HYPERGEF: A FRAMEWORK ENABLING EFFICIENT FUSION FOR HYPERGRAPH NEURAL NETWORK ON GPUS

Zhongming Yu [1 2] **Guohao Dai** [3] **Shang Yang** [4] **Genghan Zhang** [4] **Hengrui Zhang** [5] **Feiwen Zhu** [2] **Jun Yang** [2]
Jishen Zhao [1] **Yu Wang** [4]

ABSTRACT

Hypergraph Neural Network (HyperGNN) is an emerging type of Graph Neural Networks (GNNs) that can utilize hyperedges to model high-order relationships among vertices. Current GNN frameworks fail to fuse two message-passing steps from vertices to hyperedges and hyperedges to vertices, leading to high latency and redundant memory consumption. The following challenges need to be solved for efficient fusion in HyperGNNs: **(1) Inefficient partition:** hardware-efficient and workload-balanced partitions are required for parallel workers to process two consecutive message passing steps after fusion. **(2) Workload-Agnostic Format:** current data formats like Compressed Sparse Row (CSR) fail to represent a two-step computation workload. **(3) Heavy writing conflicts:** partitioning leads to heavy writing conflicts when updating the same vertex.

To enable efficient fusion for HyperGNNs, we present ***HyperGef***. ***HyperGef*** proposes an edge-split partition scheme to achieve higher efficiency and better workload balancing. To represent the workload after fusion and partition, ***HyperGef*** introduces a novel fusion workload aware format. ***HyperGef*** also introduces a shared memory-aware grouping scheme to reduce writing conflicts. Extensive experiments demonstrate that our fused kernel outperforms the NVIDIA cuSPARSE kernel by 3.31×. By enabling efficient fusion for HyperGNNs, ***HyperGef*** achieves 2.25× to 3.99× end-to-end speedup on various HyperGNN models compared with state-of-the-art frameworks like DGL and PyG.

1 INTRODUCTION

Graph Neural Network (GNN) is one of the mainstream methods for machine learning tasks on graphs (Wu et al., 2021b). Among different GNN models (Kipf and Welling, 2016; Hamilton et al., 2017; Veličković et al., 2017), Hypergraph Neural Network (HyperGNN) is becoming a prospective trend because of its promising potential for improving algorithm accuracy (Feng et al., 2019; Bai et al., 2021), and the extensible applications in various domains, such as social network recommendation, paper co-citation, and drug discovery (Yu et al., 2021; Yadati et al., 2019; Cheng et al., 2022), shown in Figure 1(a).

HyperGNNs perform GNN computation on hypergraphs, which is capable of modeling high-order relationships among vertices. Figure 1(b) shows an example hypergraph. Vertices in the hypergraph are connected with hyperedges, and hyperedges can be connected to multiple vertices (rather than two vertices in conventional graphs). Thus, the GNN computation on hypergraphs can be processed with two message passing steps, *i.e.,* messages from vertices to hyperedges and hyperedges to vertices. We can use two sparse matrices in Figure 1(c) to represent these two message-passing steps, and two matrices are mutually transposed of each other.

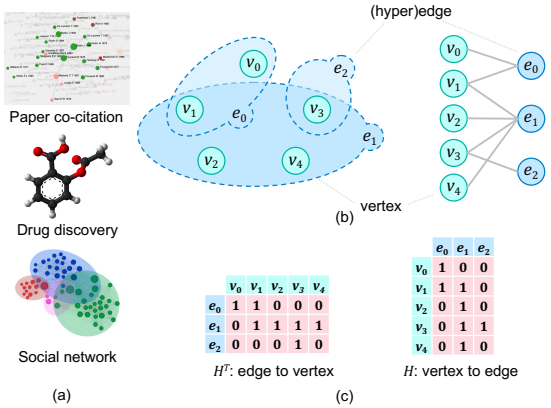

Figure 1. Hypergraph GNN examples. (a) Typical applications. (b) Hyperedges and vertices in a hypergraph. (c) The sparse matrix representation between hyperedges and vertices.

The two message-passing steps can be represented with two

[1]University of California, San Diego [2]Nvidia Corporation [3]Shanghai Jiao Tong University [4]Tsinghua University [5]Princeton University. Correspondence to: Jishen Zhao <jzhao@ucsd.edu>, Yu Wang <yu-wang@tsinghua.edu.cn>.

Proceedings of the 6^{th} MLSys Conference, Miami, Florida, USA, 2023.

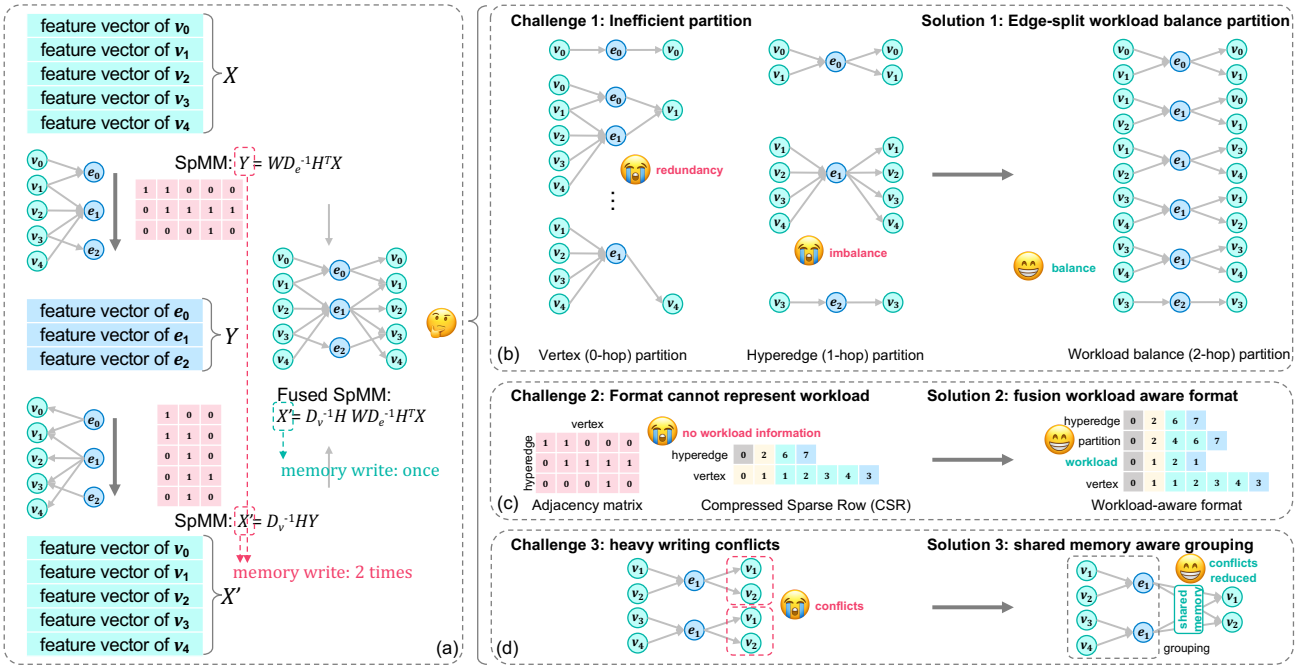

Figure 2. Overview of ***HyperGef***. (a) The efficient fusion paradigm is lacking for two SpMM operations in HyperGNNs. (b) ***HyperGef*** proposes a workload balance partitioning scheme. (c) ***HyperGef*** introduces a fusion workload aware format to represent information on workloads. (d) ***HyperGef*** utilizes shared memory to reduce conflicts in data writing.

