fuseGNN: Accelerating Graph Convolutional Neural Network Training on GPGPU

Zhaodong Chen
University of California, Santa Barbara
chenzd15thu@ucsb.edu

Mingyu Yan
University of California, Santa Barbara
yanmingyu@ict.ac.cn

Maohua Zhu
Lei Deng
University of California, Santa Barbara
maohuazhu@ece.ucsb.edu
leideng@ucsb.edu

Guoqi Li
Tsinghua University
liguoqi@mail.tsinghua.edu.cn

Shuangchen Li
Alibaba Group
shuangchen.li@alibaba-inc.com

Yuan Xie
University of California, Santa Barbara
yuanxie@ece.ucsb.edu

ABSTRACT

Graph convolutional neural networks (GNN) have achieved state-of-the-art performance on tasks like node classification. It has become a new workload family member in data-centers. GNN works on irregular graph-structured data with three distinct phases: *Combination*, *Graph Processing*, and *Aggregation*. While *Combination* phase has been well supported by *sgemm* kernels in cuBLAS, the other two phases are still inefficient on GPGPU due to the lack of optimized CUDA kernels. In particular, *Aggregation* phase introduces large volume of DRAM storage footprint and data movement, and both *Aggregation* and *Graph Processing* phases suffer from high kernel launching time. These inefficiencies not only decrease training throughput but also limit users from training GNNs on larger graphs on GPGPU. Although these problems have been partially alleviated by recent studies, their optimizations are still not sufficient. In this paper, we propose *fuseGNN*, an extension of PyTorch that provides highly optimized APIs and CUDA kernels for GNN. First, two different programming abstractions for *Aggregation* phase are utilized to handle graphs with different average degrees. Second, dedicated GPGPU kernels are developed for *Aggregation* and *Graph Processing* in both forward and backward passes, in which kernel-fusion along with other optimization strategies are applied to reduce kernel launching time and latency as well as exploit data reuse opportunities. Evaluation on multiple benchmarks shows that *fuseGNN* achieves up to 5.3× end-to-end speedup over state-of-the-art frameworks, and the DRAM storage footprint is reduced by several orders of magnitude on large datasets.

1 INTRODUCTION

In recent years, graph convolutional neural networks (GNN) that operate on graph-structured data have achieved convincing performance on tasks like node and graph classification [10, 23, 24]. Similar to other computation-intensive workloads [2, 16, 25], GNNs are usually trained on the Graphics Processing Unit (GPU) that has high programmability and rich computation resources. Therefore, developing an efficient framework for GNNs on GPU is important.

GNN has three distinct phases: *Combination*, *Graph Processing*, and *Aggregation*. *Combination* is usually a single- or multi-layer perceptron that updates feature vectors of each vertex. *Graph Processing* processes the graph to be used in *Aggregation*. It usually involves tasks like computing degree, updating or generating edge weights, and converting graph between different sparse representations. The exact execution flow varies in different GNN algorithms. *Aggregation* updates each feature vector by aggregating its neighbor feature vectors with some aggregators like max and sum [11].

While *Combination* phase is well supported by *sgemm* (Single precision General Matrix Multiply) kernels in cuBLAS [14], the other two phases implemented with current APIs in PyTorch or Tensorflow are far from efficient. We profile Graph Convolutional Network (GCN) [10] implemented with PyTorch Geometric (*PyG*) [3]. The result shows that on one hand, *Aggregation* phase introduces high volume of DRAM storage footprint and data movement. For example, dataset Reddit [6] has 114 million edges. With feature-length 128, the intermediate matrix takes up 58 GB, which is impossible to fit in a single GPU. Besides, over 300 GB data will be read and written between GPU cores and off-chip memory including 60 GB *atomic* transactions. On the other hand, the kernel launching time could take up to 85% of the whole execution time due to the complex execution flow. All in all, dedicated GPU kernels and APIs for *Graph Processing* and *Aggregation* in both forward and backward passes are required to develop an efficient GNN framework on GPU.

In recent years, several studies have been proposed to speedup *Aggregation* phase in two aspects. First, the *Gather-ApplyEdge-Scatter* (GAS) abstraction used in *PyG* is replaced with *Gather-ApplyEdge-Reduce* (GAR) abstraction to replace atomic reduction with non-atomic reduction in shared memory or registers. Second, due to the large variety of GNN algorithms, it is currently impossible to develop a kernel than optimizes all the GNN models. As a result, most of these frameworks choose to speedup one common scenario: the edge weight is scalar and gradient doesn't flow through the edge weights (e.g. Graph Convolutional Network (GCN) [10] and Graph

ICCAD '20, November 2–5, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8026-3/20/11.
https://doi.org/10.1145/3400302.3415610

Isomorphism Network (GIN) [22]). For instance, *neuGraph* [12] develops a kernel called *Fused-Gather* and Deep Graph Library (*DGL*) [20] implements the *Aggregation* phase with sparse matrix-matrix multiplication (SpMM) supported by *cuSPARSE* library [13].

However, these optimizations are not sufficient due to following reasons. First of all, the GAR abstraction requires the Compressed Sparse Row (CSR) format graph in forward pass and Compressed Sparse Column (CSC) in backward pass, while GAS abstraction can work on unsorted Coordinate List (COO) format graphs. On graphs with low average degree, the execution time saved by using GAR abstraction in *Aggregation* is offset by the format conversion overhead, so GAR is not always the best choice. Second, many GNN algorithms with attention mechanisms (e.g. Graph Attention Network [18]) have been proposed. In these algorithms, the edge weight is calculated based on the feature vector of source and target vertices as well as some trainable attention parameters. As a result, the gradient will flow through the edge weight, which is not supported by previous studies. Last but not least, neglected by previous studies, we find that accelerating the *Graph Processing* phase is critical for accelerating GNN training on small graphs. With these observations, in this paper, we propose *fuseGNN*, a highly-optimized extension of PyTorch for GNNs on GPU. Our key contributions are summarized below.

- **Dual Aggregation Models**: in *Aggregation* phase, we demonstrate that when considering all overhead, GAS should be applied to graphs with low average degree and GAR is a better choice for graphs with higher average degree.
- **Efficient CUDA kernels**: we develop dedicated CUDA kernels for *Aggregation* and *Graph Processing* phases in both forward and backward passes, in which multiple optimization strategies like kernel fusion are applied to reduce kernel launching time as well as exploit possible data reuse opportunities to reduce DRAM storage footprint and redundant data movement.

We evaluate our *fuseGNN* on multiple benchmarks and compare it with the state-of-the-art frameworks including *PyG* [3], *DGL* [20], and *neuGraph* [12]. It achieves up to 5.3× end-to-end training speedup over *PyG*, and the DRAM storage footprint is reduced by nearly 500× on Reddit dataset. Our *fuseGNN* makes it possible to train GAT on entire Reddit with a single NVIDIA V100 GPU. Our codes are publicly available at https://github.com/apuaaChen/gcnLib.

