Optimizing Irregular Dense Operators of Heterogeneous GNN Models on GPU

Israt Nisa
AWS AI
Santa Clara, CA, USA
nisisrat@amazon.com

Minjie Wang
AWS AI
Shanghai, China
minjiw@amazon.com

Da Zheng
AWS AI
Santa Clara, CA, USA
dzzhen@amazon.com

Qiang Fu
George Washington University
Washington, DC, USA
charlesfoo@email.gwu.edu

Ümit Çatalyürek
Georgia Institute of Technology
Atlanta, Georgia, USA
uvc@amazon.com

George Karypis
AWS AI
Santa Clara, CA, USA
gkarypis@amazon.com

***Abstract*—GNN models on heterogeneous graphs have achieved state-of-the-art (SOTA) performance in various graph tasks such as link prediction and node classification. Despite their success in providing SOTA results, popular GNN libraries, such as PyG and DGL, fail to provide fast and efficient solutions for heterogeneous GNN models. One common key bottlenecks of models like RGAT, RGCN, and HGT is relation-specific linear projection. In this paper, we propose two high-performing tensor operators: gather-mm and segment-mm to address the issue. We demonstrate the effectiveness of the proposed operators in training two popular heterogeneous GNN models – RGCN and HGT. Our proposed approaches outperform the full-batch training time of RGCN by up to $3\times$ and mini-batch by up to $2\times$.**

***Index Terms*—Graph Neural Network, heterogeneous GNN models, GPU, matrix multiplication**

I. INTRODUCTION

Graph representation learning is becoming a popular technique for performing various graph tasks like node classification, link prediction, and graph classification [1], [2], [3]. In recent years, several Graph Neural Network (GNN) libraries like PyG [4], TFGNN [5] and DGL [6] were built over generic machine learning frameworks like PyTorch, TensorFlow and/or MXNet. Most of these frameworks are well equipped to handle homogeneous graph-structure data (single node- and edge-type) but fall short in handling heterogeneous GNN models. Most the real-world graphs such as social networks and citation networks have heterogeneous graphs structure consisting of multiple node- and edge-types. For example, a citation network can be modeled as a heterogeneous graph with two relation types $\langle$`author-writes-paper`$\rangle$ and $\langle$`paper-cites-paper`$\rangle$; and two node types $\langle$`author`$\rangle$ and $\langle$`paper`$\rangle$.

The computational overhead of heterogeneous GNN models is higher than their homogeneous counterparts. To train a GNN model on a homogeneous graph, the computation modules such as computing messages (in algebraic term - Sampled dense-dense matrix multiplication) or gather attention scores on edges (in algebraic term - dense-dense matrix multiplication) or accumulating neighborhood information (in algebraic term - sparse-dense matrix multiplication) are invoked once for a full iteration. On the other hand, For heterogeneous GNN models,these computations are performed across different node types and edge types. For example, a popular GNN model Relational GCN [7] aggregates relation-specific linearly projected featured vectors from neighboring nodes. Usually, these type specific operations are invoked separately which often produces many small operations. When a GNN model is trained on a GPU architecture, these small functions lead to severely underutilized GPUs in addition to CUDA kernel launch overhead. Frameworks like DGL and PyG which rely on PyTorch's autograd engine to compute the gradients, have additional overheads to convert the Python calls to C++/CUDA function calls.

Our study shows that the key bottleneck in popular heterogeneous GNN models (e.g., Relational GCN [7] and heterogeneous graph transformers (HGT) [8]) are node-type- or edge-type-specific linear projections (in algebraic term - dense-dense matrix multiplication). Dense matrix multiplications can consume up to 75% of the total training time in RGCN model (section II). To speed up this kernel, one solution used by the state-of-the-art libraries (PyG and DGL) is to first copy the weight matrices of different node/edge type to individual nodes/edges and use batched matrix-multiplication to perform type-specific matrix multiplication. We refer to this solution as high-mem. Even though this solution can speed up the training process, it leads to high memory consumption due to the duplication of weight matrices. An alternative solution is to sort the node or edge feature table according to their node or relation types as a pre-processing step and perform projections on each type separately. We refer to this solution as low-mem. low-mem does not require additional memory consumption, but requires additional sorting and separate function calls, both of which have significant overheads.

To address these issues, we propose two abstractions: 1) gather-mm- which performs a list dense matrix multiplications using a single invocation and looks up the corresponding weight matrices on demand, instead of copying them to nodes

or edges, and 2) segment-mm which uses cuBLAS library to perform the multiplications but avoids the Python invocation overhead. These operators allow us to implement heterogeneous graph models in a more compact way and develop more efficient implementations for these matrix multiplications.

To summarize our primary contributions:

- Analyze the workload of different heterogeneous graph models such as RGCN, HGT, RGAT.
- Propose two high-performing dense operators: gather-mm and segment-mm to accelerate the models.
- Formulate heterogeneous models such as RGCN and HGT using the proposed operators and demonstrate speedup in end-to-end training time. For full-graph training, we accelerate RGCN and HGT by $4.9\times$ over high-mem and by $4.5\times$ over low-mem. For mini-batch training, we accelerate RGCN by $1.3\times$ over high-mem and $2\times$ over low-mem.

