edges with similar workloads into bins. Threads are dynamically allocated according to the amount of tasks in the bins. Although these methods are very insightful, they can not be introduced to SpMM directly because the computation modes are quite different.

III. Methodology

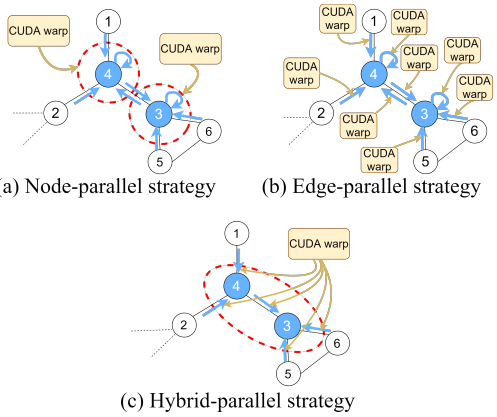
In this section, we present the design of HP-SpMM and HP-SDDMM. To address the load imbalance issue, we propose a hybrid-parallel strategy based on the hybrid CSR/COO format. We also propose optimization techniques to load data efficiently and improve the data locality.

A. Hybrid-Parallel Strategy

There are two parallel strategies to perform the MPP of the entire graph. The first is node-parallelism. As shown in Fig. 3(a), it assigns the MPP of a node to a specific execution unit. An alternative strategy is edge-parallelism. From the perspective of graph operations, one execution unit is used to complete the acquisition and computation of one adjacent node feature, as shown in Fig. 3(b). The two parallel strategies have different advantages and drawbacks.

Node-parallel strategy is coarse-grained and often not load-balanced. According to the previous analysis, the main reason is that the node degree in the graph is unevenly distributed, creating highly imbalanced loads on the execution units. Also, for those graphs with a small number of nodes but a large number of edges, the node-parallel strategy leads to fewer allocated execution units, making it difficult to fully utilize the GPU's parallelization capability. In contrast, the edge-parallel strategy is fine-grained and load-balanced. However, the memory efficiency of both reading and writing can be relatively low. As each execution unit loads an edge separately, more memory transactions are required to load the entire graph. Besides, the reduction of neighboring nodes' features can only be completed in writing the output feature matrix in global memory. Compared with node-parallel, this brings more global memory write conflicts.

To address the drawbacks of the two parallel strategies, we propose a new hybrid-parallel strategy for our sparse kernels. As illustrated in Fig. 3(c), with hybrid parallelism, the one-to-one mapping between nodes and execution units is not needed. Instead, the computing tasks of a node are dispatched to and completed by different execution units. Referring to Fig. 3(c),



(a) Node-parallel strategy (b) Edge-parallel strategy

(c) Hybrid-parallel strategy

Fig. 3: Overview of node-parallel, edge-parallel, and our proposed hybrid-parallel strategy from the perspective of graphs. The blue arrow represents a calculation (feature aggregation). Nodes 3 and 4's computations are assigned to two CUDA warps with node-parallelism. With edge-parallelism, feature aggregation of each edge is assigned to a CUDA warp. With hybrid-parallelism, we assign part of nodes 3 and 4's computations to a CUDA warp.

a CUDA warp is responsible for all the computing tasks of node 3 and part of node 4. Each execution unit's tasks are set to be almost the same, leading to more balanced loads. Memory transactions and conflicts are also reduced compared to the edge-parallel strategy. Following this idea, we next describe how to apply the hybrid-parallel strategy efficiently from the perspective of matrices and sparse kernels.

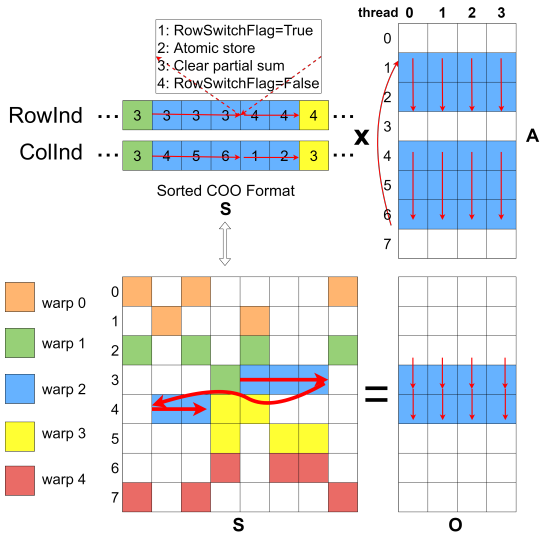


Fig. 4: HP-SpMM: Hybrid-parallel SpMM. We assign the same amount (5) of nonzero elements in S to five warps. Warp 2 with four threads first reads and accumulates rows 4, 5, and 6 from A and writes the accumulations to row 3 in O. Warp 2 then clears the accumulations and changes to read and accumulates row 1 and 2 from A and write the accumulations to row 4 in O. Other warps perform similar computations.