SpMM (Sparse Matrix-Matrix Multiplication) operations between features of vertices and hyperedges, as shown in Figure 2(a). Fusing these two steps is a promising way to accelerate HyperGNNs by reducing the memory write of intermediate data (*i.e.,* Y in Figure 2(a)). However, current HyperGNN frameworks lack the paradigm for efficient fusion (which does not expose challenges in conventional GNN frameworks) because of the following challenges:

- **Inefficient partition:** Previous partitioning methods evenly divide vertices or hyperedges into partitions. However, these two basic partition methods suffer from inefficiency or imbalance problem, shown on the left of Figure 2(b).
- **Workload-Agnostic Format:** Conventional formats (*e.g.,* adjacency matrix or Compressed Sparse Row (CSR)) cannot directly represent the two-step workload because only the one-step information is given in these formats, shown on the left of Figure 2(c).
- **Heavy writing conflicts:** Different partitions need to update the same vertex, leading to heavy writing conflicts, shown on the left of Figure 2(d).

To solve these challenges, we present ***HyperGef***, a novel framework that enables efficient fusion for HyperGNNs on GPUs. ***HyperGef*** makes the following contributions:

- ***HyperGef*** proposes an **edge-split workload balance partition** which can balance the workload of two-step computation, shown on the right of Figure 2(b). Such design leads to 1.53× average speedup for typical HyperGNN models according to our experiments.
- ***HyperGef*** introduces a novel format called the **fusion workload aware format** to represent the workload of two-step computation, shown on the right of Figure 2(c).
- ***HyperGef*** utilizes a **shared memory aware grouping scheme** to reduce conflicts of updating the same vertex, shown on the right of Figure 2(d). This scheme leads to up to 1.54× speedup compared with directly updating vertices in the GPU device memory.

To the best of our knowledge, ***HyperGef*** is the first framework that provides a series of detailed investigations to propose paradigms of efficient kernel fusion for HyperGNNs. Extensive experiments demonstrate that our fused kernel outperforms two cuSPARSE-based kernels by 3.31×. ***HyperGef*** also achieves achieves 2.25× to 3.99× end-to-end speedup on various HyperGNN models compared with state-of-the-art frameworks like DGL (Wang et al., 2019) and PyG (Fey and Lenssen, 2019) for both training and inference.

The following paper is organized as below: Section 2 introduces backgrounds and preliminaries for HyperGNN models and systems. The analysis of our ***HyperGef*** kernel fusion is introduced in Section 3, followed by three techniques for efficient kernel fusion are detailed in Section 4. We conduct comprehensive experiments in Section 5 and have some discussions about this paper in Section 6. Section 7 concludes

the whole paper.

2 Background and Motivation

This section describes the preliminaries of hypergraph and HyperGNN. We will also discuss the inefficiencies of SOTA frameworks supporting HyperGNNs.

2.1 Hypergraphs

A traditional graph is represented as $G = (V, E)$, where V is a set of vertices and E is a set of edges. The group E consists of pairs of vertices, i.e., it can either describe directed or undirected connections. However, in undirected hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, the edge is defined as a set that could include any number of vertices. The directed hypergraph differs in that its hyperedges are not sets, but ordered pairs of subsets of $\mathcal{V}$ that form the tails and heads of the hyperedges. Here we let $\mathcal{V} = (v_1, v_2, \cdots, v_n)$ and $\mathcal{E} = (e_1, e_2, \cdots, e_m)$. As in HyperGNN we mainly talk about undirected cases, then for every hypergraph, we could construct an $n \times m$ incidence matrix named H. For an undirected hypergraph, $H = a_{(ij)}$ which is 1 is $v_i \in e_j$, or 0 otherwise. The incidence matrix representation is widely used in the matrix-based computation of Hypergrah NN models (Feng et al., 2019; Bai et al., 2021; Gao et al., 2022; Dong et al., 2020).

2.2 HyperGNN

Different from only one aggregation step that happens in the message-passing process of GNN (Wang et al., 2021), convolution in HyperGNNs (Feng et al., 2019; Bai et al., 2021; Gao et al., 2022) needs to be expressed by two aggregation stages (Huang and Yang, 2021; Chien et al., 2021) or equivalent to two SpMMs, as shown in figure 2(a). In the first stage of SpMM, each hyperedge in the hypergraph figure 2(a) aggregates features from vertices that belonged to it. In the second stage of SpMM, each vertex updates its feature by aggregating messages from the connected hyperedges. This two-stage message passing can be expressed by equation 1:

$$\begin{aligned} \mathbf{h}_e^{(l)} &= \mathcal{F}_1\left(\left\{\mathbf{x}_j^{(l-1)} \mid v_j \in e\right\}\right) \\ \mathbf{x}_i^{(l)} &= \mathcal{F}_2\left(\left\{\mathbf{h}_e^{(l)} \mid e \in \widetilde{\mathcal{N}}(v_i)\right\}\right) \end{aligned} \tag{1}$$

where $\mathbf{x}_i$ refers to the feature of vertex i, $\mathbf{h}_e$ refer to the feature of edge e , $\mathcal{F}_1$ and $\mathcal{F}_2$ denote the aggregation function in the first and second stage respectively. Here we choose two representative HyperGNNs that this two-stage message-passing scheme could represent.

Hypergraph Neural Network (HGNN) (Feng et al., 2019).

This model proposed a convolution operator to learn the hidden representation under the hypergraph structure. This model outperforms traditional GNN like GCN (Kipf and Welling, 2016) in graph applications and could be extended to other applications such as visual object recognition. HGNN has a two-stage sum aggregation as shown in equation 2:

$$\begin{aligned} \mathbf{h}_e^{(l)} &= \frac{1}{\sqrt{d_{v_j}}} \sum_{v_j \in e} \left\{\mathbf{x}_j^{(l-1)}\right\} \\ \mathbf{x}_i^{(l)} &= \frac{1}{\sqrt{d_{v_i}}} \sum_{e \in \widetilde{\mathcal{N}}(v_i)} \left\{\frac{w_e}{d_e} \mathbf{h}_e^{(l)} \mathbf{\Theta}\right\} \end{aligned} \tag{2}$$

In the equation 2, d_{v_i} and d_e means the degree of v_i and e, respectively. This network also has another version with row-normalization (Bai et al., 2021; Gao et al., 2022), which enjoys similar mathematical properties with equation 2. HGNN is widely used in the backbone of many models (Li et al., 2022; Zhang et al., 2022) and has been extended to various applications including group recommendations (Jia et al., 2021), multi-label image classification (Wu et al., 2020), causal inference (Ma et al., 2022), etc.