Section II provides background on the existing solutions and their bottlenecks. Section III presents the analysis of heterogeneous graph computations. Section IV and V describe our proposed solutions gather-mm and segment-mm, respectively. Experimental evaluation against the state-of-the-art is presented in Section VI. Section VII summarizes the related work and Section VIII concludes with a summary of the findings and potential future work.

II. Background

Node- and edge-type specific operations are common in heterogeneous GNN models like RGCN, HGT, RGAT, etc. In this section, we take a look into two popular GNN models - RGCN and HGT. Let's first define a heterogeneous graph G as $G = (V, E, A, R)$ with nodes $v \in V$ and edges $e \in E$. R denotes the set of relation types and A is the set of node types. In RGCN, the forward pass to update a node embedding, as a part of the convolution process, is formulated as:

$$h_v^{(l+1)} = \sigma\left(\sum_{r\in R}\sum_{u\in N_u^r}\frac{1}{c_{r,u}}W_r^{(l)}h_u^{(l)}\right), \tag{1}$$

where h_u is the source node embedding, W_r is the learnable weight matrix of relation r, N_u^r is the neighboring nodes of u, $c_{r,u}$ is the normalization constant. The formula aggregates relation-specific linearly projected featured vectors from neighboring nodes. Figure 1a shows the time distribution between the matrix multiplication (denoted as matmul) and neighborhood aggregation (denoted as SpMM - sparse-dense matrix multiplication) on AIFB dataset using RGCN model. The models in Figure 1 are implemented using the DGL framework and utilize AWS EC2 g4 instance equipped with NVIDIA T4 GPUs. The matmul operations consumes 75% on average of the forward training time of low-mem signifying the impact of this operator.

The main architecture of HGT is composed of three components: 1) Heterogeneous Mutual Attention to compute the importance of each source node, 2) Heterogeneous Message Passing to compute messages on the source nodes, and 3) Target-Specific Aggregation to aggregate the neighborhood message by the attention weight.

Inspired by vanilla transformer model [9], HGT calculates the mutual attention between source node u and target node v by mapping v to a Query vector $Q^i(v)$ and u to a Key vector $K^i(u)$ and performing a dot product between them. To account the meta relation of the heterogeneous graph, HGT keeps a distinct edge-based matrix $W_{\phi(e)}^{ATT}$ for each edge type $\phi(e)$. For example, the mutual attention for i^{th} attention head on edge $e = (u, v)$ is computed as:

$$ATT^i(u, e, v) = K^i(u) \cdot W_{\phi(e)}^{ATT} \cdot Q^i(v)^T \tag{2}$$

$$K^i(u) = K - Linear^i_{\pi(u)}\left(h^{(l-1)}[u]\right) \tag{3}$$

$$Q^i(v) = Q - Linear^i_{\pi(v)}\left(h^{(l-1)}[v]\right) \tag{4}$$

In i^{th} attention head source node type $\pi(u)$ and target node type $\pi(v)$ are projected in the key and $query$ vector with linear projections. Also, in the target-specific aggregation step, we have another type-specific linear projection to map target node v's vector back to its type-specific distribution.

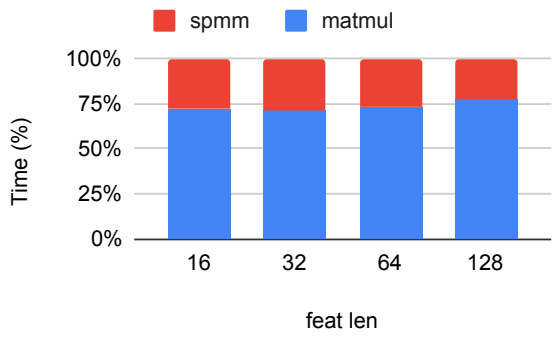
$$h_v^l = A - Linear_{\pi(v)}\left(\sigma(h^{(l)})[v]\right) + (h^{(l-1)}[v]) \tag{5}$$

The linear projection is followed by a non-linear activation and residual connection. Figure 1b shows the time distribution among its four major kernels: 1) matmul, 2) SpMM, 3) softmax computation, and 4) SDDMM (Sampled dense-dense matrix multiplication) on ACM dataset using 4 attention heads. On feature length 512, the matmul operator consumes 60% of the total time.

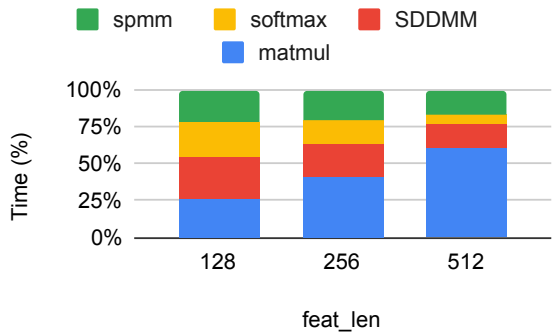
Type-wise linear projection is a common operator in many heterogeneous GNN models. In the next section, we discuss the characteristics of this dense operator and its usage in formulating GNN models in libraries such as DGL and PyG.

