GRAPHILER: A COMPILER FOR GRAPH NEURAL NETWORKS

Zhiqiang Xie [1 2] **Zihao Ye** [2] **Minjie Wang** [2] **Zheng Zhang** [2] **Rui Fan** [1]

ABSTRACT

Graph neural networks (GNNs) are a powerful and versatile machine learning technique, but programming and computing with GNNs pose a number of challenges. Current GNNs frameworks are based on a message passing paradigm, and allow the concise expression of GNN models using built-in primitives and user defined functions (UDFs). However, while built-in primitives offer high performance, they are limited in their expressiveness. Meanwhile, UDFs are flexible, but often have low performance and run out of memory on large graphs. In this paper, we propose Graphiler, a compiler stack for GNNs which achieves high performance and provides a flexible programming interface. We first show how to represent message passing processes as data flow graphs (DFGs), then apply a number of optimizations to improve efficiency and reduce memory footprint, and finally implement a set of high performance extended primitives to execute the DFGs. Experiments show Graphiler can accelerate a GNN model programmed with UDFs by up to two orders of magnitude, and achieves performance close to or sometimes faster than expert designed implementations using built-in primitives.

1 INTRODUCTION

Graph neural networks (GNNs) have recently achieved state-of-the-art performance in a variety of application domains, including recommendation systems (Ying et al., 2018), drug discovery (Chen et al., 2018a), combinatorial optimization (Li et al., 2018) and others. GNNs combine operations from deep neural networks (DNNs) with iterative graph propagation and compute features on nodes using neural networks and features from neighboring nodes and edges. GNNs can be characterized by a message passing paradigm (Gilmer et al., 2017; Hamilton et al., 2017) consisting of three stages: *message creation*, *message aggregation* and *feature update*. This simple yet powerful formulation allows for concisely expressing a broad range of GNN models, and has been adopted by a number of popular GNN frameworks, including DGL (Wang et al., 2019b), PyG (Fey & Lenssen, 2019), PGL (PGL, 2019) and Graph Nets (Battaglia et al., 2018).

These frameworks, however, face several key challenges in terms of performance and flexibility. In particular, to allow users to easily build and experiment with novel GNN architectures, several existing frameworks allow the creation of user-defined functions (UDFs). UDFs can be programmed using standard tensor operations which users are familiar with, while details of the graph operations are hidden by the frameworks. But while this design provides a simple way to construct a GNN model, naive implementations of UDFs by users often result in greatly reduced performance as well as out of memory errors due to redundant computation and memory access and excessive data materialization (Huang et al., 2021). In addition, to support complex UDFs involving nonstandard graph operations, current practices (Wang et al., 2019b) transform the graph operations into many small tensor operations supported by DNN frameworks, resulting in significant overhead from excessive function calls which further degrades performance.

To achieve higher performance, current GNN frameworks provide a limited set of built-in primitives based on highly efficient sparse matrix operations, such as $message\_and\_aggregate$ in PyG and the GSPMM / GSDDMM functions in DGL. These primitives are also used by framework developers to construct built-in modules. However, it is nontrivial for ordinary users to build nonstandard GNN models using these primitives due to the limited expressiveness they offer. Furthermore, it is even more challenging for users to optimize their models for efficiency, as doing so requires expertise in both graph and neural network computations as well as in-depth understanding of the primitives and their implementations.

Given the fundamental limitations of both UDFs and built-in primitives in current GNN frameworks, we propose *Graphiler*, a GNN compiler which automatically compiles GNNs defined using UDFs into a set of high performance primitives which can be executed using efficient execution

[1]ShanghaiTech University [2]Amazon Web Services. Correspondence to: Zhiqiang Xie <xiezhq@shanghaitech.edu.cn>, Minjie Wang <minjiw@amazon.com>.

Proceedings of the First MLSys Workshop on Graph Neural Networks and Systems (GNNSys'21) , San Jose, CA, USA, 2021.

plans. Graphiler first transforms a GNN program containing multiple UDFs into a special computational graph we term a *message passing data flow graph* (MP-DFG). This combines message passing semantics with data flow graphs, and serves as the intermediate representation of Graphiler. Graphiler then optimizes the MP-DFG using various message passing specific optimizations including operator reordering, operator split and concatenation, graph operator lowering and kernel fusion to eliminate redundant computation and memory accesses and reduce memory consumption. Finally, a high performance execution plan consisting of extended primitives is generated.