UniGNN (Huang and Yang, 2021)

GCNII (Chen et al., 2020a) is a powerful graph convolution model which could defeat over-smoothing challenges. UniGCNII put forward by UniGNN (Huang and Yang, 2021) expands the strength of GCNII to hypergraphs with the aggregation process defined as equation 3:

$$\begin{aligned} \mathbf{h}_e^{(l)} &= \frac{1}{\sqrt{d_{v_j}}} \sum_{v_j \in e} \left\{\mathbf{x}_j^{(l-1)}\right\} \\ \widetilde{\mathbf{x}}_i &= \frac{1}{\sqrt{d_e}} \sum_{e \in \widetilde{\mathcal{N}}(v_i)} \left\{\mathbf{h}_e^{(l)}\right\} \\ \mathbf{x}_i^{(l)} &= (1-\alpha)\widetilde{\mathbf{x}}_i + \alpha \mathbf{x}_i^{(l-1)} \left\{(1-\beta)\mathbf{I} + \beta \mathbf{\Theta}\right\} \end{aligned} \tag{3}$$

Another simple yet expressive model proposed by UniGNN (Huang and Yang, 2021) is UniGIN, which is generalized from Graph Isomorphism Networks(GIN) (Xu et al., 2018). UniGIN is formulated as the equation 4.

$$\begin{aligned} \mathbf{h}_e^{(l)} &= \sum_{v_j \in e} \mathbf{x}_j^{(l-1)} \\ \mathbf{x}_i^{(l)} &= \left((1+\epsilon)\mathbf{x}_i^{l-1} + \sum_{e \in \widetilde{\mathcal{N}}(v_i)} \mathbf{h}_e^{(l)}\right) \mathbf{\Theta} \end{aligned} \tag{4}$$

2.3 Kernel Fusion Support in Prior Framworks

Kernel fusion is widely used to reduce data movement, mitigate the overhead of operator execution gaps, and improve computing resource utilization on GPUs. (Sabne, 2020; Chen et al., 2018)

```
# input: graphVE, graphEV,
    feat, W, degE, degV
# vertex-to-edge aggregation
Xe = dgl.ops.copy_u_sum(
    graphVE, feat)
Xe *= degE
Xe *= W
# edge-to-vertex aggregation
Xv = dgl.ops.copy_u_sum(
    graphEV, Xe)
Xv *= degV
return Xv
```

Listing 1. HGNN in DGL

```
# input: vertexes, edges,
    feat, W, degE, degV
# vertex-to-edge aggregation
Xve = X[vertex]
Xe = scatter(Xve, edges)
Xe *= degE
Xe *= W
# edge-to-vertex aggregation
Xev = Xe[edges]
Xv = scatter(Xev, vertex)
Xv *= degV
return Xv
```

Listing 2. HGNN in PyG

```
# input: hyper_graph, feat,
    W, degE, degV
# hyper graph aggregation
Xv = HyperGef.ops.
    HyperAggregation(
    hyper_graph, feat, W,
    degE, degV)
return Xv
```

Listing 3. HGNN in ***HyperGef***

By analyzing and optimizing the computational graph logic of an existing network, kernel fusion splits, reconstructs, and fuses the original computation flow.

In traditional DNN workloads, many papers provide detailed discussions on fusion techniques, encompassing topics such as fusion exploration, code generation, and more. (Ma et al., 2020; Niu et al., 2021; Zhao et al., 2022)

However, unlike fusion in DNN, fusion in graph and hypergraph is more challenging because of dynamic shape and sparse data attributes.

The current mainstream GPU-based GNN frameworks, such as PyG (Fey and Lenssen, 2019) and DGL (Wang et al., 2019) implement few optimizations for kernel fusion. GNN-customized frameworks on GPU like (Ma et al., 2019; Wu et al., 2021a; Chen et al., 2020b; Huang et al., 2021; Fu et al., 2022; Xie et al., 2022) adopt practices that fuse GNNs within the paradigm of vertex-centric programming. (Zhang et al., 2021) extends the paradigm by using the unified thread mapping technique. However, none of these existing frameworks can optimize kernel fusion with HyperGNNs workload. As such, we need to explore high-performance fusion-based GPU frameworks optimized for HyperGNNs.

2.4 Inefficiency of Prior Frameworks

PyG (Fey and Lenssen, 2019) build up a hypergraph model with inefficient two message-passing procedures. Consider an HGNN layer. The core computation is shown in Equation 2. It first aggregates the vertex features to their connected hyperedges and generates edge messages with aggregation function *sum*, followed by normalization. It then aggregates the messages on edges back to vertices with aggregation function *sum*, followed by normalization. DGL (Wang et al., 2019) does not support hyper convolution, [1] so we use their built-in SpMM-based kernel for two steps of aggregation.

[1]Please note that DGL did not propose its HGNN model until February 2023 in its v1.0 release. Thus, we were unable to access this version while developing our paper in 2022.

The implementation in DGL breaks the update of one HGNN layer into two independent aggregation processes as shown in Listing 1, therefore causing repeated excessive global memory access as well as memory consumption.

PyG implements this process with two independent message-passing processes, which is also not efficient, as shown in Listing 2.

Table 1. Notations in message passing.

Notation	Description	Shape
$\mathbf{H}$	Normalized incidence matrix of hypergraph	$[\lvert\mathcal{V}\rvert, \lvert\mathcal{E}\rvert]$
$\mathbf{W}$	Diagonal weight parameter of hyperedge	$[\lvert\mathcal{E}\rvert, \lvert\mathcal{E}\rvert]$
$\mathbf{\Theta}$	Learning parameter of hypernode	$[K^l, K^{l+1}]$
$\mathbf{X}$	Node feature embedding	$[\lvert\mathcal{V}\rvert, K^{l+1}]$
$\mathbf{D_v}$	Degrees of hypernode	$[\lvert\mathcal{V}\rvert, \lvert\mathcal{V}\rvert]$
$\mathbf{D_e}$	Degrees of hyperedge	$[\lvert\mathcal{E}\rvert, \lvert\mathcal{E}\rvert]$

3 Analysis of Kernel Fusion

In this section, we perform a comprehensive analysis of the feasibility and benefits of kernel fusion for HyperGNNs. We adopt HGNN (Feng et al., 2019) as a concrete example to investigate the benefits of kernel fusion on HyperGNNs. Other models, such as UniGNN (Huang and Yang, 2021), can be easily adapted from HGNN by removing the trainable parameters on hyperedges (Equation 5). Therefore, benefits of kernel fusion still apply.

In general, matrix form convolution computation in HGNN can be described by the following message-passing formula:

$$\mathbf{X}^{(l)} = \sigma\left(\mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}\mathbf{D}_\mathbf{e}^{-1}\mathbf{H}^\top\mathbf{D}_v^{-1/2}\mathbf{X}^{(l-1)}\mathbf{\Theta}^{(l)}\right) \quad (5)$$

where the notations are described in Table 1. $\mathbf{H}$ is the hypergraph adjacent matrix, $\mathbf{\Theta}$ is a linear transformation, and σ is an activation function. Note that $\mathbf{W}$ is a diagonal matrix.

Table 2. Different calculating methods demonstrate the forward and backward propagation of HGNN message passing.

