uGrapher: High-Performance Graph Operator Computation via Unified Abstraction for Graph Neural Networks

Yangjie Zhou
yj_zhou@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Jingwen Leng*
leng-jw@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Yaoxu Song
Richard_K@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Shuwen Lu
lushuwen@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Mian Wang
ryanwang1996@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Chao Li
lichao@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Minyi Guo*
guo-my@cs.sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Wenting Shen
wenting.swt@alibaba-inc.com
Alibaba Group
Hangzhou, China

Yong Li
jiufeng.ly@alibaba-inc.com
Alibaba Group
Hangzhou, China

Wei Lin
weilin.lw@alibaba-inc.com
Alibaba Group
Beijing, China

Xiangwen Liu
vicki.liuxw@alibaba-inc.com
Alibaba Group
Beijing, China

Hanqing Wu
hanqin.wuhq@alibaba-inc.com
Alibaba Group
Beijing, China

ABSTRACT

As graph neural networks (GNNs) have achieved great success in many graph learning problems, it is of paramount importance to support their efficient execution. Different graphs and different operators present different patterns during execution. However, there is still a gap in the existing GNN acceleration research to explore adaptive parallelism. We show that existing GNN frameworks rely on handwritten static kernels, which fail to achieve the best performance across different graph operators and input graph structures. In this work, we propose *uGrapher*, a unified interface that achieves general high performance for different graph operators and datasets. The existing GNN frameworks can easily integrate our design for its simple and unified API. We take a principled approach that decouples a graph operator's computation and schedule to achieve that. We first build a GNN-specific operator abstraction that incorporates the semantics of graph tensors and graph loops. We explore various schedule strategies based on the abstraction that can balance the well-established trade-off relationship between parallelism, locality, and efficiency. Our evaluation shows that *uGrapher* can bring up to 29.1× (3.5× on average) performance improvement over the state-of-the-art baselines on two studied NVIDIA GPUs.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**.

KEYWORDS

Graph Neural Networks, AI Frameworks, Graphics Processing Unit

ACM Reference Format:
Yangjie Zhou, Jingwen Leng, Yaoxu Song, Shuwen Lu, Mian Wang, Chao Li, Minyi Guo, Wenting Shen, Yong Li, Wei Lin, Xiangwen Liu, and Hanqing Wu. 2023. uGrapher: High-Performance Graph Operator Computation via Unified Abstraction for Graph Neural Networks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575723

1 INTRODUCTION

In recent years, graph neural networks (GNNs) have emerged as a graph computing paradigm and achieved great success in machine learning for the graph field. Existing deep neural networks (DNNs) can only handle structured data such as images, sounds, and text. However, many real-world applications have the input of non-Euclidean graphs, such as chemistry [12], neurology [27], electronics [8], and social networks [9, 10], for which DNNs are not applicable. In contrast, GNNs can extend DNNs' learning capabilities to these graph-related tasks and have led to significant breakthroughs [3, 47, 52].

*Jingwen Leng and Minyi Guo are the corresponding authors.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9916-6/23/03…$15.00
https://doi.org/10.1145/3575693.3575723

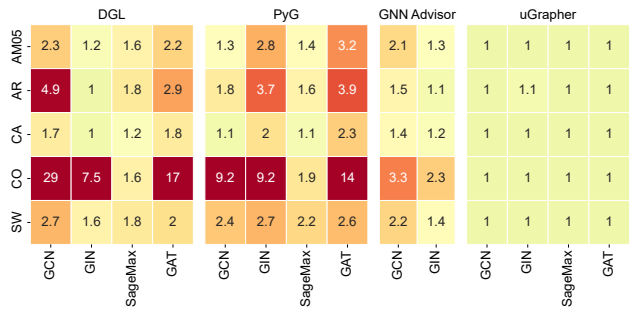


Figure 1: The normalized latency comparison of DGL, PyG, GNNAdvisor, and *uGrapher* (this work) with different models (x-axis) and graph datasets (y-axis) on V100 GPU. In the heatmap, lower numbers are better, with one being the fastest for the given model and dataset.

There is a huge architectural space for GNN models that are constantly evolving. As a result, the variability and complexity of the graph operators used by GNNs rapidly increase. For example, from the early GCN [26] to the later GAT [43] and GIN [48], the number of graph operators in GNN models increases significantly. Meanwhile, these new graph operators also become more complex, which makes the high-performance GNN execution more challenging [1].