III. Analysis of heterogeneous graph computations

In models like RGCN, the relation-specific linear transformation is used to compute messages on the edges of a graph. As shown in Equation (1), each relation r performs a matrix multiplication between weight matrix W_r of shape $(D1 \times D2)$ and the embedding matrix of the source node type h of shape $(|V| \times D1)$. Here, $D1$ and $D2$ are the input and output feature lengths, respectively. The computation is followed by an aggregation across neighboring messages. When the graph is converted to a homogeneous one, these series of multiplications can be executed in two ways: 1) high-mem: use one batched matrix multiplications, bmm, which trades off performance with high memory consumption memory, 2) low-mem: sort and split the feature table according to relation types and perform series of multiplication.



(a) RGCN on AIFB dataset



(b) HGT on ACM dataset

Fig. 1: Training time (forward) breakdown.

To use the batched matrix multiplications, the high-mem version invokes only one python call, followed by a C++/CUDA kernel to perform the multiplication. However, to retrieve relation-specific weight matrix W_r, W_r is copied to each edge according to its edge type. The resulting tensor becomes of shape ($|E| \times D1 \times D2$), where $|E|$ is the total number of edges in the graph. The original weight matrix had a shape ($|R| \times D1 \times D2$). Copying the weight matrices to the edges consumes a significant amount of memory and time. Fitting a tensor of size $|E| \times D1 \times D2$ on a GPU memory is challenging and for larger datasets becomes an impractical solution.

The low-mem adopts a similar technique to heterogeneous model to address the memory issue. It sorts the edges according to edge types and launches separate kernels for each relation specific computation. The aggregation step across neighbors are still a one step method in the low-mem version. Although, this version resolves the memory issue, on a graph with many relation types, the overhead of sorting and $|R|$ times Python and CUDA kernel invocation becomes a bottleneck.

Table I shows the end-to-end training time of the high-mem and low-mem on the AIFB dataset. The timing includes sorting and copy time for low-mem and high-mem, respectively. For small feature length such as 8, high-mem outperforms low-mem by 3 $\times$ and requires only 12 MB extra memory. However, as feature length increases, the performance of high-mem version drops significantly - 5 $\times$ slower than low-mem for feature length 128, and also costs 3 GB extra memory.

feat_size	low-mem	high-mem	additional mem req. by high-mem
8	9.84	**3.44**	12 MB
16	9.78	**4.62**	48 MB
32	11.42	**8.47**	190 MB
64	**12.65**	26.62	761 MB
128	**20.00**	95.17	3,051 MB

TABLE I: Training time (ms) of RGCN on AIFB dataset: #nodes=7262, #edges=48810, #relations=104.

In this work, we address the high memory consumption issue of high-mem by introducing a new operator gather-mm, and avoid python invocation overhead by using operator segment-mm. In the next sections, we provide the technical details in developing gather-mm and segment-mm on GPUs.

IV. GATHER-MM

To avoid sorting in the low-mem version or consuming high memory in the high-mem version, in this operator we look up the weight matrices according to their relation type and perform projections accordingly. We extend Pytorch's $autograd$ engine and use our custom C++/CUDA operator for the forward and backward computations to compute the gradients. Prior to the operation, the node embeddings from h are copied to respective edges resulting a matrix of shape ($E \times D1$), where $D1$ is the feature length. We denote the resulting matrix as H.

A. *forward operator*

The forward operator of gather-mm is formulated as follows:

$$Out[i] = H[i] * W[etype[i]],$$

where $H \in \mathbb{R}^{|E| \times D1}$ and $W \in \mathbb{R}^{|R| \times D1 \times D2}$. The parameterized weight matrix, W_r is looked up by their relation type in $W[etype[i]]$.

Each node embedding vector $H[i] \in \mathbb{R}^{D1}$ is multiplied with its corresponding weight matrix $W[etype[i]] \in \mathbb{R}^{D1 \times D2}$, which is looked up by its relation type $etype[i]$. We denote this operator as *vector-indexed_matrix multiplication*. Figure 2 demonstrates the steps of the forward operator. The rows in H matrix looks up for the corresponding weight matrix in W and writes out to the out matrix.

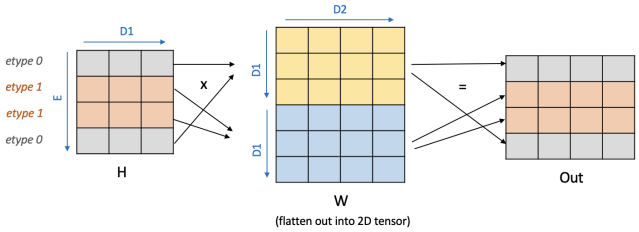


Fig. 2: Forward operator of gather-mm.

a) Vector-indexed_matrix multiplication:: Recent GPU architectures provide configurable L1/shared memory cache size. For example, V100 has 128KB/SM of shared memory which can be split between L1 cache and shared memory.

Shared memories are private to GPU thread-blocks but usually provide 100x faster memory access compared to global memory accesses (assuming no shared memory bank conflict). GPU registers are on-chip and are significantly faster than shared memory. However, registers are private to GPU threads and are also small in size (256KB/SM) in V100. Also, users must be careful not to over-exploit shared memory or not to spill registers, as it can lead to fewer thread block launches followed by low parallelization and poor performance.