Calculation method	Forward Propagation	Memory footprint	Backward Propagation	Memory footprint
Two SpMMs	$\mathbf{X_1} = \mathbf{G^T X'}$ $\mathbf{Y} = \mathbf{G X_1}$	$(\lvert\mathcal{V}\rvert + \lvert\mathcal{E}\rvert)K$	$\mathbf{\Delta X'} = \mathbf{G \Delta X_1}$ $\mathbf{\Delta X_1} = \mathbf{G^T \Delta Y}$	$(\lvert\mathcal{V}\rvert + \lvert\mathcal{E}\rvert)K$
Fused	$\mathbf{Y} = \mathbf{G G^T X'}$	$\lvert\mathcal{V}\rvert K$	$\mathbf{\Delta X'} = \mathbf{G G^T \Delta Y}$	$\lvert\mathcal{V}\rvert K$

Because the multiplication of parameter $\mathbf{\Theta}$ and $\mathbf{X}$ is performed by a GEMM (General Matrix-Matrix Multiplication) operator, we can combine these two into a whole. (Note that the GEMM operator has a standalone schedule and has been well-optimized, thus is not in the fusion scope.) Furthermore, we can simply integrate the normalized diagonal matrix $\mathbf{W}$, $\mathbf{D_v^{-1/2}}$ and $\mathbf{D_e^{-1}}$ into $\mathbf{H}$. As a result, the message-passing formula can be simplified as the following equation $\mathbf{Y} = \mathbf{G G^T X'}$ once we substitute $\mathbf{X\Theta}$ with $\mathbf{X'}$ and substitute $\mathbf{D}_v^{-1/2}\mathbf{H}\mathbf{W}^{1/2}\mathbf{D}_e^{-1/2}$ with $\mathbf{G}$.

With the above simplification, we compare the calculation between the fusion and non-fusion methods on both forward and backward propagation, as shown in Table 2. As for the non-fusion method, we interpret the overall computational process mainly as two sparse matrix multiplication (SpMM) operations. We point out that using the SpGEMM method to compute $\mathbf{G G^T}$ is inefficient, for we have to compute a new sparse matrix, which takes up large computing overhead (Parger et al., 2020; Winter et al., 2019; Liu and Vinter, 2014). Also, the sparsity of the new sparse matrix $\mathbf{G'} = \mathbf{G G^T}$ is much larger than $\mathbf{G}$, which leads to the SpMM function of calculating $\mathbf{G'X'}$ even slower than non-fusion based two SpMMs. Another drawback is that directly calculating the SpGEMM cannot accommodate other aggregation types, which commonly appear in GNN training. Please refer to our discussion Section 6 for more detail. By directly fusing the whole computation of HGNN convolution, it could save memory by eliminating the intermediate hyperedge feature as shown in table 2. Also, in Figure 2 (a) we show operator fusion could also avoid the global read/write of the hypergraph feature caused by the second SpMM.

4 *HyperGef* DESIGN

Overview. Rather than the two-step message-passing procedures, we design a simple frontend to express the message-passing procedure in hypergraph with a unified aggregation interface (Listing 3). It is easy to use and implement with efficient operator fusion and workload balance techniques.

4.1 Workload-Balanced Efficient Partition

In order to compute fusion in parallel, the partitioned workload assigned to parallel workers is essential for determining the overall performance. A well-designed partition will reduce the hardware overhead, exploit the potential parallelism and achieve better workload balancing. In particular for balancing problems on GPU, while dynamic scheduling methods (Fu et al., 2022; Wu et al., 2021a), which primarily encompass persistent threads and GPU warps' latency-hiding techniques, can address the graph imbalance to some extent, these approaches cannot allocate more warps to vertices with larger workloads. Consequently, they do not fundamentally resolve the imbalance problem. We investigate various partition strategies and create a novel solution that can achieve efficiency with a balanced workload.

4.1.1 Workload Definition

To clearly demonstrate the partition algorithm, we first need to define the workload of fusion in HyperGNN. Different partition methods can result in varying computation and memory access workloads. Here for better illustration, we set the worker to be the block under the CUDA programming semantics.

In Figure 3, we list three types of arrows representing workloads that appeared in different partitions. The grey arrow pointing from the input vertex to the hyperedge represents the read operation from the input vertex. The blue arrow symbolizes the write operation from the hyperedge to the output vertex, while the dotted blue arrow denotes the atomic operation. If multiple arrows point to the same circle, this leads to reduction operations, which are highlighted in red text. By presenting this workload definition, we can effectively analyze the advantages and disadvantages of the subsequent partition algorithms.

4.1.2 Edge-Split Workload Balance Partition

Motivation. While the hypergraph partition has been implemented in frameworks like (Heintz et al., 2019; Jiang et al., 2018), they mainly focus on the optimal cut for supporting the distributed system and targeting the traditional hypergraph algorithms. Similar to partition algorithms for solving the GNN workload on GPU such as vertex partition (Huang et al., 2020), and edge partition (Dai et al., 2022), we illustrate these two different types of partitions as shown in figure 3.

Challenges. The partition for the HyperGNN workload

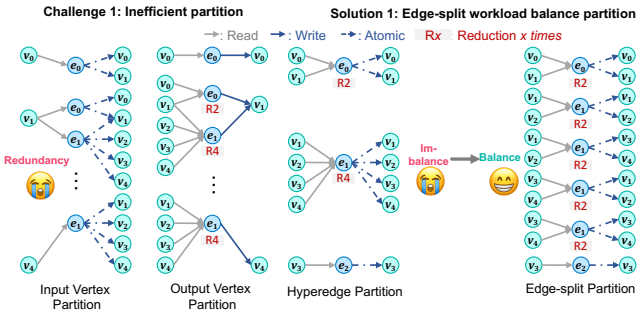

Figure 3. The workload analysis over vertex balance, edge balance, and our edge-split balance method are demonstrated by different colors of arrows. The reduction's workload is shown in red text with a number to indicate.

is not well-studied, and the partition paradigms we gave before are not optimized. Since the vertex partition needs to iterate the second-hop neighbors of the vertex, it leads to more workload compared to the original non-fusion operator SpMM. To be specific, the input vertex partition brings large atomic operations overhead because of overlapping output vertexes, while the output vertex partition results in the redundant overhead of read operations. As for the hyperedge partition, the output and input vertex indexes are the same within each edge partition, which implies it is needless to iterate the second-hop neighbors. However, this hyperedge partition has a severe imbalance brought by the hypergraph's power-law degrees attribute. In our demonstrated hypergraph, e_1 is a super hyperedge since it contains much more vertex connections than e_0 or e_2. The workload of the partition of e_1, which is four reductions and four read operations in detail, leads to a severe tail effect on GPU. (Note that atomic operations on GPU are handled by L2 cache but not warps in SM, so we do not calculate here within a partition.)

Insight. Our key insight is to eliminate the overload brought by super hyperedges, e.g. e_1 in Figure 3 to reduce the imbalance. The major problem here is that only one worker to compute the e_1 partition would hinder the overall parallel process. If some method enables us to cut the workload of e_1 into partitions that own evenly distributed workloads, then it would address the imbalance under this case.

Approach. We propose an adaptive edge-split workload balance partition that derivates from the hyperedge partition. Essentially, we cut the connections into small groups so that we could generate more balanced partitions and bring more parallelism. As for the overall computation flow in the e_1 partition, e_1 needs to reduce messages from four vertexes and scatter the reduction result to the same four vertexes. Thus, the output of each output vertex in the partition e_1 is related to all of the input vertexes. To implement our adaptive workload partition, we first select a hyper-parameter g_s that indicates a split in the super edge if there exists a number of vertexes larger than this hyper-parameter. For example, let us choose $g_s = 2$ here. Thus, we divide the vertexes into two groups, v_1 and v_2 form group one, v_3 and v_4 form group two, respectively. According to the computation flow we analyzed before, the reduction of group one needs to be scattered to all of the vertexes in group one and group two. That is the reason why the e_1 is separated into four independent partitions. Thus, the standard deviation of all partitions workload here is 0.41, much lower than that of the hyperedge-balanced partition which is 1.53. Generally, if we apply g_s to an arbitrary large hyperedge with h_e vertexes connections, then we will generate $\lceil h_e/g_s \rceil^2$ different partitions as a result. We design g_s that is adaptive to the input data attributes since g_s affects other two factors not restricted to imbalance. First of all, g_s would change the parallelism. If the input hypergraph is relatively small, more partitions will fill up SMs resources on GPU which leads to higher GPU hardware utilization. Secondly, the overall atomic operations increase with g_s going down, thus it is important to find an adequate g_s to apply a trade-off between parallelism, imbalance, and less atomics. We further discuss our tuning result in Section 5.4.