1) The design of HP-SpMM (Hybrid-Parallel SpMM): Fig. 4 shows the design of HP-SpMM. The pseudo-code of our method is given in Algo. 3. We assign the same amount (*NnzPerWarp*) of non-zero elements from sparse matrix S to CUDA warps. In each iteration, the threads in a warp cooperatively load a tile of data from *RowInd*, *ColInd*, and *Value* of S in hybrid CSR/COO format to shared memory tile (line 7). The threads in a warp read the tile one by one to get the indices that need to be accessed in matrix A (line 9). After they get one index, all the threads in a warp get and access the corresponding offset in matrix A (line 10) and accumulate into the register *res* (line 19), which stores the partial result. The tile-loading and accumulating processes continue until the row for which the warp is responsible changes. A row-switch procedure is crucial in our design. Each thread maintains a variable, *RowSwitchFlag*, to track the status of row switching. When the row processed by the warp does not change, the *RowSwitchFlag* remains *false*, and the thread continues to write the accumulation result into register *res*. When the row changes, each thread writes the partial sum *res* into the corresponding offset of the matrix O in global memory (line 15). *Res* is cleared and set to zero to prepare for the accumulation of the new row (line 16).

Algorithm 3 HP-SpMM Algorithm

Input: Hybrid CSR/COO S, A[], $NnzPerWarp$, K
Output: O[]

1: **Initialize:** $warp_{id}$, S_{offset}, A_{offset}, O_{offset}, $res \leftarrow 0$;
2: $warp_{start} \leftarrow warp_{id} \times NnzPerWarp$;
3: $warp_{end} \leftarrow warp_{start} + NnzPerWarp$;
4: __shared__ S_tile[3][32]; ▷ row, column id, and value.
5: A_tile[32];
6: **for** $i \leftarrow warp_{start}$ **to** $warp_{end}$ **step** 32 **do**
7: S_tile $\leftarrow$ LoadSparse($S_{offset} + i$);
8: **for** $j \leftarrow 0$ **to** 32 **step** 1 **do**
9: col_id $\leftarrow$ S_tile[1][j];
10: A_tile[j] $\leftarrow$ Load(A_{offset} + col_id$\times$ K);
11: **end for**
12: **for** $j \leftarrow 0$ **to** 32 **step** 1 **do**
13: ▷ row-switch procedure.
14: **if** $RowSwitchFlag == true$ **then**
15: AtomicStore(O_{offset} + S_tile[0][j] $\times$ K, res);
16: $res \leftarrow 0$;
17: $RowSwitchFlag \leftarrow false$;
18: **end if**
19: res += S_tile[2][j] $\times$ A_tile[j];
20: **end for**
21: **end for**
22: AtomicStore(O_{offset} + S_tile[0][31] $\times$ K, res);

With a proper *NnzPerWarp*, the computing load of HP-SpMM is relatively balanced. However, since HP-SpMM breaks the task dependency of the same row, it inevitably brings more global memory writes. Thus, we implement our kernel based on the hybrid CSR/COO format. It can ensure that the non-zero elements that each warp is responsible for

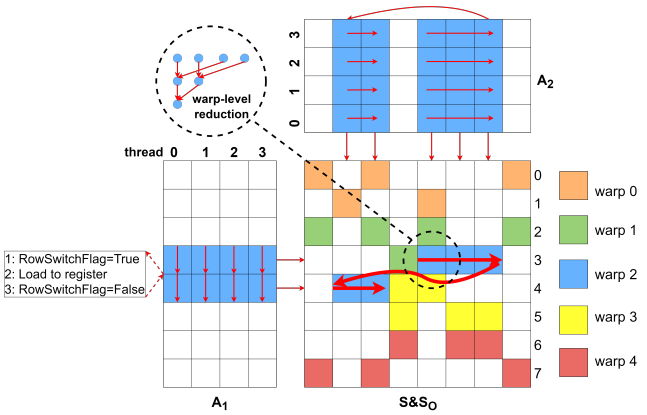


Fig. 5: HP-SDDMM: hybrid-parallel SDDMM. Warp 2 with four threads first reads row 3 in A_1 and maintains it in registers. Warp 2 then reads column 4 in A_2. Warp-level reduction is performed, and the result is written to $S_O(3,4)$. After processing the other two blue elements in row 3, warp 2 switches to row 4 and performs similar computations.

Algorithm 4 HP-SDDMM Algorithm

Input: Hybrid CSR/COO S, A_1[], $A_2{}^T$[], $NnzPerWarp$, K
Output: S_O.Value[]

1: **Initialize:** $lane_{id}$, S_{offset}, $A1_{offset}$, $A2_{offset}$;
2: Define variables; ▷ lines 2-4 of Algorithm 3.
3: **for** $i \leftarrow warp_{start}$ **to** $warp_{end}$ **step** 32 **do**
4: S_tile $\leftarrow$ LoadSparse($S_{offset} + i$);
5: **for** $j \leftarrow 0$ **to** 32 **step** 1 **do**
6: $v2 \leftarrow$ Load($A2_{offset}$ + S_tile[1][j] $\times$ K);
7: ▷ row-switch procedure.
8: **if** $RowSwitchFlag == true$ **then**
9: $v1 \leftarrow$ Load($A1_{offset}$ + S_tile[0][j] $\times$ K);
10: $RowSwitchFlag \leftarrow false$;
11: **end if**
12: $res \leftarrow$ WarpReduce($v1 \times v2$);
13: **if** $lane_{id} == 0$ **then**
14: AtomicStore(S_O.Value[$i + j$], res $\times$ S_tile[2][j]);
15: **end if**
16: **end for**
17: **end for**

belong to the same row as far as possible while achieving load balancing to reduce the number of global memory accesses. Due to the irregularity of the sparse matrix, different warps may perform a different number of row-switch procedures. The slight difference in the number of global memory writes is well hidden in many computations.