2 BACKGROUND AND RELATED WORK

We first present an overview on the background and related studies including the graphs, GNN, and existing frameworks for GNNs.

2.1 Graph

A graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of two parts: vertices and edges. Let N_v and N_e be the number of vertices and edges, respectively. Each vertex $v_i \in \mathcal{V}$ has a feature vector $\boldsymbol{x_i} \in \mathbb{R}^{1\times m}$, and the feature vectors are organized as a feature matrix $\boldsymbol{X} \in \mathbb{R}^{N_v\times m}$. Each edge $(v_i, v_j) \in \mathcal{E}$ can be directed or undirected, and may also have a feature e_{ij}. Table 1 summarizes four popular datasets for GNN. The "×2" under "#Edge" suggests that the edge is undirected. As illustrated in Figure 1 (a)&(b), the edges can be formulated as a sparse adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N_v\times N_v}$. The row and column index

Table 1: Dataset information

Dataset	#Vertex	Feature Len.	#Edge	Avg. Degree
Cora (CR)	2,708	1,433	5,429 × 2	4.0
Citeseer (CS)	3,327	3,703	4,732 × 2	2.8
Pubmed (PB)	19,717	500	44,338 × 2	4.5
Reddit (RD)	232,965	602	114,615,892	492

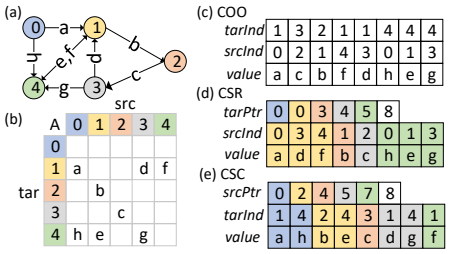

Figure 1: Graph Representations. (a) Graph; (b) Adjacent matrix; (c) COO format; (d) CSR format; (e) CSC format.

of each entry identify the target and source vertex, respectively. There are three formats for the sparse matrix (Figure 1 (c), (d), (e)):

COO (Coordinate list): edges are stored in three one-dimensional arrays including *tarInd*, *srcInd*, and *value*. The corresponding entries in *tarInd* and *srcInd* mark the coordinate of the non-zero entry and *value* is its value.

CSR (Compressed Sparse Row): edges are stored in three one-dimensional arrays including *tarPtr*, *srcInd*, and *value*. Each entry in *tarPtr* encodes the index in *srcInd* and *value* where the given row starts, while *srcInd* and *value* encodes the column index and value of each non-zero entry, respectively.

CSC (Compressed Sparse Column): this is similar to CSR, edges are stored in three one-dimensional arrays including *srcPtr*, *tarInd*, and *value*. Each entry in *srcPtr* encodes the index in *tarInd* and *value* where the given row starts, while *tarInd* and *value* encodes the row index and value of each non-zero entry, respectively.

2.2 GNN Models

Most GNN models follow neighborhood aggregation strategy [22]. Let the input feature vector of vertex v at layer k be $\boldsymbol{h}_v^{(k-1)}$ (In the first layer, $\boldsymbol{h}_v^{(0)} = \boldsymbol{x}_v$), the k-th layer of GNN is formulated as

$$\widetilde{\boldsymbol{h}_i^{(k-1)}} = MLP^{(k)}\left(\boldsymbol{h}_i^{(k-1)}\right), \boldsymbol{h}_v^{(k)} = AGG^{(k)}\left(\left\{\widetilde{\boldsymbol{h}_{u\in\mathcal{N}(v)}^{(k-1)}}, \widetilde{\boldsymbol{h}_v^{(k-1)}}\right\}\right), \tag{1}$$

where $\mathcal{N}(v)$ is a set of nodes adjacent to vertex v, $AGG^{(k)}$ is the aggregator in layer k. We take GCN [10] and GAT [18] as examples.

GCN. In GCN, each edge can have an initial scalar edge weight e_{ij} provided by the dataset. The degree of vertex i is defined as $d_i = \sum_j e_{ij}$. And the final edge weight used for aggregation is

$$\widetilde{e_{ij}} = \frac{e_{ij}}{\sqrt{(d_i+1)(d_j+1)}}. \tag{2}$$

The aggregator of GCN is as follows:

$$\boldsymbol{h}_v^{(k)} = \sum_{u\in\mathcal{N}(v)\cup\{v\}} \widetilde{e_{vu}} \otimes \widetilde{\boldsymbol{h}_u^{(k-1)}}, \tag{3}$$

where $\otimes$ is element-wise multiplication.

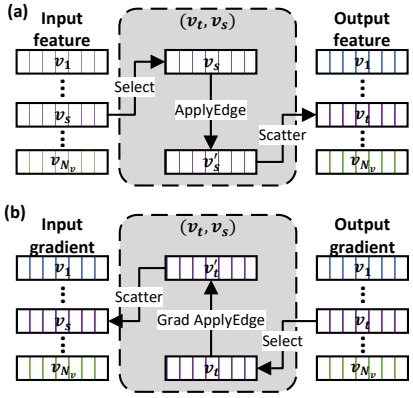

Figure 2: GAS Abstraction. (a) Forward; (b) Backward.

GAT. GAT generates edge weights as follows:

$$\widetilde{e_{ij}} = dropout\left(\frac{exp\left(lReLU\left([\widetilde{\boldsymbol{h}_i^{(k-1)}}||\widetilde{\boldsymbol{h}_j^{(k-1)}}]\boldsymbol{a}^{(k)}\right)\right)}{\sum_{q\in\mathcal{N}_i} exp\left(lReLU\left([\widetilde{\boldsymbol{h}_i^{(k-1)}}||\widetilde{\boldsymbol{h}_q^{(k-1)}}]\boldsymbol{a}^{(k)}\right)\right)}\right), \tag{4}$$

where $\boldsymbol{a}^{(k)}$ is a trainable attention vector, $||$ is the concatenate operator. GAT uses the same rule in Equation (3) for *Aggregation*.

The GNNs can be trained in two settings: transductive learning and inductive learning [6]. The former one trains the network on a fixed graph and generalizing to unseen data is not required. In inductive learning, in order to generalize to unseen nodes and graphs, the network is trained on a different graph in each iteration.

2.3 Previous GNN Frameworks

To speedup GNN training on GPU, several frameworks have been proposed in recent years. As *Graph Processing* has much less intensive computation and data access compared with the other two phases, most existing frameworks are developed in a two-phase abstraction: *Combination-Aggregation*. For former one, *sgemm* kernel in *cuBLAS* library is applied as it is efficient enough. For *Aggregation* phase, existing frameworks take one of the following abstractions.