Besides the large design space of graph operators, GNN models also operate on different graph structure datasets with distinctive characteristics such as density and cluster locality. As such, traditional graph processing systems [41, 45] explore the adaptive parallelization patterns in different scenarios to obtain high performance. However, such adaptivity is still absent for GNNs, which demonstrate different patterns and bottlenecks for different graph datasets and graph operators [32]. Meanwhile, GNNs differ from traditional graph algorithms as they do not have the complex control flow brought by frontier, but involve the traversal of feature dimensions and more complex computations while traversing the graph.

Current frameworks rely on handwritten implementation to execute graph operators incompatible with static kernels. However, these handwritten kernels only perform well for a specific GNN model and input graph dataset. They have sub-optimal performance when the GNN model and its input graph dataset change. Fig. 1 compares the normalized execution latency of DGL [44], PyG [11], and GNNAdvisor [46] for several representative GNN models and different graph datasets on the V100 GPU. Our results confirm that all studied frameworks can only achieve the best performance for a limited range of GNN models and datasets.

The reason for the above performance variability problem is that these frameworks all use a fixed execution strategy for different graph operators and input graphs. However, achieving the best performance for different GNN models and input graph structures requires a dynamic trade-off between locality, parallelism, and work efficiency. As the input to GNN models, graphs vary significantly in terms of the number of vertices, the number of edges, sparsity, size of input feature, and distribution characteristics of edges inside the graph [24, 46, 51]. Meanwhile, graph operators in different GNN models have distinctive computational and memory access characteristics [20]. For example, the graph operator in GCN [26] accumulates each vertex's feature embedding from its neighbors. In contrast, the graph operator in GAT [43] first calculates a weight for each edge and then performs a weighted accumulation. Thus, a fixed execution strategy yields varying performance results for different GNN models and datasets.

Table 1: Comparison of existing graph operator acceleration methods with our proposed uGrapher.

Work	Parallelization Strategy	Extension Overhead	Scalibility
GNNAdvisor [46]	Static	Handwritten CUDA	Low
GE-SpMM [19]	Static	Handwritten CUDA	Low
FeatGraph [18]	Static	TVM Template	Middle
uGrapher(Ours)	**Adaptive**	**Simple Op Info**	**High**

In this work, we propose *uGrapher*, a unified and high-performance interface for supporting graph operators that can be easily integrated into existing GNN frameworks. We take a principled approach that decouples the computation and schedule of a graph operator to adapt to the dynamics of different GNN operators and datasets. We analyze all the graph operators and abstract them as a unified form of nested sparse-dense for-loops. Specifically, the innermost level of the nested loop captures the semantics of different graph operators, while the outermost loop offers the opportunity for a unified and comprehensive parallelization space exploration. Based on the unified abstraction, we explore various execution strategies and their trade-off relationships of different loop transformations corresponding to different graph operators on GPUs.

Meanwhile, *uGrapher* also provides a unified and easy-to-use API for the upper-level GNN framework, for which the underlying performance tuning is transparent. We conduct a comprehensive analysis of *uGrapher* performance using fifteen real-world graph datasets with different embedding sizes and structures and six typical GNN models with different computational characteristics. We compare the performance of our work with current state-of-the-art GNN frameworks. The experimental results in Fig. 1 show that our design achieves optimal results in almost all scenarios and near-optimal results in the rest cases, compared to the existing frameworks. With the flexible and adaptive nature of *uGrapher*, we believe that our work can be useful for supporting the high-performance acceleration of fast-evolving GNN models and datasets.

Besides providing high-performance for a wide range of graph operators and datasets, *uGrapher* is extremely flexible and scalable compared to existing manual optimization methods. As shown in Tbl. 1, existing solutions only support fixed parallelization strategies, while for graph operator extensions, GE-SpMM [19] and GNNAdvisor [46] need to rewrite handwritten CUDA code completely, and FeatGraph [18] requires users to provide new TVM templates. In contrast, our proposed *uGrapher* can provide high-performance automatic CUDA code generation for all graph operators in GNNs with just minimal operator information.

Overall, our work makes the following contributions:

- We analyze the inefficiency of existing GNN frameworks at the kernel level for different graph operators and datasets.