2) The design of HP-SDDMM (Hybrid-Parallel SDDMM): The diagram of HP-SDDMM is shown in Fig. 5. The warp computes each non-zero output element in its task in turn. However, to improve the efficiency of loading sparse data, threads in the warp load a tile of the sparse matrix S to the shared memory tile cooperatively (line 4) and read them one by one from the shared memory instead of the global memory (line 6). Different from HP-SpMM, each thread calculates two corresponding offsets and accesses both matrices A_1 and A_2 to load two elements into its registers (lines 6 and 9). Then, each thread multiplies the two elements, and warp-level reduction is performed to sum all values of the threads in the warp (line 12). In HP-SpMM, we track the status of row switching to avoid redundant global memory writes as much as possible. However, for HP-SDDMM, we track to reduce global memory reads. If *RowSwitchFlag* remains false, the warp keeps the value accessed from matrix A_1 in registers and reuses them. Only if the *RowSwitchFlag* turns to be true, the warp loads a new row from matrix A_1. Through the row-switch procedure, the data in matrix A_1 is well reused, and redundant data transactions are saved.

B. Dynamic Task Partition and Hierarchical Vectorized Memory Access

With the basic implementations of HP-SpMM and HP-SDDMM, we achieve more balanced loads while competitive memory efficiency is maintained. However, there are still two challenges: 1) Our sparse kernels may still suffer from tail effects and insufficient parallelism due to the lack of careful consideration of task division. More specifically, we need to control the granularity of tasks scientifically. 2) We are not approaching the limit of GPU memory performance. Our hybrid-parallel strategy is very flexible and provides a new way to solve these problems. We can optimize the key parameter *NnzPerWarp* to control the task granularity and achieve better memory efficiency. We propose two techniques, Dynamic Task Partition (DTP) and Hierarchical Vectorized Memory Access (HVMA), to further improve the performance of our sparse kernel cooperatively.

1) Dynamic Task Partition: The smaller the value of *NnzPerWarp*, the closer it is to edge-parallel. If *NnzPerWarp* is set too large, there will be few active warps, and the parallelism is low. The allocated thread blocks are divided into waves and scheduled to SMs. The size of a wave equals the number of SMs multiplied by the number of active blocks of an SM. However, as shown in Fig. 6, the sparse kernels in GNN frameworks work on plenty of small graphs. Because the number of thread blocks is always related to the graph scale, existing sparse kernels cannot generate enough full waves of blocks. The last wave is commonly smaller and under-utilizes the GPU, as is shown in the left part of Fig. 6. Generally, *NnzPerWarp* is calculated using equation $NnzPerWarp = NNZ/M$. However, this method makes our sparse kernels fall into the same pitfall as CSR-based kernels. With DTP, we can adjust the value of *NnzPerWarp*. If the number of nodes in the graph is significantly less than the number of edges, we reduce the value of *NnzPerWarp* to ensure sufficient active warps. To better quantitatively determine the value of *NnzPerWarp*, we give a preliminary range constraint of *NnzPerWarp* through calculation. We first calculate the number of active blocks of

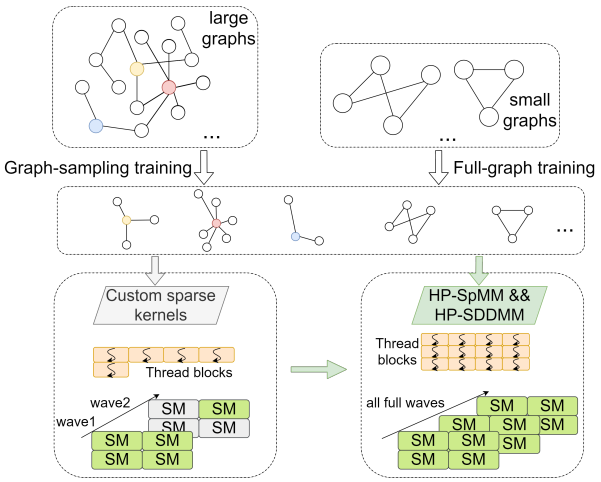


Fig. 6: Dynamic Task Partition technique for minimizing tail effect and exposing sufficient parallelism.

each SM, *ActiveblocksPerSM* can be computed as:

$$ActiveblocksPerSM = \min\{\frac{MaxWarpsPerSM}{WarpsPerBlock}, \frac{RegistersRerSM}{RegistersPerBlock}, \frac{SharedMemPerSM}{SharedMemPerBlock}\} \quad (3)$$

Then, the number of blocks of a full wave, *FullWaveSize* can be computed as:

$$FullWaveSize = NumSM \times ActiveblocksPerSM \quad (4)$$

We can obtain the preliminary numerical constraint of *NnzPerWarp*, as is shown in the following inequality:

$$\alpha \times FullWaveSize \leq \frac{NNZ}{NnzPerWarp \times WarpsPerBlock} \times \frac{K}{WarpSize \times VectorWidth} \quad (5)$$

where α is the scale factor and $VectorWidth$ is the vector width when loading the feature matrix. With the constraints in inEq. (5), our HP-SpMM and HP-SDDMM can generate sufficient waves to improve the utilization of GPU and avoid the tail effect, as shown in the right side of Fig. 6.