To evaluate the effectiveness of Graphiler, we use it to compile four GNN models written by non-expert users. After negligible code modifications, we show that Graphiler can accelerate the models by up to two orders of magnitude, and can achieve performance which is close to or sometimes superior to those from careful implementations of the models by expert users using high performance built-in primitives. To the best of our knowledge, Graphiler is the first compiler stack built for GNNs which utilizes message passing semantics as part of its optimization passes to achieve high performance without restricting users to a limited set of built-in primitives.

2 Background & Motivation

In this section we provide an overview of the message passing paradigm for GNNs, then discuss in detail some problems faced by current implementations of the paradigm.

2.1 Message Passing Paradigm for GNNs

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with the set of nodes $\mathcal{V}$ and the set of edges $\mathcal{E}$. Let $x_u, x_v \in \mathbb{R}^{f_v}$ denote the feature vectors for nodes u and v, and $w_e \in \mathbb{R}^{f_e}$ the feature vector for an edge (u, e, v)[1]. The message passing paradigm for GNNs consists of three stages:

$$\begin{aligned} m_e &= \phi\left(x_u, x_v, w_e\right), (u, e, v) \in \mathcal{E}, \\ h_v &= \rho\left(\{m_e : (u, e, v) \in \mathcal{E}\}\right), \\ x_v^{new} &= \psi\left(x_v, h_v\right), v \in \mathcal{V}. \end{aligned}$$

Message creation Each edge produces a message by applying an edge-wise message function ϕ to its own features and the features of incident nodes.

Message aggregation Each node aggregates the messages from incoming edges using an aggregation function ρ.

Feature update Each node updates its features using a node-wise update function ψ.

[1] We follow the notation adopted in (Wang et al., 2019b), where e represents the ID of this edge.

2.2 Performance Problems of User-Defined Functions

The flexible choice of the message function ϕ, aggregation function ρ and update function ψ is key to the success of GNNs. To support complex, novel GNN models, existing GNN frameworks allow users to program using *user-defined functions (UDFs)*. While the node-wise update function ψ is typically straightforward to implement, since it is only applied to a node's own data, the ϕ and ρ functions require careful design, as they may cause redundant computations and require trading off between flexibility and performance.

2.2.1 Redundancy

Since all edges have a single source and single destination node, the inputs to the message creation function ϕ have the same size on every edge. Thus, existing frameworks use scatter or similar operations to broadcast data from nodes onto edges to enable message creation UDFs to be programmed using tensor operations.

However, this design comes with a cost: not only do scatter operations consume $O(|\mathcal{E}|)$ amount of memory and bandwidth, but subsequent operations on edge data perform redundant computations and memory accesses, increasing these costs from $O(|\mathcal{V}|)$ to $O(|\mathcal{E}|)$. As we show later, these redundant operations can significantly hurt overall performance.

2.2.2 Flexibility and Performance Trade-off

Unlike the update and message UDFs, aggregation UDFs generally cannot be directly transformed to dense tensor operations because nodes can have different numbers of incoming edges. Different GNN frameworks deal with this issue using a trade-off between flexibility and performance.

- PyG only provides a limited set of built-in aggregation functions implemented in specialized CPU/GPU kernels. While this approach offers excellent performance, it is also restrictive and requires significant user effort because users need to decompose their aggregation UDFs into a combination of built-in aggregators and primitives.
- PGL and Graph Nets provide *ragged tensors* as a versatile data structure for aggregation UDFs. A ragged tensor consists of a data array to store incoming messages in a contiguous memory block and an auxiliary index array to mark the neighbor segments of each node. However, operator support for ragged tensors is currently limited, as implementing high performance kernels for ragged tensors is sometimes challenging. In addition, users need to manage both ragged and dense tensor which complicates code maintenance.
- DGL provides users both with a set of built-in aggregation functions, as well as the ability to write aggregation UDFs using dense operators provided by the

underlying deep learning framework. In order to leverage the rich operator set for dense tensors, DGL groups nodes with the same in-degree into a single bucket for execution. However, this can result in a large number of small buckets, especially in real world graphs with skew degree distributions, and can cause significant performance degradation due to the overhead from launching a large number of CPU / GPU kernels.