***Gather-ApplyEdge-Scatter* (GAS)**. GAS takes an unsorted COO format Graph and traverses all the edges within the edge list. As shown in Figure 2 (a), for edge (v_t, v_s) where v_t is the target vertex and v_s is the source vertex, the feature vector of v_s is selected from the input feature matrix. After applying the edge weight, the result is scattered to the corresponding row v_t of the output feature matrix. The backward pass is illustrated in Figure 2 (b). For each edge (v_t, v_s), the gradient on output feature vector of v_t is taken. It goes through the backward pass of the *ApplyEdge* function, and scattered to the gradient matrix of input feature matrix. The *Grad ApplyEdge* function will also generate the gradient of other operands in *ApplyEdge*, e.g., edge weight, if necessary.

PyTorch Geometric (*PyG*) [3] is basically implemented with the GAS Model. It first stacks the source feature vectors into an $N_e \times m$ intermediate feature matrix with *indexSelect* API in PyTorch, then runs the *ApplyEdge* function on the intermediate feature matrix.At last it uses a self-defined *scatter* API to generate the output features. While *indexSelect* and *Scatter* are the backward pass of each other,

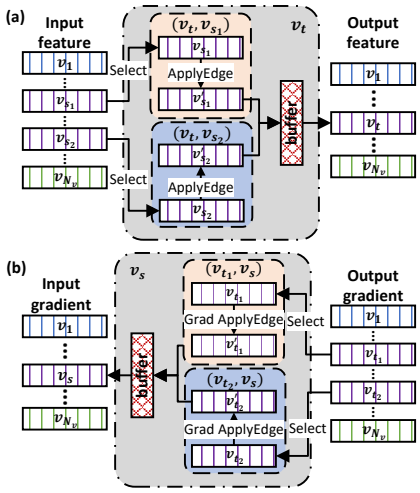

Figure 3: GAR Abstraction. (a) Forward; (b) Backward.

the *ApplyEdge* is defined in PyTorch native functions so that its backward pass is automatically handled.

***Gather-ApplyEdge-Reduce* (GAR)**. In forward pass, GAR works on the CSR format Graphs where the rows indicate target vertex. Hence, all the edges with the same target vertex are contiguous in the edge list. For each target vertex v_t, all its incoming edges are traversed. Figure 3 (a) illustrates the forward pass of GAR model when v_t has two incoming edges. For each incoming edge (v_t, v_{s_i}), the procedure is similar to GAS model. The only difference is that instead of scattering the result to output feature matrix, the output of *ApplyEdge* is reduced in an on-chip buffer (registers or user-managed data cache). After all the incoming edges are reduced, the content of the buffer is written to the output feature matrix in DRAM. In backward pass, CSC format of the graph is required, so that all the edges with the same source vertex are contiguous in the edge list. As illustrated in Figure 3 (b), all the outgoing edges are traversed. For each edge (v_{t_i}, v_s), the gradient of v_{t_i} is selected, it goes through the *Grad ApplyEdge* function, and the result is reduced to on-chip buffer. *Grad ApplyEdge* function generates the gradient of other operands in *ApplyEdge* just like GAS. At last, the content in the buffer is written to gradient matrix of input features.

NeuGraph [12] and Deep Graph Library (*DGL*) [20] choose GAR model. During forward pass, the source feature vectors are first stacked to in an $N_e \times m$ intermediate feature matrix and the *ApplyEdge* is executed, which is similar to *PyG*. Because the graph is stored in CSR format, the *Reduce* stage just slices continuous rows in the intermediate feature matrix that share the same target vertex and do the reduction with a custom CUDA kernel. A similar procedure is taken for backward pass.

When *ApplyEdge* is just element-wisely multiplying with edge weight, the forward pass is further optimized with kernel fusion: *DGL* directly uses sparse-dense matrix multiplication under CSR format, and *neuGraph* implements a *Fused-Gather* kernel that fuses *Gather-ApplyEdge-Reduce* in a single CUDA kernel. However, these optimizations don't support gradient on edge weight.

3 CHARACTERIZING GNNS ON GPU

In this section, we characterize GNN training workload on GPU. We run the forward and backward pass of a single layer GCN [10]

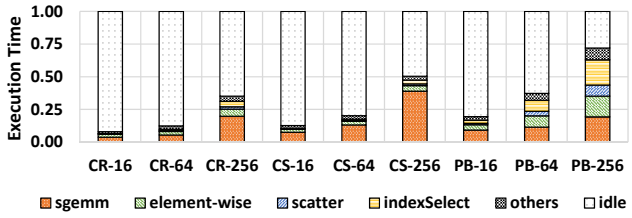

Figure 4: Execution time breakdown on V100 GPU.

on single NVIDIA V100 GPU. The input feature dimension is the native feature length of each dataset, and the output dimension is chosen from {16, 64, 256} to cover various situations.

Execution Time Breakdown. Figure 4 illustrates the execution time breakdown of GCN on Cora (CR), Citeseer (CS), and Pubmed (PB). The detailed information on these datasets is summarized in Table 1. First, when the dimension is small (e.g. 16), the GPU is idled for more than 85% of execution time. The reason behind that is the complex execution flow of GCN (especially in *Graph Processing* Phase) invokes handful small kernels. The profiling result shows that the launching time of each kernel on host (CPU) is even longer than the execution time on GPU. The GPU idle problem is less severe with higher hidden dimension (e.g. 256) or on larger graphs (e.g. Pubmed), as the execution time on *Combination* and *Aggregation* phases increases drastically, and the kernel launching time can overlap with them.

Data Movement. Here we mainly focus on 3 major kernels in forward pass of *Aggregation* phase: *indexSelect*, *elementwise*, and *scatterAdd*, as the backward pass is symmetric in GCN. We use Pubmed with hidden dimension 256 as benchmark, which has 88,676 edges, 19,717 vertices, and each feature vector takes 1 KiB.

Table 2: Data Movement (*: atomic transaction)

Kernel	L2 $ Read	L2 $ Write	DRAM Read	DRAM Write
indexSelect	90.7 MiB	86.6 MiB	58.2 MiB	86.3 MiB
elementwise	87.7 MiB	86.6 MiB	87.1 MiB	86.2 MiB
scatterAdd	87.9 MiB	90.77* MiB	142.1 MiB	58.3 MiB

In *indexSelect*, the kernel stacks the source feature vectors of all edges into a huge extended feature matrix and writes it back to DRAM. The *indexSelect* is executed after the *Combination* phase, so at most 6.144 MiB out of 20.2 MiB (30%) of feature vectors can be cached. As a result, while *indexSelect* requires 90 MiB data, 2/3 of which will be loaded from DRAM. This matches the total 58.2 MiB read from DRAM in our profiling result. When writing the extended feature matrix back, 88,676 edges take up 88 MiB DRAM.

In *elementwise* kernel, the extended feature matrix is loaded to multiply with the edge weight and written back. The L2 cache fails due to the long reuse distance, and the cache hit rate is almost 0.