2) Hierarchical Vectorized Memory Access: DTP gives a preliminary quantitative constraint on *NnzPerWarp* from the perspective of tail effect and parallelism to improve the GPU utilization of HP-SpMM and HP-SDDMM. However, the value of *NnzPerWarp* cannot be finally determined, and memory efficiency is not taken into account.

Aligned and coalesced memory accesses provide the best global memory throughput. However, achieving both of them in loading sparse data is challenging. Coalesced accesses require all threads in a warp to access a contiguous chunk of memory. In our sparse kernels, threads in each warp cooperatively load a contiguous tile of sparse data; thus, coalesced accesses are achieved naturally. However, achieving aligned accesses is more complicated. When we load the sparse data

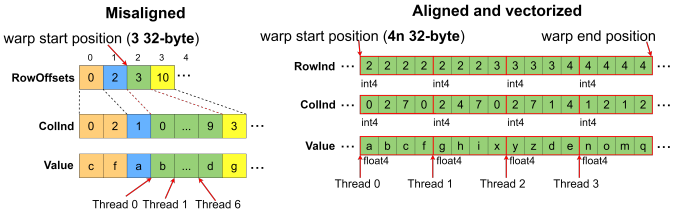


Fig. 7: Hierarchical Vectorization Memory Access approach for achieving aligned and vectorized memory accesses.

in global memory to registers, the data go through caches. Aligned accesses require the first addresses of global memory transactions to be multiples of the cache granularity (32 bytes for L2 cache and 128 bytes for L1 cache). Besides, vectorized memory instructions are essential for mitigating bandwidth bottlenecks and decreasing the number of instructions. However, it is also non-trivial to use these instructions in sparse kernels. Vectorized memory accesses require that the target values be aligned to the vector width (2 for float2 or 4 for float4). Both aligned and vectorized memory accesses require the alignment of the first addresses. As shown in the left side of Fig. 7, accesses within a CUDA warp begin at the start of a row in the sparse matrix. Due to the irregularity, these initial addresses have no alignment guarantees.

This issue still exists in our sparse kernels. By setting the proper value for *NnzPerWarp*, we can ensure alignment. We set up a candidate set for *NnzPerWarp*, $\{8, 32, 64, 128, 256, 512\}$. Combined with DTP, within the range specified in InEq. (5), we take the maximum value from the candidate and assign it to *NnzPerWarp*. If the optimal *NnzPerWarp* is 64, the implementation using int2 and float2 instructions will be called. If it is 128 or above, the int4 and float4 load instructions will be used instead, as shown on the right side of Fig. 7.

C. Graph Clustering based Reordering

The irregularity of graphs leads to poor data locality. The warps in a thread block may load features of the very different neighbor nodes from the global memory. The original layout of the sparse matrix may lead to a low L2 cache hit rate. A number of reordering methods for handling poor locality in sparse computation have been proposed [11] [28]. Locality-Sensitive Hashing (LSH) with Jaccard Similarity and Pair merging can cluster similar rows together. However, such algorithms are very time-consuming on larger graphs, and pair merging is difficult to execute in parallel on GPU.

We propose Graph Clustering based Reordering (GCR) method. We use GPU-based Louvain [29] [30] method to cluster similar nodes together before executing sparse kernels. We convert the origin format to reordered hybrid CSR/COO format according to the clustering result, as shown in Fig. 8. It is worth noting that we do not use the GCR technique in graph-sampling training. The runtime overhead accounts for a large proportion even though the efficiency is relatively high.

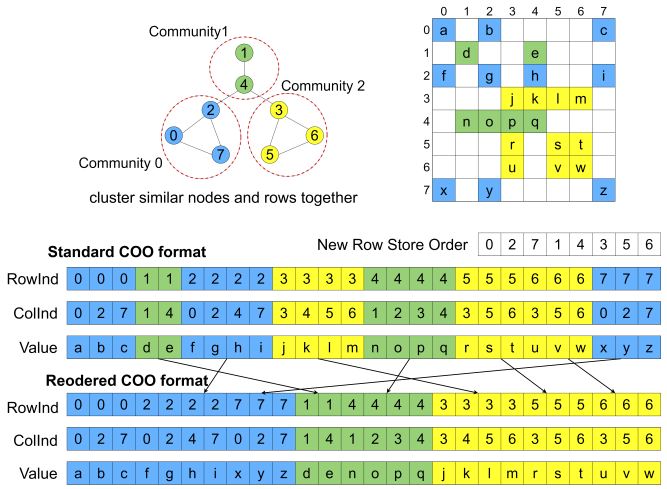


Fig. 8: Graph Clustering Based Reordering approach for enhancing data locality.

IV. Experimental Evaluation

In this section, we compare the performance with various implementations on extensive sparse matrices collected from popular GNN applications. End-to-end performance comparisons are carried out on several representative GNN models.

A. Experiment Setup

1) Datasets: To better evaluate the performance of our sparse kernels, we build and test on the following two datasets.

- **Full-graph dataset.** We select 19 representative graphs from DGL, OGB [31], GNN-benchmark [32] and GraphSAINT, which cover a wide range of scales. Table II shows the information.

TABLE II: The graphs in the full-graph dataset.