Beyond individual UDFs, there are further optimization opportunities when considering multiple UDFs together. For example, DGL has shown that fusing message and aggregation UDFs to avoid materialization of edge message data can substantially reduce memory footprint and bandwidth consumption. (Huang et al., 2021) points out that segmenting a GNN computation into multiple built-in primitives causes redundancy in memory accesses and overhead from excessive function calls. Such optimizations require expertise in both GNN workloads as well as in-depth understanding of low-level implementations. The increasing complexity of optimizing GNN performance calls for a systematic approach to enable and manage optimizations.

3 A Running Example

In the remainder of the paper, we take the widely used *graph attention network (GAT)* (Velickovic et al., 2018) as a running example to illustrate the design of Graphiler and show step-by-step how it constructs and optimizes an MP-DFG. GAT introduces an attention mechanism as a substitute for statically normalized convolution operations, and can be formulated as follows:

$$z_i^{(l)} = W^{(l)} h_i^{(l)}, \tag{1}$$

$$e_{ij}^{(l)} = LeakyReLU(\overrightarrow{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)})), \tag{2}$$

$$\alpha_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k\in\mathcal{N}(i)} \exp(e_{ik}^{(l)})}, \tag{3}$$

$$r_i^{(l)} = \sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)}, \tag{4}$$

$$h_i^{(l+1)} = \sigma(r_i^{(l)}). \tag{5}$$

Equations (1) and (2) belong to message creation stage and prepare messages z and e on each edge. Equations (3) and (4) belong to aggregation stage, where each node receives messages from incoming edges and produces an aggregated result r. Equation (5) occurs in the aggregation stage, where each node's feature h is updated by an activation function σ. These formulas can be expressed programmatically using the UDFs in Listing 1.

4 Design of Graphiler

The observations in §2 motivate Graphiler, a compiler stack for GNNs which achieves high performance and provides a flexible programming interface. Graphiler adopts message passing data flow graphs (MP-DFGs) as an intermediate representation, which extends DFGs for ordinary DNN models using message passing semantics. In §4.1, we show to construct an MP-DFG for GAT's message creation, aggregation and update UDFs. In §4.2, we describe how to deal with the redundant computations and high memory usage problems discussed in §2.2 using a set of optimization passes on the MP-DFG. Finally, in §4.3, we describe how to extend the primitives set provided by existing frameworks to cover a larger family of graph operations, such as segmented operators and coarse-grain fused primitives. The output of this process is a set of extended primitives, from which an efficient execution plan will be generated. Figure 1 summarizes the overall workflow of Graphiler.

```
def message_func(edges):
    # equation (1)
    z_src = W * edges.src['h']
    z_dst = W * edges.dst['h']
    # equation (2)
    z_concat = concat(z_src, z_dst, dim=1)
    a = W_attention * z_concat
    e = leaky_relu(a)
    return {'z': z_src, 'e': e}

def aggregation_func(message):
    # equation (3)
    alpha = softmax(message['e'], dim=1)
    # equation (4)
    x = alpha * message['z']
    h = sum(x, dim=1)
    return {'h': h}

def update_func(nodes):
    # equation (5)
    return {'h': relu(nodes.data['h'])}
```

Listing 1: Pseudo code of GAT.

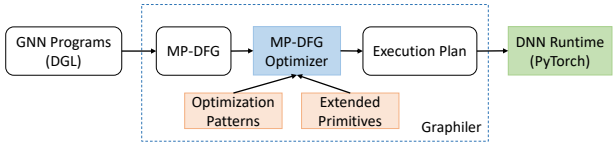
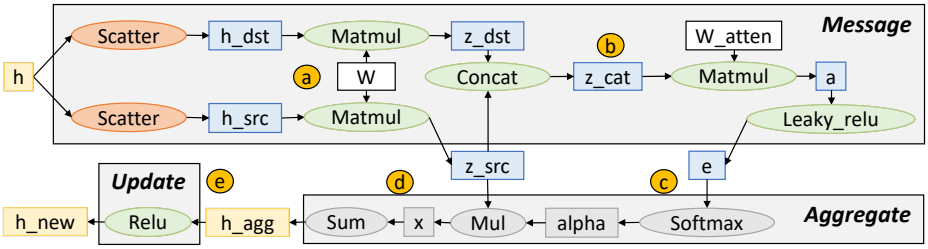