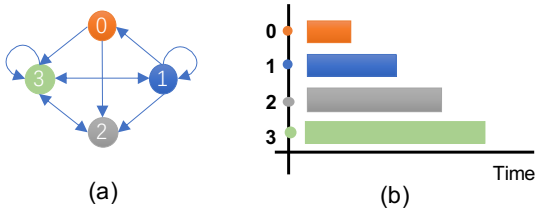


Figure 2: (a) An example graph structure; (b) Workload imbalance using a simple fixed mapping strategy.

- We propose a unified abstraction for all the graph operators in GNNs, which also defines a comprehensive optimization space for different parallel execution strategies on GPUs. We are the first work that exploits adaptive parallelization strategies for high performance in different graphs and different operators of GNNs.
- Based on the unified abstraction, we are able to automatically provide high-performance generation of cuda code for all graph operators, requiring only simple operator information, which brings significant flexibility and scalability.
- We design a unified API *uGrapher*, which can support all the graph operators in existing frameworks and explore their optimal parallel execution strategy on different graph datasets. Practically, we implement the integration of uGrapher with existing frameworks. Based on *uGrapher*, we are able to improve the performance of existing frameworks by 3.5× on average.

2 BACKGROUND AND MOTIVATION

This section presents a brief background on the existing frameworks, including DGL [44] and PyG [11] for Graph Neural Networks (GNNs), with an emphasis on their programming interface and execution strategy. We then demonstrate their execution inefficiency through experiments on GPUs.

2.1 Graph Neural Networks

In recent years, GNNs have received wide attention from academia and industry for their powerful learning and inference capabilities for graph structures in non-Euclidean spaces [47]. The output of a GNN model is a d-dimensional embedding vector for each node in the input graph. For vertices with similar properties (e.g., subgraph structures), their embeddings are also close to each other, enabling fast reasoning about graph-related problems [14, 26].

GNN Model. To obtain these embeddings, GNNs combine DNN-based feature transformation and graph-based operations that propagate and aggregate information along the graph structure. Owing to the mixture of DNN operations and graph operations, existing GNN frameworks like DGL [44] and PyTorch-Geometric (PyG) [11] extend the existing DNN frameworks like TensorFlow and PyTorch with the key concept of message, which is an intermediate feature embedding associated with each edge. We use the following equations to formalize graph operations centering around the message.

For any operation on a graph $G = (V, E)$, it can be classified into three stages, namely *message creation, message aggregation, and feature update*, based on the properties of the data and the direction of data movement as following:

Table 2: Classification of Graph operators. 'V' means vertex embedding tensor and 'E' means edge embedding tensor.

Operator Type	Message Creation			Message Aggregation	Fused Aggregation	
Input Type	V	E	V&E	E	V	V&E
Output Type	E	E	E	V	V	V
Operator Count	11	1	20	4	44	80

$$m_e = \text{Message\_Creation}\left(h_u, h_v, w_e\right), (u, e, v) \in \mathcal{E} \quad (1)$$

$$h_v = \text{Message\_Aggregation}\left(\{m_e : (u, e, v) \in \mathcal{E}\}\right) \quad (2)$$

$$x_v^{new} = \text{Combination}\left(h_v, h_v\right), v \in \mathcal{V}, \quad (3)$$

where u and v are vertex (or node) indices, e is the index for the edge between u and v; h_v refers to the feature embedding of vertex v, and m_e is the message associated with the edge e.

In Equation (1), each edge *creates* its message m_e by applying an edge-wise message function to its own edge feature and associated vertex features. In Equation (2), each vertex aggregates the messages from incoming edges using an *aggregation* function. In Equation (3), each vertex updates its features using a vertex-wise *combination* function. In GNNs, the collection of the feature embeddings of all vertices (edges) is called vertex (edge) embedding tensor.

Graph Operator Definition. We define the graph operators as ones that need to traverse the input graph structure. The *message-creation* and *message-aggregation* explained above are two types of graph operators. When the message-creation operator is a simple copy operation, it can be fused into message-aggregation operator to avoid redundant accesses, which is used by both DGL and PyG. As such, there exists a third type graph operator called *fused-aggregation* operator that fuses the original message creation and message aggregation operators. [1]

In short, graph operators include *message-creation* and *message-aggregation*, and *fused-aggregation* operators. They include both irregular memory behaviors due to graph structures and complex arithmetic computations, thus introducing a critical challenge to high-performance GNN computation. Therefore, the computational optimization of graph operators becomes the optimization scope of our work.