Source	Graph	#Nodes	#Edges
GraphSAINT	Flickr	89,250	989,006
	Yelp	716,847	13,954,819
	Amazon	1,598,960	264,339,468
DGL	CoraFull	19,793	146,635
	AIFB	7,262	44,298
	MUTAG	27,163	173,037
	BGS	94,806	656,226
	AM	881,680	7,141,524
	Reddit	232,965	114,848,857
OGB	arxiv	169,343	2.484.941
	proteins	132,534	79,255,038
	products	2,449,029	126,167,053
	collab	235,868	2,171,132
	ddi	4,267	2,140,089
	ppa	576,289	43,040,151
gnnbench	CoauthorCS	18,333	163,788
	AmazonCoBuyPhoto	7,650	245,812
	AmazonCoBuyComputer	13,752	505,474
	CoauthorPhysics	34,493	530,417

- **Graph-sampling dataset.** With the graph-sampling training mode, we train ten representative GNN models, including GraphSAGE, GraphSAINT, Care-gnn, etc. We collect 838 subgraphs to form graph-sampling dataset.

2) Baselines:

- **Vendor-provided cuSPARSE Library (not open-source).** We compare with the CSR-based SpMM (Algo 2 and 3), COO-based SpMM (Algo 4), and CSR-based SDDMM from cuSPARSE v11.8. We store the dense matrix in row-major and use FP32 precision. We use Nsight System [33] to profile the "cusparseSpMM" interface and find a partition kernel that helps CSR-based SpMM to achieve load balancing. We cannot exclude its time as it is an integral part of the "cusparseSpMM". Thus, we directly use cudaEventRecord to record the time before and after the API call and calculate the time difference to record the kernel execution time to the convention.
- **Open-source implementations. GE-SpMM [7]** is a state-of-art SpMM implementation targeted at GNNs. **Row-split [8]** is a classic SpMM implementation from GraphBLAST. **Merge-path [8]** achieves load balance by binary search preprocessing that partitions the sparse matrix into tiles. **ASpT [9]** partitions sparse matrices into dense and sparse parts to better utilize shared memory. **Sputnik [10]** introduces row-sorting to alleviate load imbalance. Neighbor grouping is used to partition long rows into small tiles to alleviate load imbalance in **Huang's method [11]**. **DGL-SDDMM [2]** is based on edge parallelism and shows competitive performance.

3) Environments: The experiments are mainly performed with two platforms: (1) NVIDIA **Tesla V100** (16G, Compute Capability 7.0) and Intel Xeon E5-2643 V4; (2) NVIDIA **Tesla A30** (24G, Compute Capability 8.0) and Intel(R) Xeon(R) Gold 6348.

We use NVCC 11.8 and GCC 8.5.0 with the -O3 flag to compile the codes. We run each experiment by 200 times and report the average results. All operations are performed with FP32. For a fair comparison[3], we directly record the actual kernel execution time and do not include format conversion time like CSRtoCOO or COOtoCSR for all kernels.

B. Kernel Benchmarks

1) Benchmarks in Full-graph dataset: We evaluate our sparse kernels' performance by benchmarking the 19 popular graphs. We set K to be 32, 64, and 128. The results on Tesla V100 are shown in Fig. 9. Table III summarizes the average speedups we achieved over the state-of-art methods. With the benefits of hybrid-parallel strategy and other optimization techniques, our kernels show significant advantages over different algorithms in cuSPARSE and other open-source kernels.

2) Benchmarks in Graph-sampling dataset: The performance comparison of our kernels and baselines on 838 collected subgraphs with V100 is shown in Fig. 10. We set K to be 64. To summarize, we list the average performance gain in Table III. Our implementations are more stable and outperform the baselines on most of the tests. To be noticed, we test

[3]In GNN frameworks, no runtime format conversion to our hybrid CSR/COO format is needed. Thus, excluding the format conversion time for all kernels is fair and appropriate.

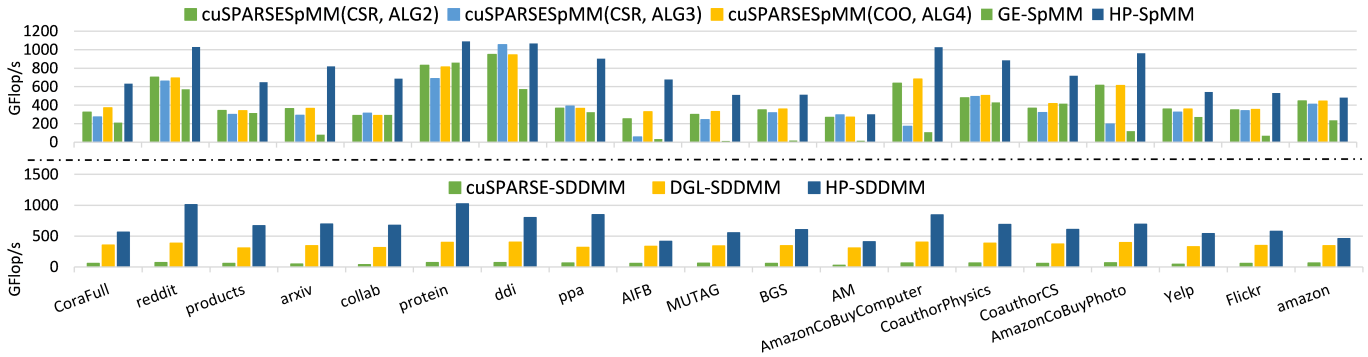


Fig. 9: Overall performance of sparse kernels in the full-graph dataset on Tesla V100 ($K = 64$).