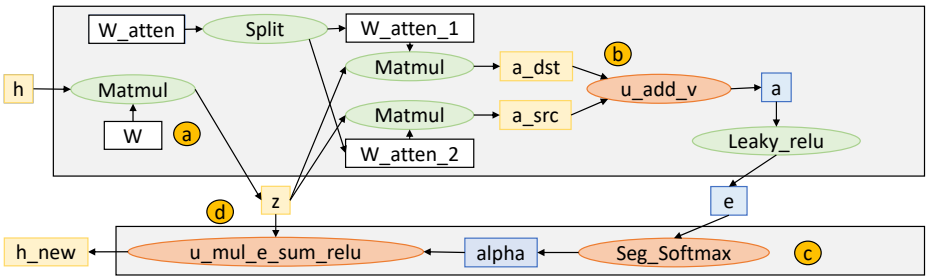
Figure 1. Overall workflow of Graphiler.

4.1 Message Passing Data Flow Graph Builder

Graphiler first converts a GNN model, such as the one shown in Listing 1 for the GAT model, into an MP-DFG, such as the one shown in Figure 2a. A key operation in nearly all GNNs is data movement between nodes and edges, and existing GNN frameworks generally provide interfaces to implicitly or explicitly specify these movements. By leveraging the semantics of these operations, Graphiler can insert corresponding graph operators and infer properties about data and operators in the MP-DFG.



(a) Original MP-DFG.



(b) Optimized MP-DFG.

Figure 2. Transformation of MP-DFG for GAT.

Specifically, $edges.src['h']$ and $edges.dst['h']$ in Listing 1 are translated into scatter operators in Figure 2 (indicated by orange ovals) which scatter data from nodes to outgoing and incoming edges respectively. Standard operators with dense tensors as inputs or outputs are represented by green ovals in the figure. Other than weight matrices, most tensors in an MP-DFG can be classified as either data on nodes or on edges, and these are represented by yellow and blue rectangles in the figure, respectively. Tensors and operators in the aggregation UDF are colored in gray, indicating that the data representation or operator semantics may change depending on the underlying implementation, *e.g.* using bucketing with dense operators in DGL, as discussed in §2.2.2.

4.2 MP-DFG Level Optimization Passes

While DFG-level optimizations have been widely used for DNN performance optimization (Jia et al., 2019; Ma et al., 2020), the use of MP-DFGs for GNN compilation and scheduling has been largely unexplored. By building MP-DFGs with rich message passing semantics in Graphiler, we open up a broad optimization space for message passing related patterns. In the following we list some of the optimizations enabled by our MP-DFG representation.

4.2.1 *Operator Reordering*

As indicated by the area marked a in Figure 2, the original MP-DFG scatters features h from nodes onto edges and then multiplies these by the weight matrix W with size $(f_v \times k)$. This "scatter-compute" approach causes redundant computations, and has $O(|\mathcal{E}|f_v k)$ computational complexity using the matrix multiplication operator $Matmul$. By reordering the computation to "compute-scatter", the computational complexity decreases substantially to $O(|\mathcal{V}|f_v k)$. In addition to $Matmul$, this optimization can be applied to other linear operations as well.

4.2.2 *Concatenation and Split*

The "concatenate-multiply" operations in equation 2 can be transformed to "split-multiply-sum" as follows:

$$\overrightarrow{a}^{(l)^T}(z_i^{(l)}||z_j^{(l)}) \implies \overrightarrow{a_1}^{(l)^T} z_i^{(l)} + \overrightarrow{a_2}^{(l)^T} z_j^{(l)}.$$

Combined with the optimization in §4.2.1, the "scatter-concatenate-multiply" operations can be transformed into "split-multiply-scatter-sum". This allows moving the multiplication of $\overrightarrow{a}^{(l)^T}$ and $(z_i^{(l)}||z_j^{(l)})$ on edges to multiplications $\overrightarrow{a_1}^{(l)^T} z_i^{(l)}$ and $\overrightarrow{a_2}^{(l)^T} z_j^{(l)}$ on nodes. In the case of GAT, this reduces the computational complexity from $O(|\mathcal{E}|f_v)$ to $O(|\mathcal{V}|f_v + |\mathcal{E}|)$ In addition, this transformation enables possible further optimization as discussed in §4.2.4.