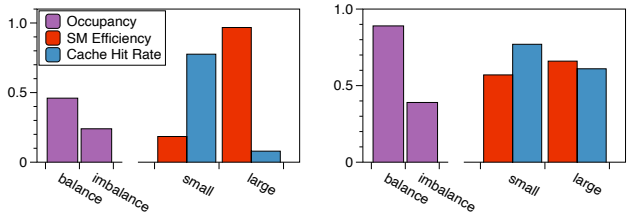
Complexity of Graph Operators. Different GNN models use different graph operators, which have a large design space. Tbl. 2 categorizes the 160 graph operators supported by DGL according to their input and output tensor types. Even with the same input/output tensors, graph operators can perform different computation patterns. Therefore, providing practical high-performance support for all these operations is challenging and requires a systematic and automatic solution.

Variability of Graph Data. Real-world graph datasets also have a large variability. As shown in the Tbl. 3, we select 15 commonly used graph datasets for our analysis. We collect the number of vertices and edges of different graphs to reflect the size scale of the graph. We also derive the standard deviation of non-zeros ("std of nnz" column) in rows of adjacency matrix, which reflects the degree of

[1] In this paper, the *aggregation* operator refers to the *fused-aggregation* operator, if not explicitly specified.

Table 3: Details of graph datasets used for evaluation.

dataset	#Vertex	#Edge	std of nnz	#Feature	#Class
cora(CO)	2708	10556	5.23	1433	7
citeseer(CI)	3327	9228	3.38	3703	6
pubmed(PU)	19717	99203	7.82	500	3
PROTEINS_full(PR)	43466	162088	1.15	29	2
artist(AR)	50515	1638396	63.47	100	12
ppi(PP)	56944	818716	23.29	50	121
soc-BlogCatalog(SB)	88784	2093195	206.81	128	39
com-amazon(CA)	334863	1851744	5.76	96	22
DD(DD)	334925	1686092	1.69	89	2
amazon0601(AM06)	403394	3387388	15.28	96	22
amazon0505(AM05)	410236	4878874	15.05	96	22
TWITTER-Partial(TW)	580768	1435116	1.52	1323	2
Yeast(YE)	1710902	3636546	0.75	74	2
SW-620H(SW)	1888584	3944206	1.16	66	2
OVCAR-8H(OV)	1889542	3946402	1.16	66	2



(a) weighted-aggr-sum kernel. (b) unweighted-aggr-max kernel.

Figure 3: Evaluation of DGL kernel limitation. The feature size for all experiments is 32, and the results are collected via nvprof on V100.

graph balancing. Different graph datasets also have diverse features and class sizes, which affects the memory usage and computation complexity of some graph operators. As can be seen from the table, the properties vary significantly between the different graphs.

2.2 Execution Efficiency Analysis on GPU

We choose Nvidia GPUs [35, 37] with CUDA programming language as the target hardware in this paper. The GPU architecture is highly parallel, and has many streaming multiprocessors (SMs). An SM executes threads in the SIMT (Single Instruction Multiple Threads) fashion, and a warp with 32 threads run simultaneously. The massive computing and memory resources make the GPU increasingly important for deep learning acceleration [42]. We show that, due to the lack of systematic optimization methodology, the underlying CUDA kernels used by existing GNN frameworks suffer from inefficiency and inflexibility. We use only DGL in this subsection, but the kernels in PyG have similar issues.

The DGL calls static CUDA kernels to support the message passing programming interface shown in the previous subsection, which does not adaptively adjust to different computing scenarios. We analyze their inefficiencies as below.

We choose two graph operators commonly used in GNN for quantitative analysis. The first one is the *weighted-aggr-sum* graph operator in GCN [26] and GAT [43], and the other one is the *unweighted-aggr-max* in SageMax [14], the former is heavier in access and computation than the latter due to the addition of edge weights. We use AR and SO as representatives of imbalance graphs, and PR and DD as representatives of balance graphs, and collect their occupancy metric via nvprof [36]. Furthermore, we also used CO and CI as representatives of small graphs, and SW and OV as examples of large graphs, and collected their sm efficiency and L2 cache hit under different operators.