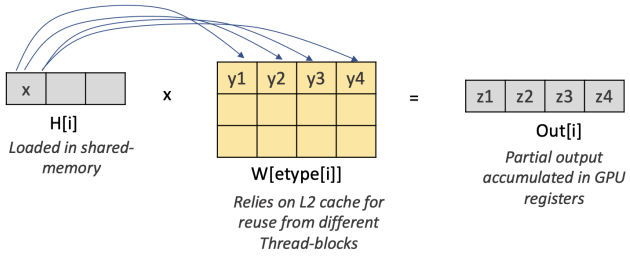


Fig. 3: Each element of H is loaded from shared memory and multiplied with a row of W to generate partial output in register.

To exploit the GPU's massive parallelism, we assign one warp (group of 32 threads) to perform one vector-matrix multiplication. Computing a dot product between the row of H and columns of W matrix would have provided the least number of memory write for the output. But matrices that are allocated by PyTorch have row-major layout. Hence, accessing W will be strided, which will be a waste of cache line (128 bytes on GPU) and result in un-coalescence memory accesses. Transposing W could be another easy but expensive option. Hence, we decide to exploit GPU registers, which can accommodate frequent writes without major performance loss. As illustrated in fig. 3, we kept the original layout of W, and generated partial output by multiplying one element of H and one row of W and added it to the registers. GPU threads processes elements of $H[i]$ in parallel in a block cyclic fashion. Once a warp completes the full computation of a row, it brings the output back from the register to the global memory.

We want to efficiently use GPU's memory hierarchy. Each thread-block (group of warps) loads a set of rows of H matrix into shared memory (thread-block private). Unlike H, W_r is looked up by indices. As we can not guarantee the reuse of a W_r within a thread-block, loading W to shared memory will not be beneficial. L2 cache (6MB in V100 GPU), on the other hand is shared across all thread-blocks and larger in size. We decided to rely on L2 for the accesses of W so that all the thread blocks looking up to the same W matrices can have the reuse advantage. Note that, this optimization technique is highly dependent on the feature length and the number of relations, as larger weight matrix increases the chance of eviction from cache.

By using the proposed method, we successfully utilized 80% of GPUs SM and memory usage in training RGCN.

Note that, here we are being specific to RGCN model. This operator can be extended to index the rows of H matrix as well.

B. backward operator

In the backward pass, we compute the gradients of H and W.

1) Computing H.grad: We can compute the gradients of H as follows:

$$H.grad[i] = Out.grad[i] \times W^T[etype[i]] \tag{6}$$

$H.grad$ can be computed using the same operator - vector-indexed_matrix multiplication as in the forward pass. An additional step is required to transpose the W matrix per relation type. We use cuBLAS's $xgeam$ operator to transpose the matrix.

2) Computing W.grad: The gradients of W is computed as follows:

$$W.grad[etype[i]] = H^T[i] \times Out.grad[i] \tag{7}$$

The computation of $W.grad$ can be formulated as a multiplication between a column vector ($H[i] \in \mathbb{R}^{1 \times D1}$) and row-vector ($Out[i] \in \mathbb{R}^{1 \times D2}$). The output is a matrix of dimension ($D1 \times D2$) which is looked up by the relation type. Figure 4 shows an overview of the computation.

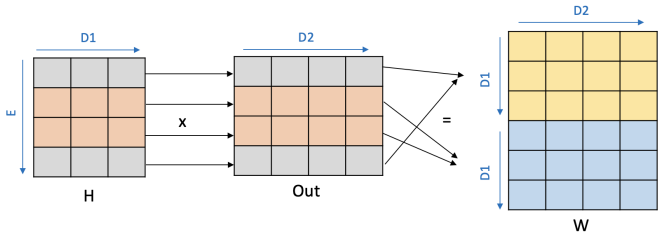


Fig. 4: Backward operator of gather-mm to compute $W[etype[i] = H^T[i] * Out[i]$. The output W_r is looked up.

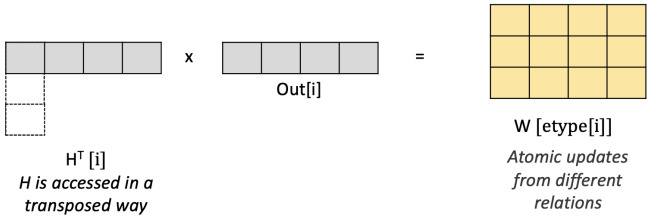


Fig. 5: vector-vector multiplication with indexed output. H is loaded into shared memory and accessed in a transposed way to amortize the strided accesses.

To avoid explicit transposition, we devised an algorithm to access H in a transposed way. This causes strided accesses (uncoalesced) in GPU memory which adversely affects the performance. This issue can be overcome by loading H into shared memory at the beginning as shared memory has negligible latency for random memory access. GPU warps

loads the rows of H in parallel, performs the multiplication and updates the output matrix W_r. As multiple warps might update the same weight matrix, to avoid write conflict, we use atomic adds.

V. Segment-MM

This operator is an optimization of DGL's low-mem version where each relation uses PyTorch's $matmul$ operator. We extend Pytorch's $autograd$ engine and use our custom C++ operator to loop over the relation avoiding python invocation overhead. The forward operator of segment_mm using matrix notation is formulated as

$$Out[segment_i] = H[segment_i] * W[i]$$

. Here, H is sorted and segmented according to relation types and can be processed as segmented multiplication with W_r. For each relation type, $H[i] \in \mathbb{R}^{|V_r| \times D1}$ multiplies a $W[etype[i]] \in \mathbb{R}^{D1 \times D2}$. Both the forward and backward operators of segment_mm can be implemented as a series of matrix-matrix multiplication. We use the NVIDIA cuBLAS library because of its high-performing solutions.