4.2 Fusion Workload Aware Format

Motivation. Many existing techniques to indicate the workload are applying dynamic scheduler (Yang et al., 2018; Merrill and Garland, 2016; Dai et al., 2022), with no transformation of the original CSR format. However, these types of dynamic schedulers need additional operations like searching and synchronizing (Dai et al., 2022), which cause overhead in the schedule. Some existing solutions in GNN (Huang et al., 2021; Wang et al., 2021) reduce this runtime overhead by generating the static format to indicate different partitions, which motivates us to promote a workload-aware format for HyperGNN.

Challenge. An adequate format that is aware of the balanced workload partition reduces the runtime overhead, thus vital to the parallel kernel acceleration. The major challenge here is the lack of a fusion-workload-aware format to indicate our partition algorithm introduced above. As shown in Challenge (c) in figure 2, the widely-used CSR format only represents the vertex workload within each hyperedge(or the hyperedge workload within each vertex). Thus, using the CSR format is unable to represent our balanced partition in Section 4.1. Other variations of sparse formats (Maggioni and Berger-Wolf, 2013; Liu and Vinter, 2015; Chou et al., 2018) do not target this workload such fail to solve the problem well. Naive formats can indicate every detailed information within each partition while ending up with a large memory access overhead in the runtime kernel schedule. For example, the second partition related to e_1 has the input connections of v_1 and v_2, output connections of v_3, v_4. Thus, the naive format records all the output and input

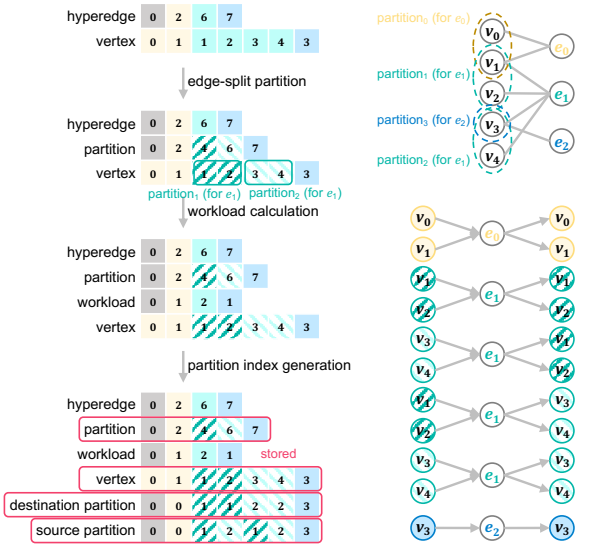

Figure 4. From the original CSR format (left top) to the fusion workload aware format (left bottom).

vertex indexes, leading to a large memory access overhead in scheduling.

Insight. To generate a workload-aware format that is also friendly to schedule, we are inspired by the fact that the input vertexes and output vertexes share equal partition groups. To be specific, here we take the second partition of e_1 in the two-hop balanced algorithm for example. With the knowledge of v_1 and v_2 belong to group one, and v_3 and v_4 belong to group two, we only need to take down the group indexes of input and output.

Approach. Our workload-aware format is deduced by the following process. The first step is to mark the boundary of different partitions given the g_s condition. In this example, we still set g_s equal to two. Observing that e_1 is larger than g_s, so we iterate through the vertexes in e_1 to find the boundary according to g_s. Since e_1 here contains four vertexes, there exists only one boundary that lies between vertex 2 and vertex 3. To indicate the boundary's information, we use the starting location of the second group within e_1, which is the fourth nonzero of the whole vertex index array. Thus, we could derive the partition array by applying the group information from the original CSR row pointer. After introducing the partition array, we still need to indicate each input and output of independent partitions. To better describe our approach, we utilize a workload array here which demonstrates how many groups are located within each hyperedge. Note that there are total $\lceil h_e/g_s \rceil^2$ within each hyperedge that has h_e vertexes, we generate every pair of input and output within the form of (g_i, g_o). g_i and g_o would first iterate through 0 to $\lceil h_e/g_s \rceil - 1$, respectively. Plus, both g_i and g_o need to add the starting offset of h_e, which is indicated by the prefix sum of the workload before h_e.

4.3 Shared Memory Aware Grouping

Motivation. Our two-hop partition utilizes a large number of atomic operations for reduction to compute the final output vertex. We want to exploit the potential benefits of shared memory so as to reduce the overhead brought by the atomic operations.

Challenge. Unlike applying shared memory techniques to dense workloads, the unpredictability of sparse workloads makes shared-memory-aware schedules difficult to implement. Besides, the well-studied shared memory optimization in traditional GNN workload cannot be tailored to HyperGNN workload. For example, (Huang et al., 2020) proposes a coalesced row caching technique that is restricted to the vertex-balanced partition, while the method given by (Wang et al., 2021) is under the condition of GNN's neighbor-group partition. Thus, we have to design a shared memory scheduler that is tailored to our two-hop balanced partition algorithm.

Insight. Our key insight here is that we find the adjacent partitions by our two-hop balanced algorithm that could target the same output group. Thus, if we combine them

Algorithm 1 Shared-memory grouping schedule

Input: group key g_k, vertex indexes v_i input groups g_i, output groups g_o, input vertex feature $\mathbf{F}_i$
Output: output vertex feature $\mathbf{F}_o$
# P Warps, $\mathcal{P}$ Partitions
warpId = partitionId
# Reduce the input vertexes feature
inVertexes = `getVertex`(v_i, g_i)
acc ← `reduceAll`(inVertexes, $\mathbf{F}_i$, acc)
# Store temporary reduction in Shared Memory
warpId.sharedAcc = acc
__syncthreads()
outVertexes = `getVertex`(v_i, g_o)
# Use a list to record warpId for grouped output
OutWarpList[warpId] = `calcOutWarp`(g_o, $\mathcal{P}$)
if warpId in OutWarpList **then**
  nextOutWarpId = OutWarpList[warpId+1]
  # Shared memory based prefix sum
  for p in `range`(warpId, nextOutWarpId) **do**
    warpId.sharedAcc += p.sharedAcc
  end for
end if
__syncthreads()
# Only use atomic when warpId belongs to output list
if warpId in OutWarpList **then**
  for v **in** outVertexes **do**
    `atomicAdd`($\mathbf{F}_o[v]$, warpId.sharedAcc)
  end for
end if

Table 3. Hypergraph datasets used in our experiments.