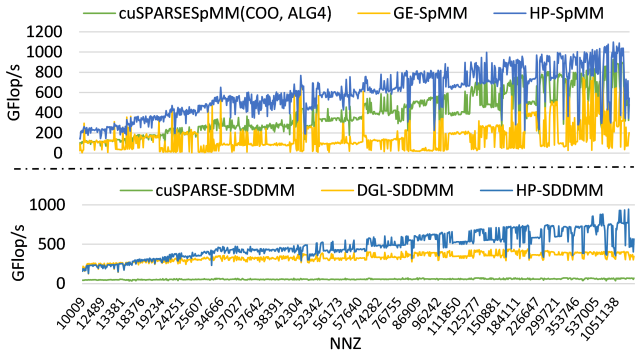


Fig. 10: Overall performance of sparse kernels in the graph-sampling dataset on Tesla V100 ($K = 64$).

without the GCR technique because the subgraphs are sampled at runtime in the graph-sampling training mode. The results show that our kernels can still benefit from DTP and HVMA techniques without reordering.

C. Comparison with preprocess-based and low-precision kernels

Some works require preprocessing to enhance data reuse [9] or achieve balanced loads [8], [10], [11]. We select three graphs of different scales and test their preprocessing and execution time with Tesla A30. We do not use GCR for our kernel in the test. Table IV shows the results. Preprocessing can take up to 43 times of the execution time, which indicates that they can not be applied well to graph-sampling training or inference. Our kernel can achieve competitive or better performance compared to these methods (without preprocessing time). We also compare with TC-GNN [21], which introduces TF32 Tensor Cores (TCs) to low-precision SpMM. We use the same RTX3090 GPU as in [21] to ensure that the computing power of TCs is consistent. For *Yelp* dataset, 8.28ms and 17.40ms are reported for HP-SpMM and TC-GNN, respectively. Nevertheless, introducing TCs is still promising. Low-level optimization techniques from [34] can potentially help to reach the limit of TCs.

D. Comparision with reordering techniques

Although reordering techniques can be used as offline processing in GNN computation, their efficiency is also important. Compared with the methods in [35] and [11], our proposed GCR can effectively improve efficiency. For the large dataset *protein*, 4.6s, 15.56s and over 120 minutes are reported for GCR, [35] and [11], respectively.

E. Ablation Study

We conduct an ablation study to understand the impact of different optimization techniques. We select four representative graphs and test the performance gain of each technique and their combinations. Fig. 11 shows the results. We find that the DTP and HVMA techniques are robust to various graphs. With DTP and HVMA, our kernels can outperform the baselines on most of the graphs. With only the GCR technique, our kernel can not gain significant performance benefits. Combined with DTP and HVMA, GCR can bring benefits to all graphs. However, the acceleration effect is sensitive to the characteristics of the graphs. For *AM* and *DDI*, GCR brings less than 10% performance improvement, while for *Yelp* and *PPA*, it brings about 40% benefits.

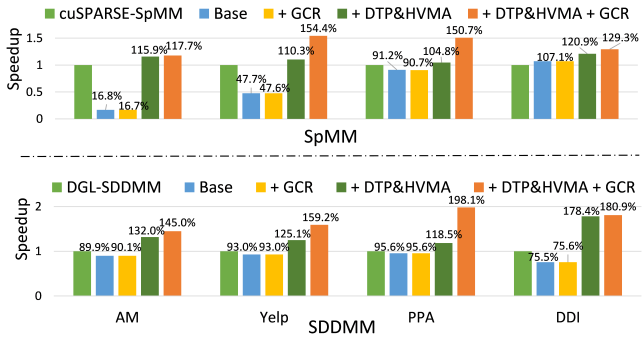


Fig. 11: Effects of proposed techniques on Tesla V100.

F. Sensitivity Analysis

We first study our kernels' sensitivity to node degree variance by comparing with GE-SpMM on some graphs with similar average node degrees but different variances. We

TABLE III: Summary of kernel benchmark results in the two datasets. The percentage is the ratio of graphs we can speed up to the total number (838 subgraphs).

		Tesla V100			Tesla A30		
		Full-graph	Graph-sampling		Full-graph	Graph-sampling	
	Baseline	Average speedup	Average speedup	Percentage	Average speedup	Average speedup	Percentage
SpMM	cuSPARSE(CSR, ALG2)	1.90×	2.06×	100%	2.53×	2.05×	100%
	cuSPARSE(CSR, ALG3)	2.75×	3.33×	98%	3.52×	3.40×	100%
	cuSPARSE(COO, ALG4)	1.82×	1.68×	100%	2.29×	1.65×	100%
	GE-SpMM	6.50×	8.71×	97.38%	8.45×	8.61×	98.93%
	Row-split	10.85×	10.09×	100%	13.33×	8.75×	100%
SDDMM	DGL-SDDMM	1.81×	1.31×	88.66%	2.08×	1.54×	99.17%
	cuSPARSE(CSR,DEFAULT)	10.90×	7.87×	100%	11.17×	10.49×	100%

TABLE IV: Comparision with preprocess-based kernels on Tesla A30. **Pre.** is the preprocessing time. **Exe.** is the kernel execution time. The time unit is milliseconds.