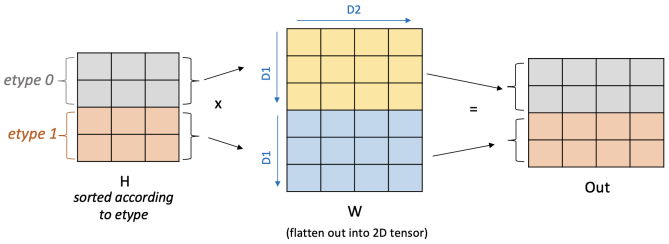


Fig. 6: Segmented matrix-matrix multiplication. H is sorted and each segment of H multiplies a segment of W.

The backward operators are computed as:

$$H.grad[segment_i] = Out.grad[segment_i] * W^T[i]$$
$$W.grad[i] = H^T[segment_i] * Out.grad[segment_i]$$

The transposition of the H and W matrix can be performed in two different ways: 1) using the in-built support to transpose inside the GEMM operator, 2) explicitly transposing the input using another operator Xgeam from cuBLAS. We benchmarked the implicit and explicit transpose cases and found 10%-40% better performance with implicit transposition.

VI. Experiments

Name	#nodes	#edges	#node-type	#relations
AIFB	7,262	48,810	7	104
MUTAG	27,163	148,100	5	50
BGS	94,806	672,884	27	122
AM	881,680	5,668,682	7	108

TABLE II: Properties of the graphs in the dataset.

a) Dataset:: We evaluate our model on four heterogeneous graphs: AIFB, MUTAG, BGS, and AM. The properties of graphs in our dataset can be found in Table II.

b) Experiment setup:: The experiments are executed on AWS EC2 g4 instances. The instances have NVIDIA Turing TU104 GPU which has 2560 CUDA cores, 320 Tensor Cores, and 16GB of GDDR6 GPU memory. T4 instances can provide up to 8.1 TFLOPS single-precision floating-point performance. Supported memory bandwidth is up to 320GB/s.

c) Benchmarks:: We have used DGL's open source code base to implement our proposed operators, segment-mm and gather-mm. We collect the end-to-end training time on RGCN and HGT models using various heterogeneous graph on varying length of features. The timings are bench-marked against the official examples of popular GNN libraries, PyG and DGL. We use 16 as the hidden dimension size in all the datasets. The training time includes forward pass, loss computation, backward pass, and sampling time when minibatch is used. All the results are reported based on the averages of 100 runs.

A. Training time on full batch

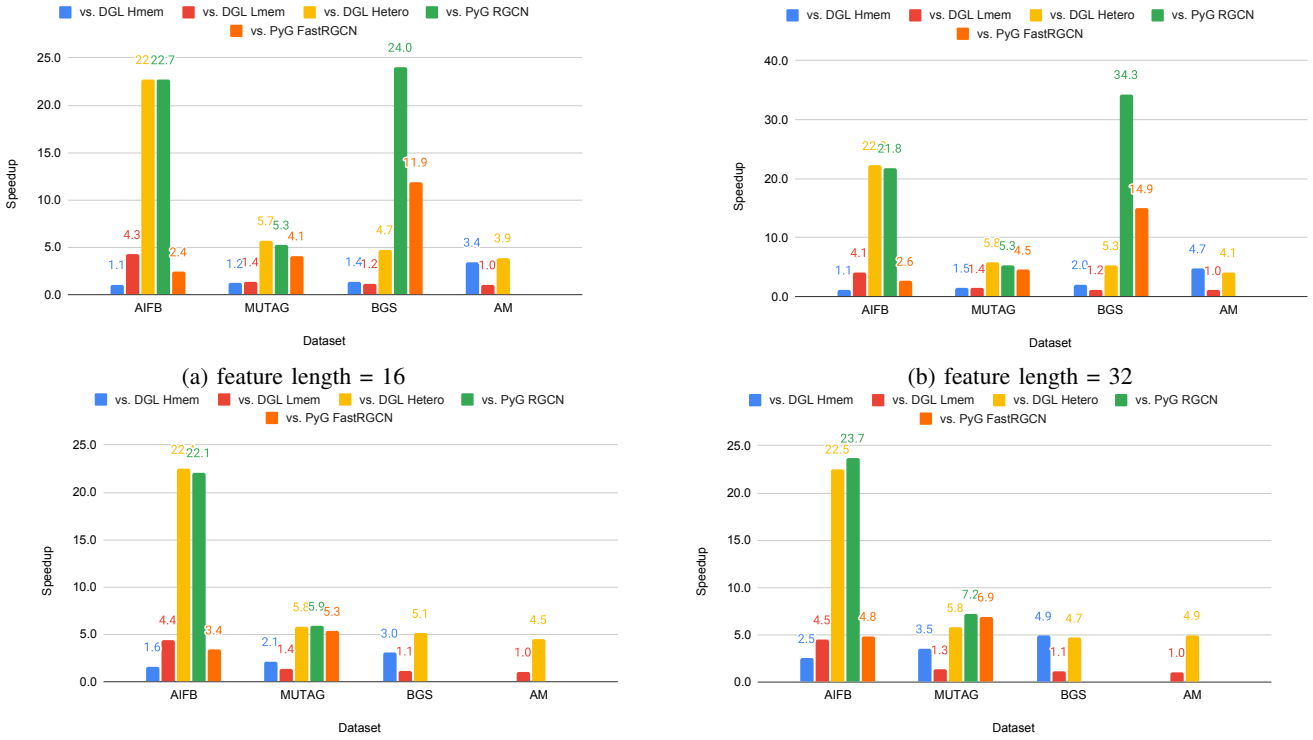
1) RGCN: Both PyG and DGL libraries maintain two versions of RGCN implementation - high-mem and low-mem. For the DGL versions, we denote them as DGL Hmem and Lmem, respectively. Similarly, the ones from PyG are denoted as RGCN and FastRGCN. We also include the benchmark when the model is directly applied to the original heterogeneous graph - DGL-Hetero. Figure 7 shows the speedup against the benchmarks for feature lengths 16, 32, 64, and 128. We compute the speedup against the best performing kernel between gather-mm and segment-mm.