4.2.3 *Aggregation UDF Lowering*

As we discussed in §2.2.2, existing GNN frameworks cannot effectively support aggregation UDFs due to issues such as inflexible interfaces or poor performance from bucketing. To overcome these problems, Graphiler lowers (Rotem

et al., 2018) aggregation UDFs programmed using dense tensor operations to MP-DFGs, and then automatically infers and replaces the operations in aggregation UDFs by extended primitives. For example, in the case of GAT, as indicated by c in Figure 2, Graphiler replaces the $Softmax$ operator marked in grey by the segmented softmax operator $Seg\_Softmax$ in orange. The Mul operator in grey is inferred to be a tensor operator, and the Sum operator is replaced by a Seg_Sum operator. These operators will further be fused in §4.2.4. Graph operations in aggregation UDFs which are supported by our extended primitives are thus executed using high performance kernels, while operations which are as-yet unsupported use the fallback bucketing approach. Part of our current work on Graphiler involves broadening the range of supported primitives.

4.2.4 *Kernel Fusion*

In DNNs, the tensor produced by one operator is often immediately consumed by the following operator. In these cases, the two operators can be fused to greatly reduce memory bandwidth use and kernel launch overhead. In the case of GNNs, the benefits from operator fusion can be even greater. This is because of the prevalence of "scatter-aggregate" computation patterns in GNNs, where the input and output tensors are node data with total size $O(|\mathcal{V}|)$, while the intermediate tensors which would be produced without operator fusion have size $O(|\mathcal{E}|)$. This suggests that operator fusion in GNNs can significantly reduce memory footprint and execution time, especially in high degree graphs.

Labels b, d and e in Figure 2 indicate three common patterns where kernel fusion can be applied.

- b "scatter-add" is fused to a u_add_v primitive, which is implemented using high performance SDDMM. Likewise the same pattern applies to d.
- d "u_mul_e-sum" is fused to a $u\_mul\_e\_sum$ primitive, which is implemented using high performance SPMM.
- e For the "compute-elementwise" pattern, the injective element-wise activation function (ReLU) can be fused with the prior operation to a $u\_mul\_e\_sum\_relu$ primitive (Chen et al., 2018b).

4.3 Extended Primitive Set

The primitives provided by existing GNN frameworks are largely simple variants of SPMM and SDDMM operations using a single binary operator (add, multiplication, etc) and an aggregation operator (sum, mean, max, etc), and which are hard-coded using C++ templates. By extending the programming interface proposed in graph computing frameworks (Wang et al., 2016) with feature dimension parallelism from neural network computations (Hu et al., 2020b), we create a versatile template for extending the primitives set. While the template covers all the primitives we have used so far, we leave generalizing it for additional graph operations and performance optimization as future work.

Table 1. Graph datasets.

Dataset	Vertex	Edge	Feature
Cora	2708	10,556	1433
Pubmed	19,717	108,365	500
PPI	56,944	1,644,208	700
ogbn-proteins	132,534	79,122,504	8
ogbn-arxiv	169,343	1,166,243	128
Reddit	232,965	114,615,892	602

5 Evaluation

5.1 Experimental Setup and Methodology

Benchmark model set and datasets. Our evaluation is performed using a set of representative GNN models which cover different architectures and a wide range of application domains. Among them, GCN (Kipf & Welling, 2017) is the most widely used graph convolutional network for learning on graph-structured data; a two layer GCN with hidden size 32 is used for evaluation. GAT (Velickovic et al., 2018) introduces an attention mechanism to significantly improve accuracy in many applications; a two layer GAT with hidden size 32 is used for evaluation. ConstrainedGAT (Wang et al., 2019a) is an improved version of GAT. Finally, GCNN (Gasse et al., 2019) has been used for combinatorial optimization. To make our evaluation more realistic, we collected, largely via the DGL forum, implementations of these models written by DGL users, and made only basic modifications, such as adding type hints for UDF signatures. We focus our evaluation on model inference. While there is no fundamental reason Graphiler cannot be used for GNN training, adding training support requires more engineering effort for operators, including DFG representations and auto-differentiation, and thus we leave these as future work.