In *scatterAdd* kernel, the extended feature matrix is loaded again, then atomically scattered to the corresponding target feature vector. When a kernel issues an atomic request, the request is transmitted to a tag look-up unit to check whether the corresponding data is cached. If it is not cached, then the data will be loaded from DRAM to L2 cache. Then the atomic command is executed with an arithmetic logic unit (ALU) residing external to the L2 cache [5]. As a result, for each edge, two feature vectors will be loaded, and 176 MiB data are required. However, as 2/3 of the target feature vectors can be cached, the final DRAM Read should be 146 MiB, this also matches the profiling result.

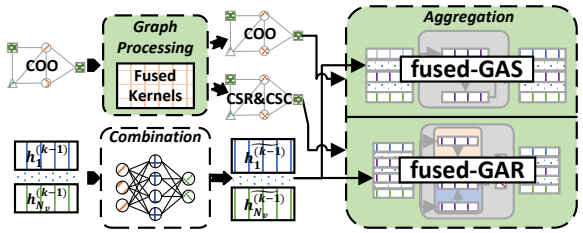

Figure 5: Design Overview of fuseGNN. The phases we optimized are marked with green.

The problem will be more severe on larger graphs. E.g. Reddit has 114,615,892 directed edges, which will consume over 58.7 GB when feature vector length is 128, not to mention the data movement.

Conclusions. First, in small graphs with short feature vectors, the kernel launching time has great impact on execution time. Secondly, in large graphs with long feature vectors, while the kernel execution time is long enough to overlap with the launching time (e.g. Pubmed 256), the large intermediate extended feature matrix results in high volume of data movement, DRAM storage footprint and low cache hit rate. To solve these problems, dedicated CUDA kernels are required in which kernel fusion and other optimization strategies are exploited to reduce the total number of kernels and explore data reuse opportunities.

4 DESIGN OVERVIEW

Here we provide a brief overview of our design. Different from previous studies, we use a three-phase abstraction: *Combination-Graph Processing-Aggregation* as shown in Figure 5.

Input Graph Format. Compared with CSC and CSR, unsorted COO format has the lowest cost to construct or modify. So our framework takes unsorted COO format graphs as input.

Design of Combination. As this stage is identical to a fully-connected layer that has been fully optimized in mainstream deep learning frameworks, we just use the primitive class *torch.nn.Linear* in PyTorch like previous studies.

Design of Graph Processing. Unlike previous studies, we take *Graph Processing* as a stand-alone phase as it has distinct execution pattern compared with the other two phases. *Graph Processing* updates the edge weight of the graph, and convert the COO format to CSR and CSC format for *GAR* model. As the edge weights are updated based on different rules (e.g., Equation (2) for GCN and Equation (4) for GAT), special CUDA kernels for each algorithm are developed. We exploit kernel fusion technique to reduce the total number of invoked kernels and increase on-chip data reuse. For format conversion, we encapsulate several efficient operations in cuSPRASE to convenient Python APIs.

Design of Aggregation. The advantage of GAR abstraction comes from its on-chip reduction that reduces data movement and eliminates atomic transactions. However, as the input graph is unsorted COO format, GAR model introduces extra overhead when converting it to CSR and CSC format. As a result, unlike previous studies, our *fuseGNN* provides both GAR and GAS abstractions. The former one is applied on graphs with high average degree (e.g. Reddit), and the latter one is used when the average degree is low (e.g. Cora). The forward and backward pass of both abstractions are optimized with strategies like kernel-fusion to reduce DRAM storage footprint as well as redundant data movement.

5 KERNEL DESIGN

In this section, we present the detailed design of the CUDA kernels used in *fuseGNN* along with the key optimization strategies.

5.1 Fused Graph Processing

Kernel fusion is one of the most popular ways of reducing kernel launching time and unnecessary data movement, in which a sequence of CUDA kernels are fused into a single one, and intermediate results can be stored in on-chip registers rather than written back to DRAM [19]. As different GNN models have distinct algorithms for *Graph Processing* phase, different fused kernels are required for different algorithms. Here we use GAT [17] as an example, the *Graph Processing* of which is shown in Equation (4). We assume that $\boldsymbol{a}^{(k)} \in \mathbb{R}^{2m\times 1}$ and feature vectors $\widetilde{\boldsymbol{h}_i^{(k-1)}} \in \mathbb{R}^{1\times m}$ are concatenated at the first dimension to form $\widetilde{\boldsymbol{H}^{(k-1)}} \in \mathbb{R}^{N_v\times m}$.

We first reshape $\boldsymbol{a}^{(k)}$ to an $m \times 2$ matrix and compute $\widetilde{\boldsymbol{a}^{(k)}} = \widetilde{\boldsymbol{H}^{(k-1)}} \times \boldsymbol{a}^{(k)}$ with dense matrix-matrix multiplication. Then, we launch a single kernel, in which each thread handles a single edge. It first computes the attention coefficient with

$$[\widetilde{\boldsymbol{h}_i^{(k-1)}} || \widetilde{\boldsymbol{h}_j^{(k-1)}}]\boldsymbol{a}^{(k)} = \widetilde{\boldsymbol{a}^{(k)}}[i][0] + \widetilde{\boldsymbol{a}^{(k)}}[j][1], \tag{5}$$

and store the result in a register. In this way, the original N_e inner products can be reduced to N_v. Then, *leaky ReLU* and *exp* are applied to the attention coefficient, and the result is not only written to DRAM as the numerator of each edge but also accumulated with *atomicAdd* to calculate the denominator of Equation (4). At last, a second kernel is launched, where each thread still handles an edge by dividing the numerator and denominator and applying dropout.

Compared with the naive implementation with PyTorch that has more than a dozen kernels and N_e inner products, our new implementation only takes a much smaller *sgemm* kernel and two dedicated fused kernels.

5.2 Parallel Reductions

Two kinds of parallel reductions are used in our *Aggregation* kernels to perform reductions of features and gradients in shared memory.

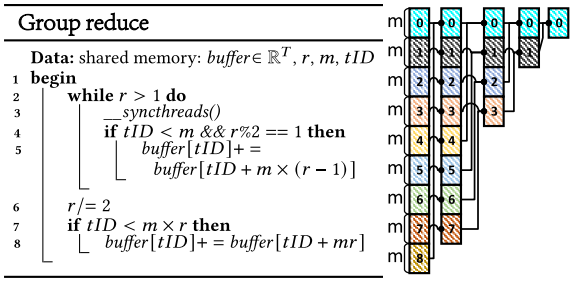

Figure 6: Group Reduce

Group Reduce: For an $m\times r$ vector $\boldsymbol{v}$, each consecutive m entries form a group, so there are totally r groups. Our target is to reduce the corresponding entries of each group into the first one: $\forall i < m$, $\boldsymbol{v}[i]+ = \sum_{j=1}^{r-1} v[i+mj]$. Figure 6 shows how group reduce works. When r is even, we perform parallel reduction to halve the number of groups. Otherwise, we reduce the last group to the first one, until there is only one left.