Our proposed solution outperforms the SOTA implementations of the RGCN models by PyG and DGL libraries. A maximum speedup of $22.7\times$ over DGL-Hetero, $4.5\times$ and $4.9\times$ for DGL Lmem and Hmem, and $34.3\times$ and $14.9\times$ against PyG and PyG FastRGCN is achieved for AIFB dataset. AIFB has 107 relation types with only 48K edges (450 edges on average per relation). For such small dataset, the python invocation overhead is usually the key bottleneck. As the proposed solutions avoid the large number of invocation overhead, smaller dataset like AIFB gains significant advantage. For feature length=32, we outperform PyG's FastRGCN by $2.6\times$, $4.5\times$ and $14.9\times$ for AIFB, MUTAG, and BGS datasets respectively.

2) HGT: We compare the training time of HGT with the official examples provided by PyG 2.0 and DGL. Note that, DGL and PyG directly use heterogeneous graph to model HGT. We convert the graph to a homogeneous one and use homogeneous APIs to train the model while preserving the model accuracy.

In HGT, the workload increases in proportion to the number of attention heads. Figure 8a and Figure 8b shows the speedup for feature length 16 and 32 with one attention head. On AIFB dataset, we observe $30\times$ and $13\times$ speedup respectively against DGL and PyG examples for feature length 16. We observe similar speedup for larger feature length such as 64 and 128.

Figure 8c and Figure 8d show the speedup using 4 attention heads using feature length 16 and 32. It outperforms DGL and PyG by up to $12\times$ and $4\times$, respectively on AIFB. For



(a) feature length = 16

(b) feature length = 32

(c) feature length = 64

(d) feature length = 128

Fig. 7: Speedup in RGCN training time. The missing bars denote out-of-memory computation of the benchmark.

smaller number of attention head, the computations are small and GPU kernels are under utilized which results in larger speedups when our proposed operators are used.

B. Training time on minibatch

In order to overcome the memory constraints that arise during the training of GNN models on large graphs, DGL offers graph-sampling-based minibatch training. This method entails feeding a subgraph, sampled from the original graph, into the GNN model at each step to compute the gradients of the model's weight parameters. The size of the subgraph is controlled by two parameters: the *fan out* and the *batch size*. The *fan out* refers to the number of neighbors to be sampled for each vertex in each layer of the GNN model, while the *batch size* represents the number of seed nodes in the subgraph.

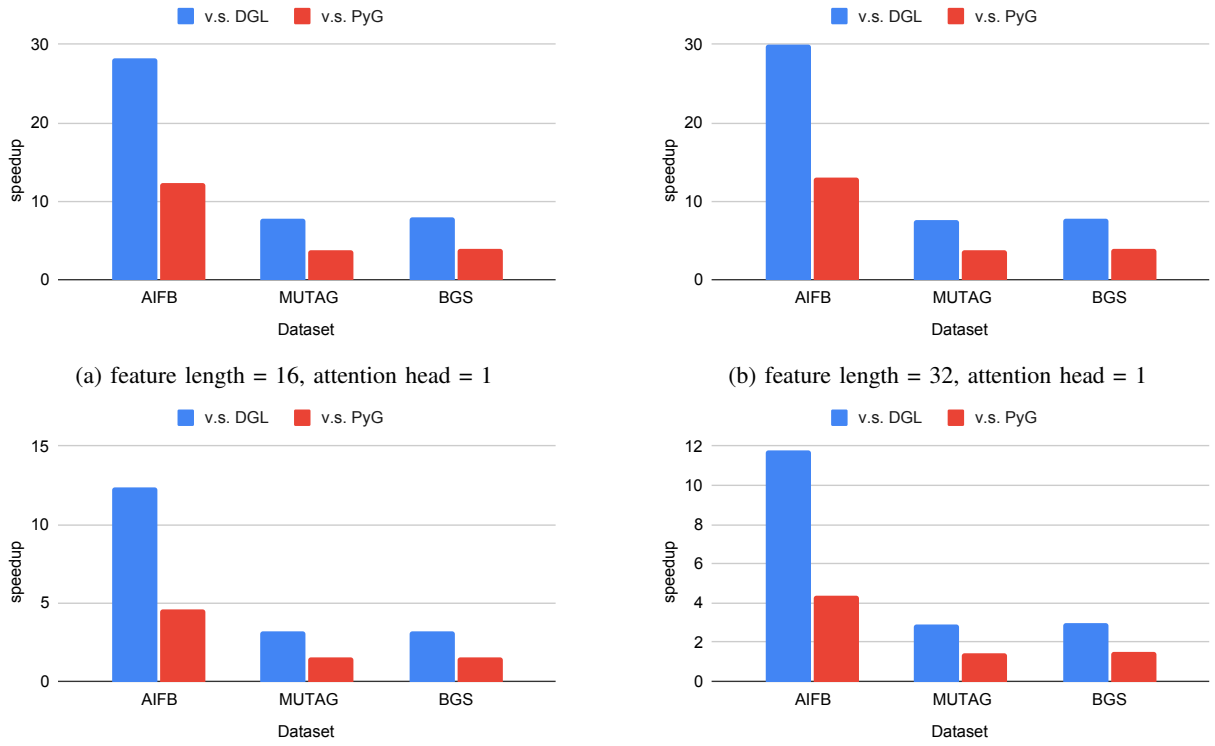
To demonstrate the effectiveness of our solution on the minibatch training scenario, Figure 9 shows the end-to-end minibatch RGCN training time on AM dataset with $fan\ out = [16, 16]$ and $batch\ size = 4096$. One can observe that on average, the performance of gather-mm is 1.3× (up to 1.74×) faster compared to high-mem, and segment-mm demonstrates an even greater improvement, with an average speed increase of 1.9× (up to 2.0×) times compared to low-mem.