To validate the speedups we obtain on different graphs, we evaluated all models on six popular datasets listed in Table 1. Cora (Sen et al., 2008), Pubmed (Sen et al., 2008) and ogbn-arxiv (Hu et al., 2020a) are citation network datasets; PPI (Zitnik & Leskovec, 2017) and ogbn-proteins (Hu et al., 2020a) contain a number of protein-protein interaction graphs; Reddit (Hamilton et al., 2017) is a dataset which connects posts in the online forum with comments from the same users. These six datasets cover a wide range of graph sizes and sparsity levels.

Machine environment. We conducted our experiments on an AWS p3.2xlarge instance equipped with an NVIDIA Tesla V100 (16GB version) and Intel(R) Xeon(R) E5-2686 v4@2.30GHz CPUs, using Ubuntu 1804 and CUDA 11.0.

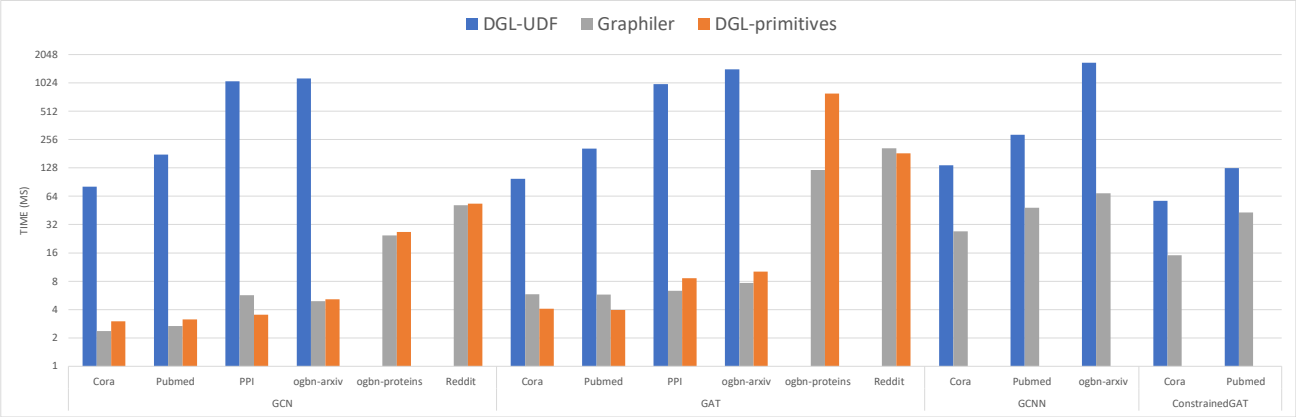


Figure 3. Inference time of GNN models on different datasets.

5.2 Overall Performance

We first demonstrate the efficiency of Graphiler by comparing it with UDFs written in DGL, in which aggregation UDFs are executed using bucketing as discussed in §2.2. As shown in Figure 3, Graphiler significantly outperforms DGL-UDF on all four models. For example, on the ogbn-arxiv dataset it achieves speedups of 232×, 186× and 24× compared to GCN, GAT and GCNN, respectively, while on Pubmed it is 3× faster than ConstrainedGAT. We believe Graphiler can further accelerate the latter two models as we add more optimization patterns. Our speedup increases as the graphs get larger because the inefficiency of DGL's bucketing approach increases as the variation in node degrees grows.

The missing blues bars in Figure 3 indicate out of memory errors when executing DGL-UDF on large graphs, and arise primarily because of edge data materialization. On GCN and GAT, Graphiler significantly reduces memory usage through kernel fusion and enables successful execution on large datasets such as obgn-proteins and Reddit. Lastly, while Graphiler also reduces memory consumption for GCNN and ConstrainedGAT, the large amount of memory these models use for edge features precluded their use on obgn-proteins and Reddit.

Figure 3 shows orange "DGL-primitives" bars for GCN and GAT because these two popular models have been carefully optimized and wrapped as built-in modules by DGL developers. The comparison between these primitives and Graphiler demonstrates that we can achieve competitive performance on most datasets compared to hand optimized implementations.

5.3 Breakdown Analysis

To better understand how different optimizations in Graphiler contributed to its performance gain, we conducted a breakdown analysis on GAT.