The results are shown in the Fig. 3, and there are some similar patterns of results in both different operators. The occupancy is significantly lower for the imbalance graphs compared to the balance graphs. Moreover, smaller graphs get higher L2 cache hit rate while obtaining lower sm efficiency compared to larger graphs. Additionally, there is a variation in the results between the operators. For the lightweight *unweighted-aggr-max*, the difference in occupancy results between imbalance and balance graphs is larger, but the difference in sm efficiency and cache hit rate between small and large graphs is relatively smaller.

These results imply that when executing imbalance graphs, the low occupancy of the GPU leads to the underutilization of hardware resources. And when executing small graphs, GPU performance is usually bounded by the underutilization of hardware resources due to parallelism, while access bandwidth becomes a bottleneck due to low locality when executing large graphs. Meanwhile, these metrics can vary from operators.

Summary. Existing GNN frameworks rely on handwritten kernels with fixed execution strategies, which have inefficiencies for executing GNN models owing to the diversity of graph-related operations and the diversity of real-world graph structures. This motivates us to design a unified interface *uGrapher* for supporting existing GNN frameworks. Our unified interface captures the complete semantic representation of all common graph operators in GNNs, while enabling different dynamic and flexible execution strategies for input graph data and graph operators with different characteristics.

3 UNIFIED GRAPH OPERATOR ABSTRACTION

Previous works only decompose GNNs into different stages [11, 31, 44], but lack modeling abstraction of the underlying graph-related operators. This section describes our unified abstraction for all the graph operators in current GNN models, which adopts the nested sparse-dense loops. We first start with the *aggregation-sum* operator as an example to illustrate our abstraction. We then describe its generalization capability to represent all the graph operators.

3.1 Nested For-Loop Notation for Graph Operators

We use the same example of *aggregation-sum* operator in Sec. 2.2 to illustrate our graph operator abstraction. This operator is widely used in GNNs, and for each vertex in the graph, the operator traverses its neighboring vertices and accumulates their feature embeddings.

As shown in Fig. 4, it is natural to use the nested-loops to represent the *aggregation-sum* operator. The graph operator abstraction consists of three loops, where Line 5 and 6 represent the traversal of all vertices in the graph and their respective incoming edges, and

```
1  Input: Graph G=(V,E), Vertex Embedding Tensor X[V][F]
2  Output: Vertex Embedding Tensor Y[V][F]
3
4  Aggregation-Sum:
5      for dst in V:
6          for edge in dst.get_inedges():
7              src = edge.src_v
8              for feat in F:
9                  Y[dst][feat] += X[src][feat]
```

Figure 4: The nested-loop-based *aggregation-sum* graph operator.

Line 8 represents the traversal in the feature dimension. The innermost statement (Line 9) implements the combined accumulation of the data from the source vertex towards the destination vertex.

Our abstraction also incorporates several GNN-specific data structures that capture the operator's graph-level semantics. In Fig. 4, the inputs to the *aggregation-sum* graph operator are the graph G and the vertex feature embedding tensor X, which outputs a new vertex embedding tensor Y. A graph is a pair formed by combining two sets, where V and E denote the sets of all vertices and all edges inside the graph. Each element of the set V represents a vertex, and each vertex can get the incoming and outgoing edges of the vertex by *get_inedges()* and *get_outedges()* interface, respectively. Each element of the set E represents an edge, which is a pair of vertices, and the corresponding source and destination vertices can be obtained by *src_v* and *dst_v*.

3.2 Unified Abstraction Design

We analyze all graph operators in GNNs and find that they share a common pattern. Specifically, they contain the following three execution stages: moving data from vertices to edges, performing the edge-wise computation for all edges, and executing a reduction function from edges to their associated vertices. Different operators perform different edge-wise and reduction computation, and may skip certain stages.

For example, the *aggregation-sum* operator in the *SageSum* [14] model simply copies each edge's source vertex feature to form the edge feature, and performs no edge computation. For each vertex, it then reduces the edge features of all its incoming edges to a new vertex feature. By contrast, *GAT* [43] model contains several graph operators with other different computation patterns. The first *message-creation* operator is very lightweight, where the features of the source vertex and destination vertex of each edge are summed as edge feature for calculating the attention weight, skipping the final reduction stage. In contrast, the second *aggregation-sum* operator involves the computations in all three stages. This operator first copies features from the source vertex, then performs an edge-wise multiplication with the previously generated edge weights, and lastly reduces the transformed edge features to vertex features. As such, the second operator is more computation-heavy than the first operator.