It is noteworthy that a significant proportion of the execution time during minibatch training is dedicated to the sampling of the input graph. To provide a comprehensive evaluation of the training process, we have obtained the execution time breakdowns for minibatch RGCN training, as depicted in Table III. As can be observed from Table III, the graph sampling process accounts for approximately 4.5 ms in the execution time of all four implementations, constituting up to 45% of the total execution time. However, when excluding the graph sampling time, it can be seen that the performance improvement of our solution is even more significant compared to the end-to-end training time. Specifically, gather-mm shows an average speedup of 1.5× (up to 2.3×) over high-mem and segment-mm demonstrates an average speedup of 2.4× (up to 2.5×) over low-mem. Therefore, it can be inferred that for larger datasets such as OGBN-MAG, where the graph sampling process only accounts for 21% of the execution time, the advantages of our approach will become even more pronounced.

C. Performance comparison of segment-mm and gather-mm

The performance of gather-mm relies on the capacity GPU's L2 cache and the reuse scopes of W_r matrices. When the same W_r is accessed by different rows of H and the shape of W_r is small enough to fit multiple W_r in the L2 cache, gather-mm achieves higher L2 cache hit rate and outperforms segment-mm. On the other hand, segment-mm launches separate CUDA kernels for different relations. Hence, for larger W_r, segment-mm is often the preferable option as the CUDA kernel launch overhead becomes negligible and it can harness GPU power to process each relation separately.

Table IV shows the pattern on AIFB and MUTAG dataset. AIFB and MUTAG has 450 and 2962 edges on average per



(a) feature length = 16, attention head = 1

(b) feature length = 32, attention head = 1

(c) feature length = 16, attention head = 4

(d) feature length = 32, attention head = 4

Fig. 8: Speedup in HGT training time.

Size	high-mem			low-mem			gather-mm			segment-mm		
	Sampling	Fwd	Bwd	Sampling	Fwd	Bwd	Sampling	Fwd	Bwd	Sampling	Fwd	Bwd
8	4.5	3.7	1.7	4.5	9.3	11.1	4.7	3.5	1.1	4.5	5.8	3.4
16	4.4	3.9	2.0	4.6	9.4	11.0	4.7	3.5	1.3	4.5	5.8	3.4
32	4.4	4.4	2.4	4.7	9.6	11.5	4.6	3.5	1.4	4.5	6.0	2.7
64	4.3	5.4	3.3	4.7	9.8	11.7	4.6	3.9	1.8	4.5	6.2	2.6
128	4.3	7.9	5.7	4.5	10.0	12.3	4.3	4.0	2.0	4.5	6.5	2.4

TABLE III: The execution time (ms) breakdowns, i.e., Sampling, Forward (Fwd), and Backward (Bwd), of the minibatch RGCN training on AM dataset across different sizes of feature (8-128) with a *fan out* of [16, 16] and a *batch size* of 4096.

Size	AIFB		MUTAG	
	gat-mm	seg-mm	gat-mm	seg-mm
8	4.3	8.6	31.9	39.4
16	4.7	8.5	39.0	31.7
32	4.9	8.2	40.2	29.0
64	4.8	8.4	44.0	30.4
128	4.9	8.6	52.9	33.9

TABLE IV: comparison of training time using segment-mm and gather-mm on varying feature size (8-128).

relation. On AIFB dataset, gather-mm consistanly outperform segment-mm. However, for MUTAG dataset, with larger feature length segment-mm is the preferable choice. In fig. 7 we empirically choose the best operator between segment-mm and gather-mm to demonstrate the comparison against DGL and PyG library.