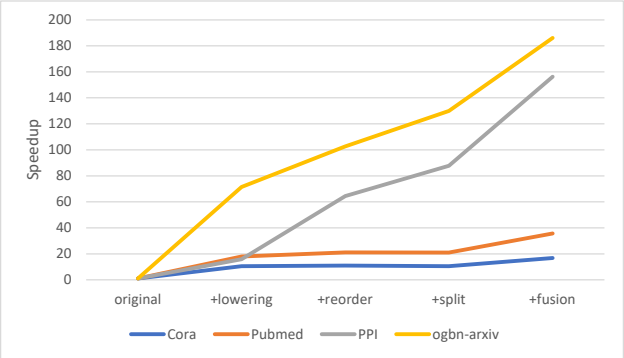


Figure 4. Speedup Breakdown.

As shown in Figure 4, by simply lowering the aggregation UDF, Graphiler can already achieve significant speedups, *e.g.* by 70× on ogbn-arxiv. Other optimizations also contribute noticeably to the execution speed, with their utility varying by graph size and density. For optimizations reducing redundant computation, such as operator reordering and splitting, the contribution on small datasets such as Cora and Pubmed is limited. On larger graphs with higher density and thus greater redundancy, these techniques produce significant speedups, *e.g.* by 5.5× on PPI. Finally, since graph operations are usually memory bound, kernel fusion accelerates GAT on all datasets, *e.g.* by up to 1.7× on PPI, through reducing memory accesses.

6 Future Work

We are actively working on three parts to make Graphiler more powerful: incorporating message passing semantics from heterogeneous graphs to accelerate popular hetero-GNN models, developing user friendly tools to register new pattern substitution rules for optimization passes, and generating high performance kernel code for additional graph operations used in GNNs.

REFERENCES

Paddle Graph Learning, 2019. https://github.com/PaddlePaddle/PGL.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Chen, H., Engkvist, O., Wang, Y., Olivecrona, M., and Blaschke, T. The rise of deep learning in drug discovery. *Drug discovery today*, 23(6):1241–1250, 2018a.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., et al. {TVM}: An automated end-to-end optimizing compiler for deep learning. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 578–594, 2018b.

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

Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. Exact combinatorial optimization with graph convolutional neural networks. In *NeurIPS*, 2019.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In Precup, D. and Teh, Y. W. (eds.), *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1263–1272, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/gilmer17a.html.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, pp. 1025–1035, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

Hu, W., Fey, M., Zitnik, M., Dong, Y., Ren, H., Liu, B., Catasta, M., and Leskovec, J. Open graph benchmark: Datasets for machine learning on graphs. In Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., and Lin, H. (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020a. URL https://proceedings.neurips.cc/paper/2020/hash/fb60d411a5c5b72b2e7d3527cfc84fd0-Abstract.html.

Hu, Y., Ye, Z., Wang, M., Yu, J., Zheng, D., Li, M., Zhang, Z., Zhang, Z., and Wang, Y. Featgraph: a flexible and efficient backend for graph neural network systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2020b.

Huang, K., Zhai, J., Zheng, Z., Yi, Y., and Shen, X. 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*, pp. 119–132, 2021.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 47–62, 2019.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Li, Z., Chen, Q., and Koltun, V. Combinatorial optimization with graph convolutional networks and guided tree search. In *NeurIPS*, 2018.

Ma, L., Xie, Z., Yang, Z., Xue, J., Miao, Y., Cui, W., Hu, W., Yang, F., Zhang, L., and Zhou, L. Rammer: Enabling holistic deep learning compiler optimizations with rtasks. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 881–897. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/ma.

Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Sen, P., Namata, G., Bilgic, M., Getoor, L., Galligher, B., and Eliassi-Rad, T. Collective classification in network data. *AI Magazine*, 29 (3):93, Sep. 2008. doi: 10.1609/aimag.v29i3.2157. URL https://ojs.aaai.org/index.php/aimagazine/article/view/2157.

Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net,

2018. URL https://openreview.net/forum?id=rJXMpikCZ.

Wang, G., Ying, R., Huang, J., and Leskovec, J. Improving graph attention networks with large margin-based constraints. *arXiv preprint arXiv:1910.11945*, 2019a.

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

Wang, Y., Davidson, A., Pan, Y., Wu, Y., Riffel, A., and Owens, J. D. Gunrock: A high-performance graph processing library on the gpu. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12, 2016.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983, 2018.

Zitnik, M. and Leskovec, J. Predicting multicellular function through multi-layer tissue networks. *Bioinformatics*, 33 (14):i190–i198, 2017.