	AsPT		Sputnik		Merge-path		Huang's method		Ours
	Pre.	Exe.	Pre.	Exe.	Pre.	Exe.	Pre.	Exe.	Exe.
CoraFull	0.82	0.06	0.94	0.04	0.05	0.04	1.11	0.03	**0.02**
AM	16.47	3.35	37.98	14.44	0.59	2.82	73.00	2.57	**1.48**
Amazon	160.81	29.26	74.1	**19.65**	38.00	45.00	1058.84	24.84	22.35

choose GE-SpMM because it is open source and explicitly adopts the node-parallel strategy. Fig. 12 shows the result. Our kernels achieve higher speedups over GE-SpMM when the standard deviation of node degree gets larger. The Pearson's r is 0.90, indicating that our kernels are sensitive to node degree variances and our hybrid-parallel strategy is effective in dealing with the load imbalance issue.

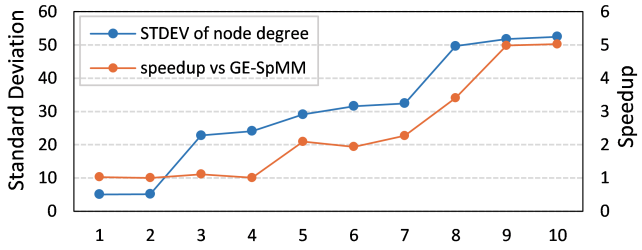


Fig. 12: Relationship between speedups and standard deviation of node degree. We select 10 graphs from the graph-sampling dataset with an average node degree between 21 and 25. We arrange them in ascending order according to the standard deviation of their node degrees. The larger the standard deviation, the more load imbalance.

We then use the *Flickr* dataset to test the kernels' sensitivity to K (i.e., the dimension of the feature vector) and show the results in Fig. 13. Although the throughput achieved by our kernels is basically unchanged with the increase of K, the performance of cuSPARSE and GE-SpMM is gradually increasing, which leads to the reduction of the relative speedups we have achieved. GE-SpMM proposes a data reuse technique to assign more elements from the dense matrices to each thread and reduce the reads of sparse data. We have tried to adopt this method in our kernels and found no improvements. This is mainly because the threads in our kernel consume more registers than GE-SpMM. If the data reuse method is introduced, we need to consume nearly twice as many registers as before, which makes the register file size a bottleneck. How to reduce the use of registers and improve performance when K gets very large is worthy of our further study.

Fig. 13: Sensitivity of performance to K on Tesla V100.

G. End-To-End GNN Training

We embed our kernels into DGL and PyG with the version, 0.9.1 and 2.1.0, respectively. We evaluate the performance in end-to-end GNN training on several GNN models with different hidden sizes and numbers of layers. For DGL, we train two models: an 8-layer GCN model on the *arxiv* dataset with full-graph mode and a 4-layer GraphSAINT model on the *amazon* dataset with graph-sampling mode. For PyG, we train a 4-layer GCN model on the *Flickr* dataset and a 3-layer GraphSAINT model on the *Yelp* dataset. For PyG with a version over 1.6.0, there are two modes to implement MPP, "Gather and Scatter" and "SparseTensor". The "SparseTensor" mode utilizes the kernels from the torch-sparse library and outperforms the other one, so we choose to compare our implementations with the "SparseTensor" mode. We utilize NSys [33] to profile the entire training process and record the total CUDA computation time from the report.

As shown in Table V, our kernels speed up the training of all the models. With our implementations integrated, DGL achieves up to 1.68× speedup over the original one with the cuSPARSE library. For PyG, our kernels bring up to 1.68× and 1.72× speedup for full-graph and graph-sampling training modes, respectively. However, with the increase in hidden sizes, the speedup ratio is getting lower. As discussed in

Section IV-F, the benefit of our kernel decreases as K gets larger due to the scarcity of registers.

TABLE V: Speedups of end-to-end GNN training brought by integrating our sparse kernels on Tesla V100.

	Model/ Dataset/ Training mode	Hidden size	w/o HP-SpMM (unit: sec)	w/ HP-SpMM (unit: sec)	Speedup
DGL	GCN/ arxiv/ full-graph	32	3.23	1.92	1.68×
		128	7.95	6.25	1.27×
		256	16.65	13.86	1.20×
	GraphSAINT/ Amazon/ graph-sampling	32	28.27	22.55	1.25×
		128	44.17	39.62	1.12×
		256	74.83	70.16	1.07×
PyG	GCN/ Flickr/ full-graph	32	2.34	1.40	1.68×
		128	4.66	3.21	1.45×
		256	8.64	6.634	1.30×
	GraphSAINT/ Yelp/ graph-sampling	32	0.70	0.40	1.72×
		128	1.30	0.87	1.49×
		256	2.15	1.64	1.31×

V. Conclusion

In this paper, we devised two sparse GPU kernels, HP-SpMM and HP-SDDMM, for accelerating GNN training. Our kernels perform SpMM and SDDMM with a unified hybrid parallel strategy that mixes nodes and edges. We also proposed Dynamic Task Partition and Hierarchical Vectorized Memory Access to minimize the tail effect and improve the efficiency of global memory accesses, respectively. We further proposed Graph Clustering-based Reordering to efficiently group similar nodes together for improved data locality. Extensive experiments on various collected sparse matrices and popular GNN models, trained in full and sampled graphs, demonstrate the effectiveness of our design. Our sparse kernels can be easily integrated into the existing GNN frameworks.