Block-wide Reduce: Given vector $\boldsymbol{v} \in \mathbb{R}^r$, block-wide reduce calculates $\sum_{i=1}^{r} \boldsymbol{v}[i]$. We follow the implementation in Harris, Mark (2017) [8] in which multiple optimization strategies including loop unrolling, divergent avoiding are applied.

Algorithm 1: fused-GAS Forward Kernel

```
Data: Input & output features: H_in, H_out ∈ ℝ^{N_v×m};
      COO Row & Col. Index: tarInd, srcInd ∈ ℕ^{N_e};
      Edge weight.: w_e ∈ ℝ^{N_e}; feature dim.: m ∈ ℕ;
      Block size T ∈ ℕ.
1  begin
2      tID = thread ID, bID = thread block ID.
3      if m < T then
4          stride = ⌊T/m⌋, fId = tID%m
5          B = ⌊(N_e + stride − 1)/stride⌋
6          if tID < m × stride then
7              for eId = bID × stride + ⌊tID/m⌋ to N_e
8              step B × stride do
9                  atomicAdd{&H_out[tarInd[eId]][fId],
10                 H_in[srcInd[eId]][fId] × w_e[eId]}
11     else
12         eId = bId, w = w_e[eId].
13         for fId = tID to m step T do
14             atomicAdd{&H_out[tarInd[eId]][fId],
15             H_in[srcInd[eId]][fId] × w}
```

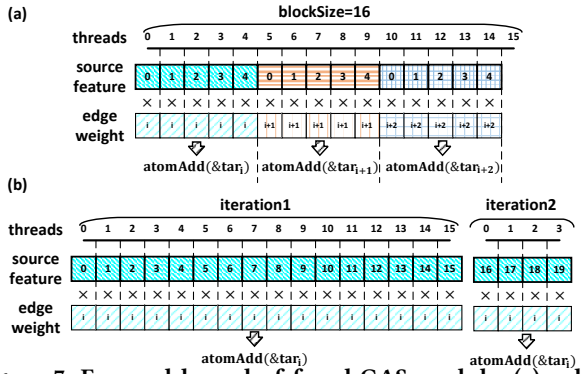

Figure 7: Forward kernel of fused-GAS module. (a): when feature length $m < T$; (b) when feature length $m \geq T$.

5.3 fused-GAS Forward and Backward Kernels

Fused-GAS partitions the workload to thread blocks in edge-centric way. For thread block size T and feature length m, each thread block handles the GAS of $max(\lfloor T/m \rfloor, 1)$ edges.

Forward Pass. Algorithm 1 shows the forward pass of *fused-GAS* model. It takes an $N_v \times m$ input feature matrix $\boldsymbol{H_{in}}$ and a COO format graph. We set $T = 256$ to maintain high occupancy.

If feature dimension m is smaller than block size T, as shown in line 3-10 in Algorithm 1, each thread block will handle $stride = \lfloor T/m \rfloor$ edges simultaneously. The consecutive entries in feature vector of each edge are handled by consecutive threads. Figure 7 (a) shows a toy example in which $T = 16$, $m = 5$. The thread block works on 3 edges: i, $i+1$, and $i+2$. Each of the first 15 threads loads the corresponding entry of source feature and multiplies it with the edge weight, then accumulates the result of multiplication to the address that stores the target feature vector with *atomicAdd*.

Otherwise, as shown in line 11-15 in Algorithm 1, each thread block only handles a single edge. At beginning, we load the scalar edge weight and store it in a register for reuse. Each iteration of the for loop at line 13 processes T consecutive entries of the feature vectors: it loads the source feature entry in, multiplies it with the edge weight in the register, and writes it to output target feature vector with *atomicAdd*. This process is illustrated in Figure 7 (b).

Algorithm 2: fused-GAS Backward Kernel

Data: Output & input gradient: $G_{in}, G_{out} \in \mathbb{R}^{N_v \times m}$;
Input feature: $H_{in} \in \mathbb{R}^{N_v \times m}$;
COO Row & Col. Index: $tarInd, srcInd \in \mathbb{N}^{N_e}$;
Edge weight.: $w_e \in \mathbb{R}^{N_e}$; Edge weight gradient: $g_e \in \mathbb{R}^{N_e}$
feature dim.: $m \in \mathbb{N}$; Block size $T \in \mathbb{N}$.

1 **begin**
2 ***shared*** *buffer*$[T-1:0]$
3 tID = thread ID, bID = thread block ID, *buffer*$[tID] = 0$
4 **if** $m < T$ **then**
5 $stride = \lfloor T/m \rfloor$
6 $B = \lfloor (N_e + stride - 1)/stride \rfloor$, $step = stride \times B$
7 $N_s = \lfloor (N_e - bID \times stride + step - 1)/step \rfloor$
8 $fId = tID\%m$, $gId = \lfloor tID/m \rfloor$, $eId = bID \times stride + gId$
9 **for** $i = bID \times stride$ ***to*** $N_s \times step + bID \times stride$
10 ***step*** $step$ **do**
11 *__syncthreads()*
12 **if** $tID < stride \times m$ && $eId < N_e$ **then**
13 $g = G_{out}[tarInd[eId]][fId]$
14 *atomicAdd*$\{\&G_{in}[srcInd[eId]][fId], g \times w\}$
15 *buffer*$[gId + fId \times stride] =$
$g \times H_{in}[srcInd[eId]][fId]$
16 *__syncthreads()*
17 *group reduce*(*buffer*, m, $stride$)
18 *__syncthreads()*
19 **if** $tID < stride$ && $i + tID < N_e$ **then**
20 $g_e[i + tID]$ =*buffer*$[tID]$
21 *buffer*$[tID] = 0$, eId+= $stride$
22 **else**
23 $eId = bId$, $w = w_e[eId]$
24 **for** $fId = tID$ ***to*** m ***step*** T **do**
25 $g = G_{out}[tarInd[eId]][fId]$
26 *atomicAdd*$\{\&G_{in}[srcInd[eId]][fId], g \times w\}$
27 *buffer*$[tID]$+= $g \times H_{in}[srcInd[eId]][fId]$
28 *__syncthreads()*
29 *block-wide reduce*(*buffer*)
30 **if** $tID == 0$ **then**
31 $g_e[eId]$ =*buffer*$[0]$

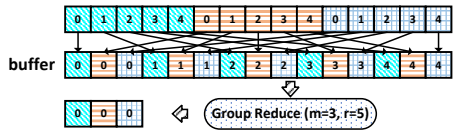

Figure 8: Illustration of line 15 & 17 in Algorithm 2

Backward Pass. When the gradient for edge weight is not required, we can directly use the forward kernel for backward pass by replacing input H_{in}, H_{out} with G_{out}, G_{in} and switching the $srcInd$ and $tarInd$. Otherwise, we use the kernel in Algorithm 2. Line 9-14 and 24-26 calculate the gradient matrix of input features. They are basically the same as line 6-9 and 13-14 in Algorithm 1 with changed inputs and different looping way. Line 15-21 and line 27-31 calculates the gradient of each scalar edge weight with three steps:1) save the gradient contributed by each entry in a buffer in shared memory (line 15 & 27); 2) do reduction to generate the gradient of edge weight (line 17 & 29);3) write the result to DRAM (line 20 & 31).