VII. Related Work

In recent years, there has been significant effort from academia and industry to accelerate GNN models. The main computational kernels of GNN models are sparse kernels such as sparse-dense matrix multiplication (SpMM) and sampled dense-dense matrix multiplication (SDDMM), as well as dense kernels such as dense-dense matrix multiplication (GEMM). Most of the previous work have focused on optimizing the SpMM [10], [11] and SDDMM [12] kernels, primarily because of their challenges to accelerate on massively parallel architectures like GPU. Due to the availability of highly efficient dense operator from NVIDIA's cuBLAS and Intel's MKL library, dense computation on homogeneous graph received less attention from the community. Our study on heterogeneous GNN models signifies the impact of irregular dense operators. Recently, PyG-lib which is a low-level GNN have

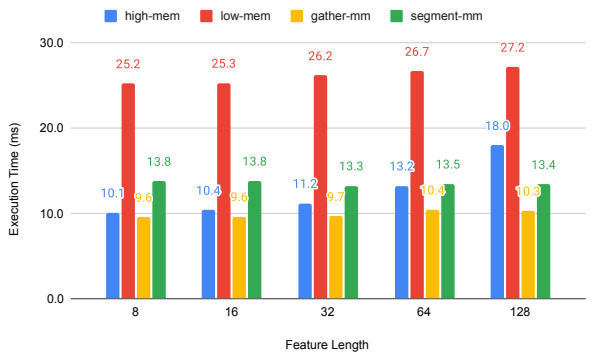


Fig. 9: End-to-end minibatch RGCN training time on AM dataset with a *fan out* of [16, 16] and a *batch size* of 4096.

integrated NVIDIA CUTLASS library to provide the type specific linear projections. PyG-lib is integrated with PyG [4] (>2.2 version). DGL (Deep Graph Library) is a framework agnostic GNN library which extends PyTorch [13], MXNET [14], TensorFlow [15] to support GNN models. It can train billion-scale graphs using its distributed training infrastructure and compatibility with multi-GPU training. We have integrated our solutions with DGL framework to speed up GNN training process. TFGNN [5] is another GNN library based on TensorFlow framework. TFGNN has delivered promising results on heterogeneous GNN model.

A few compiler and runtime framework like GNNAdvisor [16], SparseTIR [17], Graphiler [18], and PIGEON [19] have achieved significant speed up in end to end training pipeline.

VIII. Conclusion

In this work, we have addressed one common bottleneck in heterogeneous GNN models - type specific linear projections. Through experiments we have demonstrated that the proposed solutions can outperform SOTA implementations by PyG and DGL libraries on popular models like RGCN and HGT.

We want to further improve the gather_mm operator by introducing tiling. We also want to investigate the next bottleneck in heterogeneous models.

References

[1] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, "Translating embeddings for modeling multi-relational data," *Advances in neural information processing systems*, vol. 26, 2013.

[2] X. Wang, H. Ji, C. Shi, B. Wang, Y. Ye, P. Cui, and P. S. Yu, "Heterogeneous graph attention network," in *The world wide web conference*, 2019, pp. 2022–2032.

[3] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, 2019, pp. 793–803.

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

[5] O. Ferludin, A. Eigenwillig, M. Blais, D. Zelle, J. Pfeifer, A. Sanchez-Gonzalez, S. Li, S. Abu-El-Haija, P. Battaglia, N. Bulut *et al.*, "Tf-gnn: Graph neural networks in tensorflow," *arXiv preprint arXiv:2207.03522*, 2022.

[6] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *ICLR workshop on representation learning on graphs and manifolds*, 2019.

[7] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," in *European semantic web conference*. Springer, 2018, pp. 593–607.

[8] Z. Hu, Y. Dong, K. Wang, and Y. Sun, "Heterogeneous graph transformer," in *Proceedings of The Web Conference 2020*, 2020, pp. 2704–2710.

[9] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/9d63484abb477c97640154d40595a3bb-Paper.pdf

[10] C. Hong, A. Sukumaran-Rajam, B. Bandyopadhyay, J. Kim, S. E. Kurt, I. Nisa, S. Sabhlok, Ü. V. Çatalyürek, S. Parthasarathy, and P. Sadayappan, "Efficient sparse-matrix multi-vector product on gpus," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, 2018, pp. 66–79.

[11] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 300–314.

[12] I. Nisa, A. Sukumaran-Rajam, S. E. Kurt, C. Hong, and P. Sadayappan, "Sampled dense matrix multiplication for high-performance machine learning," in *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE Computer Society, 2018, pp. 32–41.

[13] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, 2019.

[14] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[15] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: a system for large-scale machine learning." in *Osdi*, vol. 16, no. 2016. Savannah, GA, USA, 2016, pp. 265–283.

[16] 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 *15th USENIX symposium on operating systems design and implementation (OSDI 21)*, 2021.

[17] Z. Ye, R. Lai, J. Shao, T. Chen, and L. Ceze, "Sparsetir: Composable abstractions for sparse compilation in deep learning," *arXiv preprint arXiv:2207.04606*, 2022.

[18] Z. Xie, M. Wang, Z. Ye, Z. Zhang, and R. Fan, "Graphiler: Optimizing graph neural networks with message passing data flow graph," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 515–528, 2022.

[19] K. Wu, M. Hidayetoğlu, X. Song, S. Huang, D. Zheng, I. Nisa, and W.-m. Hwu, "Pigeon: Optimizing cuda code generator for end-to-end training and inference of relational graph neural networks," *arXiv preprint arXiv:2301.06284*, 2023.