	Cora	Citeseer	Pubmed	Cora-CA	DBLP-CA	Zoo	20News	Mushroom	NTU2012	ModelNet40	Yelp	House	Walmart
#Vertex	2708	3312	19717	2708	41302	101	16242	8124	2012	12311	50758	1290	88860
#Edge	1579	1079	7963	1072	22363	43	100	298	2012	12311	679302	341	69906
#Feature	1433	3703	500	1433	1425	16	100	22	100	100	1862	100	100
#Class	7	6	3	7	6	7	4	2	67	40	9	2	11
max $\|e\|$	5	26	171	43	202	93	2241	1808	5	5	2838	81	25

together to compute, we could reduce the atomic operations by being aware of the shared memory schedule.

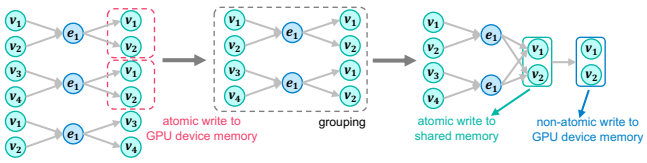

Figure 5. Grouping partitions to utilize shared memory, which reduces atomic write to the GPU device memory.

Approach. With the knowledge of adjacent partitions is more likely to have the same output, we implement a group-wise combination schedule that utilizes the shared memory to generate a temporary reduction result. Essentially, we group the adjacent partitions to be executed together in one GPU block. Different partitions will be distributed to different warps within a block. To better express our shared-memory-aware schedule, we write down the specific process as shown in Algorithm 1. Here, we allocate P warps per block mapping to $\mathcal{P}$ partitions respectively. Our schedule will check whether adjacent partitions have the same output, and calculate the leading output warpID in the OutWarpList. Then, if the current warp is in the OutWarpList, this warp will go through a for loop to reduce the results stored in the shared memory until it meets the next leading output warpID, which is similar to a prefix sum operator. After the prefix sum, we only apply atomic operations to warpId which belongs to the output list, thus it enables us to reduce the overall atomic overhead.

5 Experiment

5.1 Experimental Setup

Baselines: First we choose two commonly used GNN frameworks that support HyperGNN training on GPUs, and we also use the cuSPARSE vendor library as a baseline for our kernel design.

Deep Graph Library (DGL) (Wang et al., 2019) is one of the SOTA GNN frameworks that support both GPU and CPU with a backend built upon framework PyTorch, Tensorflow, and MXNet. DGL utilizes a generalized-SDDMM and generalized-SpMM abstraction to express the message-passing paradigm with sparse matrix operations. We use DGL's PyTorch backend on GPU for comparison.

PyTorch-Geometric (PyG) (Fey and Lenssen, 2019) is another SOTA GNN framework for GPU and CPU built upon PyTorch. PyG provides the user with a general message-passing interface that is highly flexible and customizable.

cuSPARSE (Naumov et al., 2010) is the standard sparse linear algebra library maintained by NVIDIA. We use the cusparseSpMM API in the library to simulate the prototype of the non-fusion operator in HyperGNN.

Datasets: We use the datasets from Allset (Chien et al., 2021), which use thirteen real-world hypergraphs for node classification tasks. Statistics are summarised in Table 3.

Platform & Metrics: We implement our proposed technique with a CUDA backend and a PyTorch-based frontend. The evaluation platform we used is a server with one 10-core 20-thread Intel Xeon Silver 4210 CPU @ 2.2GHz and an NVIDIA 3090 GPU with CUDA 11.3. For the software version, we use Pytorch 1.11, PyG 2.0, and DGL 0.9.

5.2 End-to-end Performance

Figure 6 shows the end-to-end performance on all 13 datasets under representative hidden dimensions (*i.e.,* 32, 64, and 128). ***HyperGef*** outperforms the other two SOTA systems on all three HyperGNN networks in every dataset. The geomean speedup of ***HyperGef*** is 2.25× and 2.60× over DGL and PyG for training, respectively. For inference, ***HyperGef*** gains 3.50× and 3.99× geomean speedup compared to DGL and PyG. With the growth of feature size, ***HyperGef*** continuously outperforms DGL and PyG in both inference and training for all model settings. In particular, ***HyperGef*** reaches up to 16.59× training speedup compared to PyG and 4.38× speedup compared to DGL. Note that here we choose the hyperparameter g_s as a result of our kernel-based tuning. In our framework implementation, we utilize the mutual transposition information between H and H_t, needless to use transpose kernel within the backward propagation. It is worth noting that kernel fusion will largely reduce kernel launching times, which also makes our framework much more efficient compared to DGL and PyG.

5.3 Kernel Performance

We test the performance of kernels within our ***HyperGef*** framework. To be specific, the fusion kernel is compared to two cascades of SpMM from the vendor cuSPARSE library. The fusion kernel shown in Figure 7 is optimized by all

Figure 6. The overall end-to-end results of ***HyperGef*** compared to PyG and DGL. The horizontal coordinates correspond to different datasets, in the same order as in Table 3. In each model respectively, we set DGL performance as the normalization baseline.

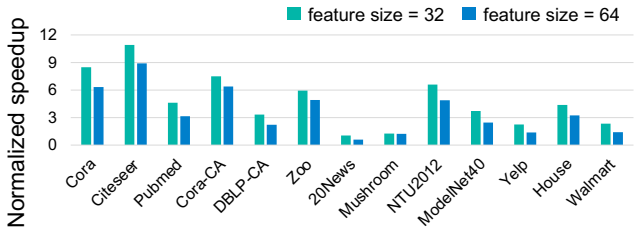

Figure 7. Overall normalized kernel performance of ***HyperGef*** under feature size of 32 and 64. We set the cuSPARSE as our normalization baseline.

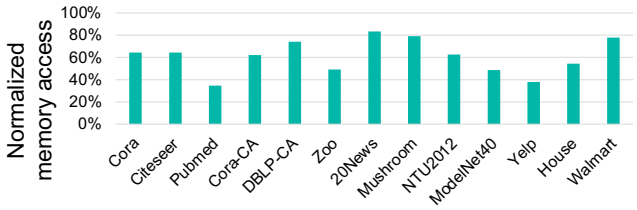

Figure 8. The normalized global memory access of edge-balanced partition compared to cuSPARSE under feature size is 32.

of our three techniques with a selected hyperparameter g_s after tuning. Each kernel runs 200 times and we perform the geomean time as the final result. As shown in Figure 7, our kernel performance beats cuSPARSE in all of our datasets under feature size 32, with a geomean of 3.89× and a maximum speedup of up to 10.9×. Under feature size 64, we gain a geomean of 2.81× and a maximum of 8.91× speedup.

5.4 Ablation study

In this section, we present the performance gains brought by each technique separately and have an analysis of the reasons for acceleration.

Kernel fusion. As we introduced in Figure 2 (a), kernel fusion can reduce memory access by eliminating the read and write access caused by $\mathbf{Y}$, thus bringing a significant speedup over two SpMMs' computation flow. Figure 8 shows that we can save an average of 39% and a peak up to 65% global memory access compared to cuSPRARSE-based SpMM by utilizing the edge-balanced partition kernel fusion. Note that here the global memory access is defined as the total read and write transactions from the DRAM of GPUs. In our experiments, we find that kernel fusion with edge balanced itself achieves an average of 2.58× and up to 9.06× speedup over cuSPARSE.