As $\lfloor T/m \rfloor$ edges are handled simultaneously when $m < T$, we store the gradients in an interleaved fashion (line 15 in Algorithm 2) as shown in Figure 8. Then we calculate the gradient of each edge weight with *group reduce* under group size $\lfloor T/m \rfloor$ and number of group m. The major benefit brought by the interleaved fashion is that in step 2) and 3), all the active threads are consecutive, therefore we can avoid warp divergence (different threads of the same warp take different branch) [7] to the most extent. When we have $m \geq T$, as the thread block only handles a single edge, we store the gradient contributed by each entry consecutively (line 27 in Algorithm 2) and do block-wide reduction (line 29 in Algorithm 2).

Algorithm 3: fused-GAR Forward Kernel

Data: Input & output features: $H_{in}, H_{out} \in \mathbb{R}^{N_v \times m}$;
CSR Row Ptr & Col. Index: $tarPtr \in \mathbb{N}^{N_v+1}, srcInd \in \mathbb{N}^{N_e}$;
Edge weight.: $w_e \in \mathbb{R}^{N_e}$, Self-loop weight: $w_{sl} \in \mathbb{R}^{N_v}$;
feature dim.: $m \in \mathbb{N}$; Block size $T \in \mathbb{N}$; Grid size: $B \in \mathbb{N}$.

1 **begin**
2 tID = thread ID, bID = thread block ID.
3 $start = tarPtr[bID]$, $stop = tarPtr[bID + 1]$
4 **if** $m < T$ **then**
5 ***shared*** *buffer*$[T-1:0]$
6 $stride = \lfloor T/m \rfloor$, $fId = tID\%m$, *buffer*$[tID] = 0$
7 **for** $eId = start + \lfloor tID/m \rfloor$ ***to*** $stop$ ***step*** $stride$ **do**
8 *buffer*$[tID]$+= $H_{in}[srcInd[eId]][fId] \times w_e[eId]$
9 *group reduce*(*buffer*, $min(m, stop - start)$, m)
10 **if** $tID < m$ **then**
11 $H_{out}[bID][fId] = H_{in}[bID][fId] \times w_{sl}$+*buffer*$[tID]$
12 **else**
13 $w = w_{sl}[bId]$
14 **for** $fId = tID$ ***to*** m ***step*** T **do**
15 *buffer*= $H_{in}[bID][fId] \times w$
16 **for** $eId = start$ ***to*** $stop$ *step* 1 **do**
17 *buffer*+ = $H_{in}[srcInd[eId]][fId] \times w_e[eId]$
18 $H_{out}[bID][fId]$ =*buffer*

5.4 fused-GAR Forward and Backward Kernels

Fused-GAR kernel partitions the workload to thread blocks in the vertex-centric way. Each thread block will handle all the edges with the same target vertex in forward pass and all the edges with the same source vertex in backward pass.

Forward Pass. First of all, the thread block identifies the index to the first and last edge it handles based on $tarPtr$ (line 3). Similar to the *fused-GAS* forward kernel, we can process $\lfloor T/m \rfloor$ edges simultaneously when $m < T$ (line 4-11).

As all the edges share the same target vertex, we can do on-chip reduction: consecutive m threads form a thread group, and the edges are partitioned evenly to the thread groups. Each thread group processes the edges assigned to it in sequence (loop at line 7). After all the partial results are stored in the buffer in shared memory, we apply *group reduce* (line 9) to the buffer under group size m and number of group $\lfloor T/m \rfloor$ to get the final output feature vector. When we have $m \geq T$, as all the threads work on the same edge in each iteration, we just use a single register for each thread to do reduction (line 12-18).

Backward Pass. Similar to *fused-GAS*, the forward kernel can be used for backward when gradient on edge weights are not required. Otherwise, the same strategy in *fused-GAS* backward kernel is applied. Specifically, to generate gradient on edge weight, when $m < T$, the gradient contributed by each entry is stored in shared memory in an interleaved fashion and reduced with *group reduce*. Otherwise, the gradient are stored consecutively and reduced with block-wide reduction. The gradient on input feature vectors is calculated in the symmetric way of forward pass.

Besides, we fully exploit the data reuse opportunities. In backward pass, as all the edges share the same source vertex, the feature vector of which will be used to generate the gradient of all the edge weight. So we cache it at the beginning in registers when $m < T$ or shared memory otherwise. Besides, the gradient of the target vertex is required for both input feature gradient and edge weight gradient, so it is cached in a register for reuse.

5.5 Discussion on Kernel Design

Optimization Strategies. First of all, as consecutive threads work on consecutive entries in feature vectors, and each thread block handles consecutive edges, the kernels can achieve good DRAM transaction coalescing and high atomic transaction bandwidth as they are in the same cache line [15]. Second, the DRAM transactions are further reduced with extensive data reuse. For instance, as the three stages of Aggregation are fused in the single kernel, the data that will be used multiple times are cached with shared memory or registers. Moreover, giving the credit to interleaved fashion in Figure 8, reduction strategies, as well as the looping strategy in backward kernels (e.g. line 9 in Algorithm 2), the active threads are always kept consecutive to avoid warp divergence to the most extent. Last but not least, multiple edges can be handled concurrently so that our kernels can maintain high occupancy even with short feature vectors.

Flexibility vs. Performance. Although fused kernels have much lower latency and memory footprint, their re-usability are limited. As a result, it is impractical to produce libraries consisting of already-fused kernels [4]. Previous studies solve this problem by following "*Make the Common Case Fast*" idea. The "common case" in them refers to *Aggregation* phase in which *ApplyEdge* is element-wise operation and gradient on edge weight is not required. For example, models like GCN [10], GIN [22], SGC [21], and GraphSAGE [6] (except for LSTM aggregator) directly use a scalar edge weight. For these common cases, *neuGraph* provides the *Fused-Gather* kernel while *DGL* exploits *SpMM* in *cuSPARSE* library [13]. Other uncommon cases are still implemented with simple and re-usable kernels like *PyG*.

While "*Make the Common Case Fast*" strategy is also exploited in our *fuseGNN*, the "common case" in our work is relaxed to *Aggregation* phase in which *ApplyEdge* is element-wise operation, because gradient on edge weight is supported. Therefore, recent models with complex attention mechanism like GAT [18] and AGNN [17] treated as uncommon case in previous studies are included into "common case" in our *fuseGNN*.