Acknowledgment

This work was partially supported by National Natural Science Foundation of China under Grant No. 62272122 and a Hong Kong RGC Research Impact Fund under Grant No. R6021-20.

References

[1] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[2] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang. Deep Graph Library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[3] Y. Hu, Z. Ye, M. Wang, J. Yu, D. Zheng, M. Li, Z. Zhang, Z. Zhang, and Y. Wang. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC*, pages 1–13. IEEE, 2020.

[4] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl. Neural message passing for quantum chemistry. In *ICML*, pages 1263–1272. PMLR, 2017.

[5] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, pages 1025–1035, 2017.

[6] M. Fey and J. E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[7] G. Huang, G. Dai, Y. Wang, and H. Yang. Ge-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *SC*, pages 1–12. IEEE, 2020.

[8] C. Yang, A. Buluç, and J. D. Owens. Design principles for sparse matrix multiplication on the GPU. In *Euro-Par*, pages 672–687. Springer, 2018.

[9] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *PPoPP*, pages 300–314, 2019.

[10] T. Gale, M. Zaharia, C. Young, and E. Elsen. Sparse GPU kernels for deep learning. In *SC*, pages 1–14. IEEE, 2020.

[11] K. Huang, J. Zhai, Z. Zheng, Y. Yi, and X. Shen. Understanding and bridging the gaps in current GNN performance optimizations. In *PPoPP*, pages 119–132, 2021.

[12] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. Cusparse library. In *GTC*, 2010.

[13] Nvidia. Nsight compute documentation. https://docs.nvidia.com/nsight-compute/index.html, 2022. Accessed: 2022-10-04.

[14] P. Micikevicius. GPU performance analysis and optimization. In *GTC*, 2012.

[15] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna. Graph-SAINT: Graph sampling based inductive learning method. In *ICLR*, 2019.

[16] D. Merrill and M. Garland. Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format. *ACM SIGPLAN Notices*, 51(8):1–2, 2016.

[17] S. Shi, Q. Wang, and X. Chu. Efficient sparse-dense matrix-matrix multiplication on GPUs using the customized sparse storage format. In *ICPADS*, pages 19–26. IEEE, 2020.

[18] Z. Wang. SparseRT: Accelerating unstructured sparsity on gpus for deep learning inference. In *PACT*, pages 31–42, 2020.

[19] P. Koanantakool, A. Azad, A. Buluç, D. Morozov, S.-Y. Oh, L. Oliker, and K. Yelick. Communication-avoiding parallel sparse-dense matrix-matrix multiplication. In *IPDPS*, pages 842–853. IEEE, 2016.

[20] V. Bharadwaj, A. Buluç, and J. Demmel. Distributed-memory sparse kernels for machine learning. In *IPDPS*, pages 47–58. IEEE, 2022.

[21] Y. Wang, B. Feng, and Y. Ding. TC-GNN: Accelerating sparse graph neural network computation via dense Tensor Core on GPUs. *arXiv preprint arXiv:2112.02052*, 2021.

[22] M. K. Rahman, M. H. Sujon, and A. Azad. FusedMM: A unified SDDMM-SpMM kernel for graph embedding and graph neural networks. In *IPDPS*, pages 256–266. IEEE, 2021.

[23] W. Liu and B. Vinter. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *IPDPS*, pages 370–381. IEEE, 2014.

[24] M. Parger, M. Winter, D. Mlakar, and M. Steinberger. Speck: Accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In *PPoPP*, pages 362–375, 2020.

[25] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. In *ICPPW*, pages 1–10, 2018.

[26] J. Fox, O. Green, K. Gabert, X. An, and D. A. Bader. Fast and adaptive list intersections on the GPU. In *HPEC*, pages 1–7. IEEE, 2018.

[27] O. Green, J. Fox, A. Watkins, A. Tripathy, K. Gabert, E. Kim, X. An, K. Aatish, and D. A. Bader. Logarithmic radix binning and vectorized triangle counting. In *HPEC*, pages 1–7. IEEE, 2018.

[28] P. Jiang, C. Hong, and G. Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *PPoPP*, pages 376–388, 2020.

[29] P. De Meo, E. Ferrara, G. Fiumara, and A. Provetti. Generalized Louvain method for community detection in large networks. In *ISDA*, pages 88–93. IEEE, 2011.

[30] S. Raschka, J. Patterson, and C. Nolet. Machine Learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *arXiv preprint arXiv:2002.04803*, 2020.

[31] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *NeurIPS*, 33:22118–22133, 2020.

[32] O. Shchur, M. Mumme, A. Bojchevski, and S. Günnemann. Pitfalls of graph neural network evaluation. *Relational Representation Learning Workshop, NeurIPS*, 2018.

[33] Nvidia. Nsight systems documentation. https://docs.nvidia.com/nsight-systems/index.html, 2022. Accessed: 2022-10-04.

[34] D. Yan, W. Wang, and X. Chu. Demystifying Tensor Cores to optimize half-precision matrix multiply. In *IPDPS*, pages 634–643. IEEE, 2020.

[35] Y. Wang, B. Feng, G. Li, S. Li, L. Deng, Y. Xie, and Y. Ding. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs. In *OSDI*, pages 515–531, 2021.