Worload Balanced Partition. We proposed an Edge Split workload balanced partition to explore the parallelism within fusion and reduce the imbalance. Intuitively, the performance of kernels should be affected by the number and distribution of non-element elements in the adjacency matrix. To deal with the graph input dynamics that affect the partition strategy, we tune the performance of our fusion ker-

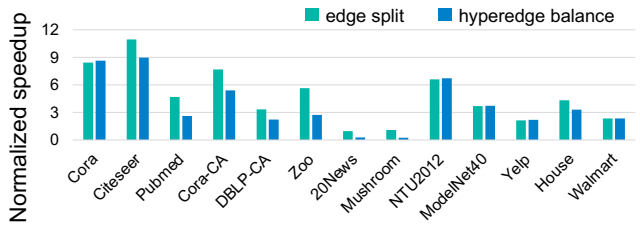

Figure 9. The normalization speedup of edge split and hyperedge balance algorithm based on cuSPARSE.

nels under different h_s to get better overall performance. In Figure 9, our kernel empowered by workload-balanced partition reaches a 5.31× speedup compared to naive-version, with an average speedup of 1.53×. Note that we find using the edge split method gains extremely large speedup, especially for those data suffering from severe imbalance such as Mushroom and 20News.

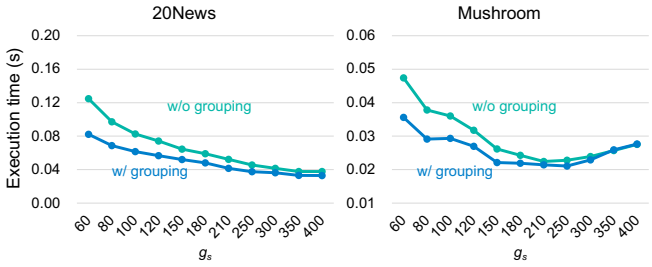

Figure 10. The execution time of schedule that is aware of shared memory grouping compared to schedule only with edge split partition. The horizontal axis shows different values of g_s.

Shared Memory Aware Grouping. To show the benefit of the shared memory grouping method, we use two representative datasets Mushroom and 20newsW100 which have salient attributes of imbalance. The number of partitions in block $\mathcal{P}$ is set to 2. As illustrated in Figure 10, kernels using shared memory exhibits better performance compared to those without shared memory grouping. In particular, in all of the partition settings we choose, our shared memory aware grouping algorithm gains a speedup of 1.33× on the Mushroom dataset and a speedup of 1.52× on the 20News dataset. This is mainly because our grouping algorithm effectively reduces the number of atomic operations. The gaps between shared memory grouping and schedule without it will become closer when g_s goes larger since a larger g_s will only cut those bigger hyperedges into different partitions. As a result, fewer hyperedges will be separated into different partitions which reduces the chances of adjacent partitions. Another interesting thing we find is that under the setting when g_s goes larger, we find the performance with shared memory grows higher at first. Then it would reach an optimized status with a specific partition size g_s, as shown in the Mushroom dataset in Figure 10. Though it is hard to fully explain the phenomenon, we summarized this phenomenon as a tradeoff of atomic operations, balanced workload, and parallelism, as we analyzed in 4.1.2.

6 Discussions

Overhead of Fusion Workload Aware Format. To support our kernel backend, we need to generate our fusion-workload aware format deduced from the original CSR. Note that we have already chosen the heuristic of h_s in our end-to-end implementation, thus we only need to generate such a format once. Our experiments show this preprocessing time of format conversion is trivial compared to training time, which is less than 5% of training time on average.

Using SpGEMM In section 3 we mentioned the inefficiency of the SpGEMM method to calculate the aggregation according to our experiments. In specific, take the dataset widely-used Pubmed as an example, setting the incidence matrix of Pubmed as $\mathbf{G}$, input feature as $\mathbf{X}$, the time of calculating $\mathbf{G}'\mathbf{X}$ is even larger than that of calculating two cascaded SpMMs. This is due to the fact that the sparsity of $\mathbf{G}'$ is much larger than that of $\mathbf{G}$. (2.9× times sparsity growth).

Mini-batch training. Though we didn't find models that partition a large hypergraph into mini-batches for hypergraph training, we discover research like (Lim et al., 2022) uses mini-batch training and HGNN as its backbone network. In this setting, our work can be easily extended to adapt to mini-batches by using the format conversion for every hypergraph batch.

Non-SpMM aggregation. For system implementation, non-SpMM aggregation impacts our techniques, such as partition strategy. Our technique fully supports the widely used mean aggregation. However, when it comes to supporting max aggregation, our methods will degrade to the original hyperedge partition.

7 Conclusions

In this paper, we introduce ***HyperGef***, the first framework that proposes paradigms of efficient kernel fusion for Hypergraph Neural Networks. ***HyperGef*** proposes an edge-split workload balance partition method to process the fused kernel in parallel. The workload can be calculated using the fusion workload aware format, rather than inferred from conventional formats with runtime overheads. ***HyperGef*** also utilizes shared memory to reduce data writing conflicts to GPU device memory. The fused kernel outperforms two cuSPARSE-based kernels by 3.31×, and ***HyperGef*** achieves 2.25× to 3.99× end-to-end speedup on various HyperGNN models compared with state-of-the-art frameworks like DGL (Wang et al., 2019) and PyG (Fey and Lenssen, 2019) for both training and inference.

A ARTIFACT APPENDIX

A.1 Abstract

Our work proposes a novel kernel fusion methodology to optimize the computation of HyperGNNs on GPUs. Our work mainly consists of two parts. The first part is the GPU kernels responsible for the aggregation of HyperGNN models and is implemented with our proposed kernel fusion, efficient format, and shared memory optimization. The second part is the Python code that wraps the kernels to provide a PyTorch-based front-end and uses them as building blocks to build up different HyperGNN models. Our experiments show our proposed methods significantly improve the performance of representative HyperGNN models on mainstream hypergraphs datasets.

A.2 Artifact check-list (meta-information)

- **Program:** https://github.com/AEtmp/HyperGef_AE.
- **Hardware:**
  - Intel CPU x86_64 with host memory $\geq$ 64GB. Tested on Intel Xeon Gold 6226R (16-core 32-thread) CPU with 251 GB host memory and 2.90GHz frequency.
  - 24 GB device memory RTX3090, which has 82 SM cores and 128KB L1 cache per SM.
- **Compilation:** Ubuntu 20.04+, nvcc(CUDA 11.3+).

A.3 Description

A.3.1 How delivered

The source code and scripts are available at *https://github.com/AEtmp/HyperGef_AE* and archirved at *https://doi.org/10.5281/zenodo.7894072*

A.3.2 Hardware dependencies

Our implementation works on Intel x86 CPUs and Nvidia GPUs.

A.3.3 Software dependencies

- CUDA 11.3+
- PyTorch 1.8.0+ (Do not support 2.0+)
- DGL 0.9.0+
- PyG 2.1+
- Ninja 1.10+
- GPUtil 1.4+

A.4 Installation

To build our software, you need to install Ninja and PyTorch as shown in the dependencies. We use the PyTorch extension to build a runtime library.

A.5 Experiment workflow

We have prepared the README in the source code repository, which included other setup steps like downloading datasets, etc. Please make sure to finish these steps before doing the following experiments.

- Go to `experiment/` directory.
- **Figure 6 result:** `python fig6.py` to run end-to-end experiments on three HyperGNN models. Generate `fig6.csv`.
- **Figure 7 result:** `python fig7.py` to run overall normalized kernel performance. Generate `fig7.csv`.
- **Figure 8 result:** `python fig8.py` to run an ablation study for operator fusion. Generate `fig8.csv`.
- **Figure 9 result:** `python fig9.py` to run the ablation study for the edge split partition algorithm. Generate `fig9.csv`.
- **Figure 10 result:** `python fig10.py` to run an ablation study for shared memory aware grouping and generate `fig10.csv`.