Besides, unlike *DGL* that uses APIs in closed source library *cuSPARSE*, fused kernels in our *Aggregation* phase are developed in neighborhood aggregation fashion [3] so that they can be used as templates when developing new *Aggregation* kernels.

6 EVALUATION

In this section, we evaluate the performance of our *fuseGNN* and compare it with state-of-the-art studies on a single NVIDIA V100 GPU [9]. The benchmarks are denoted as "Model-Dataset-Hidden". For "Model", we pick GCN [10] and GAT [18] to cover GNNs with simple and complex *Graph Processing*. "Dataset" includes Cora, Pubmed, and Reddit to cover various scale and average degree. "Hidden" (output dimension of *Combination* phase) is chosen from $\{16, 64, 128, 256, 512\}$. Both transductive learning and inductive learning are evaluated, where the *Graph Processing* phase is executed at each iteration or the result is cached in DRAM in the first execution and reused in later iterations, respectively.

6.1 Latency

As our first motivation is to reduce latency of GNN training, here we first evaluate the latency of our *fuseGNN* on several benchmarks.

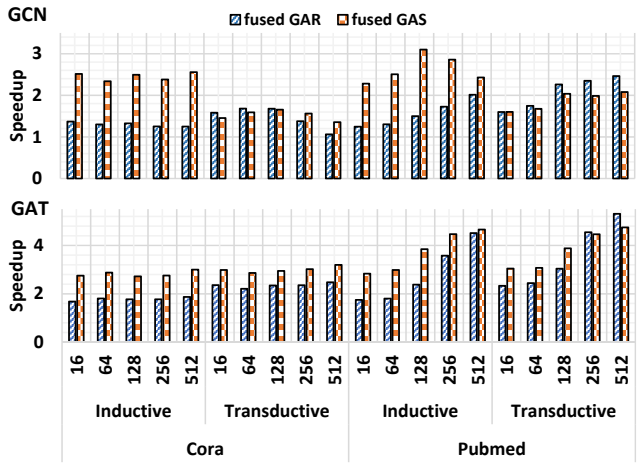

Figure 9: Speedup of fused GAR and fused GAS over PyG under different configurations.

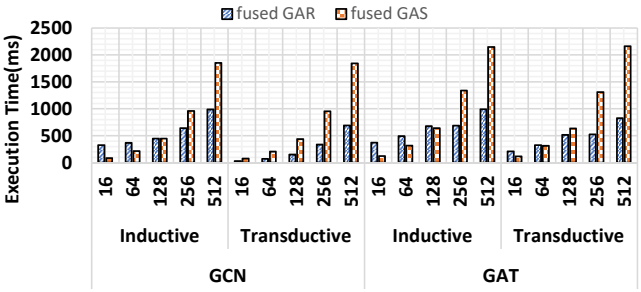

Figure 10: Latency of fused GAR and fused GAS on Reddit.

GAS v.s. GAR. Figure 9 summarizes the relative end-to-end speedup over *PyG* [3] achieved under different configurations. It shows that the speedup provided by our study is consistent and significant. On small graphs like Cora, *fused GAS* could achieve higher speedup compared with *fused GAR* abstraction. Figure 10 compares the latency of *fused GAR* and *fused GAS* on *Reddit*. It shows that *fused GAR* module is more effective than *fused GAS* on graphs with high average degree like *Reddit*.

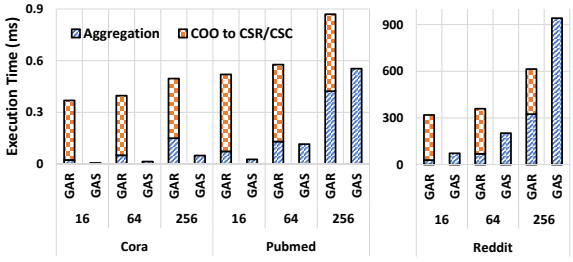

Figure 11: Latency of Aggregation and format conversion

To further justify this, Figure 11 shows the execution time of aggregation and graph format conversion on different benchmarks. First, on graphs with short feature vector and low average degree, *fused GAS* has lower *Aggregation* latency, this is because its kernel is simpler than *fused GAR* so that fewer registers are used and higher occupancy can be achieved. On graphs with long feature vectors and high average degree, *fused GAR* achieves lower *Aggregation* latency which outweighs the additional overhead of format conversion.

With all these observations, empirically, one should select *fused-GAR* for graphs with high average degree and *fused-GAS* for others. While a dedicated performance models can be built in future studies, as our *fused-GAS* and *fused-GAR* share the same interface, the user can just try both of them and pick the better one.

Table 3: Comparison of Training Latency (in millisecond) between DGL and our implementation.

Dataset	Model	hidden=16		hidden=64		hidden=512	
		DGL	Ours	DGL	Ours	DGL	Ours
Cora	GCN	2.35	**1.05**	2.37	**1.09**	2.51	**1.10**
	GAT	7.46	**1.65**	7.59	**1.68**	8.07	**1.70**
Pubmed	GCN	2.69	**1.07**	2.79	**1.11**	3.61	**3.00**
	GAT	7.75	**1.72**	7.85	**1.72**	10.43	**3.74**
Reddit	GCN	**22.5**	29.2	**58.9**	71.9	**452.9**	690.4
	GAT	*OOM*	**120.0**	*OOM*	**314.3**	*OOM*	**825.3**

Comparison to DGL [20]. We choose single layer GCN and GAT on Cora, Pubmed, and Reddit under dimension 16, 64, and 512 as benchmarks. The results are summarized in Table 3. For "Ours", the lower one in the latency of GAR and GAS is taken.

On small datasets like *Cora* and *Pubmed*, our implementation consistently achieves much lower latency. The major speedup comes from kernel-fusion applied to *Graph Processing* phase. For example, in *Cora-GAT*, 52 kernels are invoked in *DGL*, while our *fused-GAS* only launches 24 kernels.

On *GCN-Reddit* where *Aggregation* phase becomes the major bottleneck, *DGL* has lower latency due to two major reasons. First, unlike our model, *DGL* does the COO to CSR/CSC conversion offline, so that this overhead is not included in their latency. Second, *DGL* directly exploits the *CSR SpMM* kernels in *cuSPARSE* library [13] that is optimized by more experienced experts of NVIDIA in SASS. However, compared with the closed source *cuSPARSE* library, our kernel can be easily modified to support new GNN algorithms.

On *GAT-Reddit*, as gradient on edge weight is not support by the *SpMM* implementation, *DGL* suffers from *OOM* with hidden=16. Oppositely, our framework can even support hidden=512.