A.6 Evaluation and expected result

Once you have run the experiment workflow, you can see the `.csv` result under the `experiment/` directory. The related results are stored in `figX.csv`. To be specific, the result of figure8 is stored in the `experiment/profile/` directory.

The data we used for figures in the paper is given in the folder `example_data`.

A.7 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

REFERENCES

Song Bai, Feihu Zhang, and Philip HS Torr. 2021. Hypergraph convolution and hypergraph attention. *Pattern Recognition* 110 (2021), 107637.

Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. 2020a. Simple and deep graph convolutional networks. In *International Conference on Machine Learning*. PMLR, 1725–1735.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799* (2018).

Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. 2020b. fuseGNN: accelerating graph convolutional neural network training on GPGPU. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

Dian Cheng, Jiawei Chen, Wenjun Peng, Wenqin Ye, Fuyu Lv, Tao Zhuang, Xiaoyi Zeng, and Xiangnan He. 2022. IHGNN: Interactive Hypergraph Neural Network for Personalized Product Search. In *Proceedings of the ACM Web Conference 2022*. 256–265.

Eli Chien, Chao Pan, Jianhao Peng, and Olgica Milenkovic. 2021. You are allset: A multiset function framework for hypergraph neural networks. *arXiv preprint arXiv:2106.13264* (2021).

Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2018. Format Abstraction for Sparse Tensor Algebra Compilers. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 123 (oct 2018), 30 pages. https://doi.org/10.1145/3276493

Guohao Dai, Guyue Huang, Shang Yang, Zhongming Yu, Hengrui Zhang, Yufei Ding, Yuan Xie, Huazhong Yang, and Yu Wang. 2022. Heuristic Adaptability to Input Dynamics for SpMM on GPUs. *arXiv preprint arXiv:2202.08556* (2022).

Yihe Dong, Will Sawin, and Yoshua Bengio. 2020. HNHN: hypergraph networks with hyperedge neurons. *arXiv preprint arXiv:2006.12278* (2020).

Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. 2019. Hypergraph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 3558–3565.

Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

Qiang Fu, Yuede Ji, and H Howie Huang. 2022. TLPGNN: A lightweight two-level parallelism paradigm for graph neural network computation on GPU. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 122–134.

Yue Gao, Yifan Feng, Shuyi Ji, and Rongrong Ji. 2022. HGNN$\hat{}$+: General Hypergraph Neural Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022).

Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.

Benjamin Heintz, Rankyung Hong, Shivangi Singh, Gaurav Khandelwal, Corey Tesdahl, and Abhishek Chandra. 2019. MESH: A flexible distributed hypergraph processing system. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 12–22.

Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: 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.

Jing Huang and Jie Yang. 2021. UniGNN: a Unified Framework for Graph and Hypergraph Neural Networks. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*.

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*. 119–132.

Renqi Jia, Xiaofei Zhou, Linhua Dong, and Shirui Pan. 2021. Hypergraph convolutional network for group recommendation. In *2021 IEEE International Conference on Data Mining (ICDM)*. IEEE, 260–269.

Wenkai Jiang, Jianzhong Qi, Jeffrey Xu Yu, Jin Huang, and Rui Zhang. 2018. HyperX: A scalable hypergraph framework. *IEEE Transactions on Knowledge and Data Engineering* 31, 5 (2018), 909–922.

Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

Mengran Li, Yong Zhang, Xiaoyong Li, Yuchen Zhang, and Baocai Yin. 2022. Hypergraph Transformer Neural Networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* (2022).

Jongin Lim, Sangdoo Yun, Seulki Park, and Jin Young Choi. 2022. Hypergraph-induced semantic tuplet loss for deep metric learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 212–222.

Weifeng Liu and Brian Vinter. 2014. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 370–381.

Weifeng Liu and Brian Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. 339–350.

Jing Ma, Mengting Wan, Longqi Yang, Jundong Li, Brent Hecht, and Jaime Teevan. 2022. Learning Causal Effects on Hypergraphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 1202–1212.

Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. 2020. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. 881–897.

Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs.. In *USENIX Annual Technical Conference*. 443–458.

Marco Maggioni and Tanya Berger-Wolf. 2013. AdELL: An adaptive warp-balancing ELL format for efficient sparse matrix-vector multiplication on GPUs. In *2013 42nd international conference on parallel processing*. IEEE, 11–20.

Duane Merrill and Michael Garland. 2016. Merge-based sparse matrix-vector multiplication (spmv) using the csr storage format. *ACM SIGPLAN Notices* 51, 8 (2016), 1–2.

Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. 2010. Cusparse library. In *GPU Technology Conference*.

Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 883–898.

Mathias Parger, Martin Winter, Daniel Mlakar, and Markus Steinberger. 2020. spECK: accelerating GPU sparse matrix-matrix multiplication through lightweight analysis. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 362–375.

Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

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. (2019).

Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. {GNNAdvisor}: An Adaptive and Efficient Runtime System for {GNN} Acceleration on {GPUs}. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. 515–531.

Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2019. Adaptive sparse matrix-matrix multiplication on the gpu. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. 68–81.

Xiangping Wu, Qingcai Chen, Wei Li, Yulun Xiao, and Baotian Hu. 2020. Adahgnn: Adaptive hypergraph neural networks for multi-label image classification. In *Proceedings of the 28th ACM International Conference on Multimedia*. 284–293.

Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021a. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 359–375.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021b. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. *Proceedings of Machine Learning and Systems* 4 (2022), 515–528.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks? https://doi.org/10.48550/ARXIV.1810.00826

Naganand Yadati, Madhav Nimishakavi, Prateek Yadav, Vikram Nitin, Anand Louis, and Partha Talukdar. 2019. Hypergcn: A new method for training graph convolutional networks on hypergraphs. *Advances in neural information processing systems* 32 (2019).

Carl Yang, Aydın Buluç, and John D Owens. 2018. Design principles for sparse matrix multiplication on the gpu. In *European Conference on Parallel Processing*. Springer, 672–687.

Junliang Yu, Hongzhi Yin, Jundong Li, Qinyong Wang, Nguyen Quoc Viet Hung, and Xiangliang Zhang. 2021. Self-Supervised Multi-Channel Hypergraph Convolutional Network for Social Recommendation. In *Proceedings of the Web Conference 2021* (Ljubljana, Slovenia) *(WWW '21)*. Association for Computing Machinery, New York, NY, USA, 413–424. https://doi.org/10.1145/3442381.3449844

Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2021. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective. *arXiv preprint arXiv:2110.09524* (2021).

Zizhao Zhang, Yifan Feng, Shihui Ying, and Yue Gao. 2022. Deep Hypergraph Structure Learning. *arXiv preprint arXiv:2208.12547* (2022).

Jie Zhao, Xiong Gao, Ruijie Xia, Zhaochuang Zhang, Deshi Chen, Lei Chen, Renwei Zhang, Zhen Geng, Bin Cheng, and Xuefeng Jin. 2022. Apollo: Automatic partition-based operator fusion through layer by layer optimization. *Proceedings of Machine Learning and Systems* 4 (2022), 1–19.