Comparison to NeuGraph [12]. As the authors haven't yet released their code, it is hard to have a thorough comparison across multiple benchmarks. However, first, the backward kernel for *fused-Gather* is not provided, so the high volume memory storage footprint and data movement remain unsolved for models like GAT. Second, their *fused-Gather* kernel is also less effective compared with our *fused-GAR* kernel under certain scenarios. We implement the *fused-Gather* kernel based on their description and compare it on benchmark *GCN-Reddit-16*. As shown in Table 4, when dimension is 16, our *fused-GAR* kernel processes 8 edges simultaneously to fully exploit the thread block size 128. On the other hand, the *fused-Gather* kernel doesn't involve such design, so only 16 threads in each thread block are activated. As a result, it has much fewer active warps per SM which leads to lower thread-level parallelism and low DRAM Bandwidth [1]. Besides, our *fused-GAR* reduces the number of steps to process n edges from $O(n)$ to $O(\lceil n/r \rceil + \log r)$, $r = \lfloor T/m \rfloor$ where T is thread block size and m is the dimension.

Table 4: Comparison on *GCN-Reddit-16*

Kernel	Latency	Active Warps	Occupancy	DRAM Bandwidth
fused-GAR	13.6 ms	63.13 / SM	98.6%	327.1 GB/s
fused-Gather[12]	23.5 ms	30.28 / SM	47.3%	196.1 GB/s

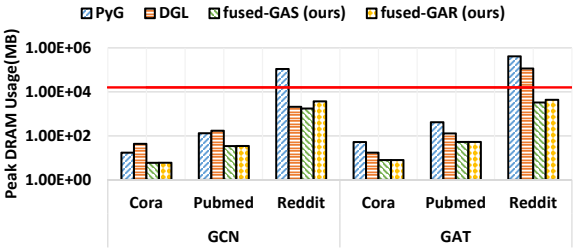

Figure 12: Peak DRAM usage of the first layer of GCN/GAT with output dimension 128. The red line marks 16 GB.

Table 5: Data Movement (*: atomic transactions)

Kernel	L2 $ Read	L2 $ Write	DRAM Read	DRAM Write
GCN-Pubmed-256				
fused-GAS	92.5 MiB	90.78* MiB	107.8 MiB	57.0 MiB
fused-GAR	107.6 MiB	19.3 MiB	61.6 MiB	20.9 MiB
GCN-Reddit-128				
PyG(theoretical)	178 GiB	118 + 58* GiB	222 GiB	161 GiB
fused-GAS	59.4 GiB	58.7* GiB	100.9 GiB	50.2 GiB
fused-GAR	55.5 GiB	113.8 MiB	47.1 GiB	116.4 MiB

6.2 Memory

Here we evaluate the DRAM storage footprint and data movement reduction with our *fuseGNN* over existing studies.

Peak DRAM Usage. Figure 12 compares the peak DRAM Usage of different frameworks with hidden dimension 128. Our *fuseGNN* reduces the storage footprint by several orders. In particular, the peak DRAM usage is reduced by 95× and 26× on *GAT-Reddit-128* compared with *PyG* and *DGL*, respectively. This makes it possible to fit all these models in a single V100 GPU.

Data Movement. Table 5 summarizes the data movement in our *fused-GAS/GAS* forward kernels under the same benchmark in Table 2. On *GCN-Pubmed-256*, the *fused-GAS* kernel reduces non-atomic L2 transactions (read and write) by 4.75× and DRAM transactions by 3.14×. On the other hand, the non-atomic L2 and DRAM transaction of *fused-GAR* are reduced by 3.46× and 6.28×, and the atomic transactions are eliminated. First, transaction related to the intermediate extended feature vectors are eliminated. Second, while the L2 cache hit rate of element-wise kernel is around 0%, our kernels has around 30%, as a larger portion of the input feature matrix can be cached compared with the huge extended feature matrix. On *GCN-Reddit-128*, *fused-GAR* can reduce the L2 cache write transaction by more than 1,500×.

7 CONCLUSIONS

In this paper, we provide a highly optimized extension library for PyTorch for GNN training on GPU. Attributed to our dual abstraction design (GAR and GAS) and dedicated CUDA kernels, our work not only significantly improves the training throughput but also reduces DRAM storage footprint, which makes it possible to train GNN on larger graphs without adding more hardware. The designs can be easily extended to multi-GPU or CPU+GPU scenarios, in which the graph is partitioned and assigned to each GPU.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation (Grant No. 1817037, 1725447, and 1730309).

REFERENCES

[1] C CUDA. [n.d.]. Best Practices Guide-CUDA Toolkit Documentation.

[2] Yangdong Deng, Bo David Wang, and Shuai Mu. 2009. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*. 539–546.

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

[4] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Luděk Matyska. 2015. Optimizing CUDA code by kernel fusion: application on BLAS. *The Journal of Supercomputing* 71, 10 (2015), 3934–3957.

[5] David B Glasco, Peter B Holmqvist, George R Lynch, Patrick R Marchand, Karan Mehra, and James Roberts. 2012. Cache-based control of atomic operations in conjunction with an external ALU block. US Patent 8,135,926.

[6] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Advances in neural information processing systems*. 1024–1034.

[7] Tianyi David Han and Tarek S Abdelrahman. 2013. Reducing divergence in GPGPU programs with loop merging. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*. 12–23.

[8] Mark Harris et al. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 70.

[9] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele P Scarpazza. 2018. Dissecting the nvidia volta gpu architecture via microbenchmarking. *arXiv preprint arXiv:1804.06826* (2018).

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

[11] Guohao Li, Matthias Müller, Guocheng Qian, Itzel C Delgadillo, Abdulellah Abualshour, Ali Thabet, and Bernard Ghanem. 2019. Deepgcns: Making gcns go as deep as cnns. *arXiv preprint arXiv:1910.06849* (2019).

[12] 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 *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 443–458.

[13] M Naumov, LS Chien, P Vandermersch, and U Kapasi. [n.d.]. Cusparse library.

[14] CUDA Nvidia. 2008. Cublas library. *NVIDIA Corporation, Santa Clara, California* 15, 27 (2008), 31.

[15] Lars Nyland and Stephen Jones. 2013. Understanding and using atomic memory operations. In *4th GPU Technology Conf.(GTC'13), March*.

[16] Hao Qian and Yangdong Deng. 2011. Accelerating RTL simulation with GPUs. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 687–693.

[17] Kiran K Thekumparampil, Chong Wang, Sewoong Oh, and Li-Jia Li. 2018. Attention-based graph neural network for semi-supervised learning. *arXiv preprint arXiv:1803.03735* (2018).

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

[19] Guibin Wang, YiSong Lin, and Wei Yi. 2010. Kernel fusion: An effective method for better power efficiency on multithreaded GPU. In *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE, 344–350.

[20] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019). https://arxiv.org/abs/1909.01315

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

[22] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*. https://openreview.net/forum?id=ryGs6iA5Km

[23] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Characterizing and Understanding GCNs on GPU. *IEEE Computer Architecture Letters* (2020).

[24] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[25] Hongbo Zhang, Tan Yan, Martin DF Wong, and Sanjay J Patel. 2011. Accelerating aerial image simulation with GPU. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 178–184.