Given the similarities and differences in these graph operators, we keep the nested loop as the basis of our graph operator abstraction and allow users to customize input tensors and element-wise

```
1  edge_op_list = [copy_lhs, copy_rhs, mul, add, sub, div]
2  gather_op_list = [copy_lhs, copy_rhs, sum, max, min, mean]
3  tensor_type_list = [Src_V, Dst_V, Edge]
4  type_idx_dict = {Src_V: src, Dst_V: Dst, Edge: edge, NULL:
                    NULL}
5
6  Input: edge_op, gather_op, Tensor A, Tensor B, Tensor C,
       Graph G=(V,E)
7  Output: Tensor C
8
9  Unified Abstraction:
10     a_idx = type_idx_dict[A.type]
11     b_idx = type_idx_dict[B.type]
12     c_idx = type_idx_dict[C.type]
13
14     for dst in V:
15         for edge in dst.get_inedges():
16             src = edge.src_v
17             for feat in F:
18                 edge_tmp = edge_op(A[a_idx][feat],
                                      B[b_idx][feat])
19                 C[c_idx][feat] = gather_op(C[c_idx][feat],
                                              edge_tmp)
```

Figure 5: Our unified graph operator abstraction.

operations to represent different operators. Fig. 5 gives the details of unified abstraction. Compared to the *aggregation-sum* representation in Fig. 4, the nested loops remain identical, but the innermost code block introduces two additional dynamic operators: *edge_op* and *gather_op*, which can be defined by users.

The *edge_op* implements the edge-wise computation on each edge, while *gather_op* implements the edge-to-vertex reduction operation. For example, to represent the *aggregation-sum* in Fig. 4, the two functions can be set to *copy_lhs* (i.e., copy from the left-hand side) and *copy_rhs* (i.e., copy from the right-hand side), respectively.

Besides the input of *edge_op*, *gather_op* and graph structure G, the unified abstraction in Fig. 5 also requires three additional input embedding tensors. To maintain the flexibility to represent different graph operators, the types of these three embedding tensors can be any of the following ones: source vertex embedding tensor (*Src_V*), destination vertex embedding tensor (*Dst_V*), edge embedding tensor (*Edge*), and *NULL*. The different data types also determine the different addressing patterns in the loop computation (Line 10 to 12). For instance, the output tensor Y of *aggregation-sum* in Fig. 4 corresponds to the C tensor with destination vertex feature type, while its addressing dimension of Line 9 is always based on *dst*.

In summary, the combination of *edge_op* and *gather_op*, together with the tensor types in A, B, C capture the complete semantics of a graph operator, including its computation and memory movement patterns. The equation below formally defines our unified abstraction, with ψ as the *edge_op* function, and ρ as the *gather_op* function.

$$C_{c\_idx,f} = \sum_{dst \in V} \sum_{src \in V} \sum_{f \in F} \psi(C_{c\_idx,f}, \rho\left(A_{a\_idx,f}, B_{b\_idx,f}\right))$$

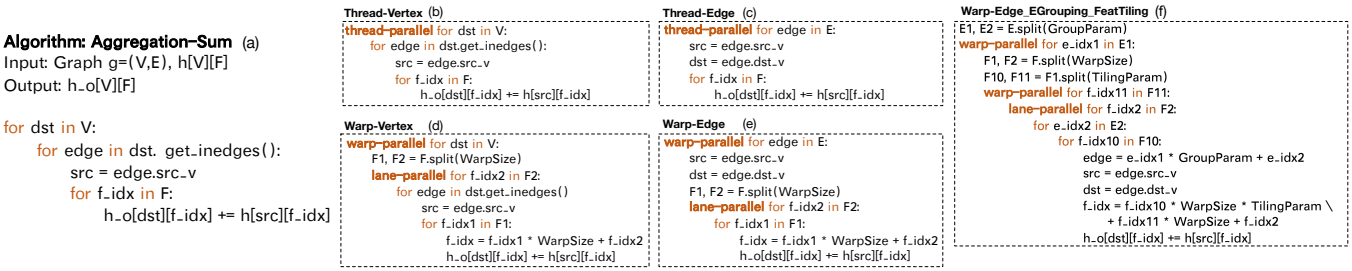


Figure 6: Examples of different parallelization strategies for the *aggregation-sum* operator.

Table 4: Complete graph operator representation of *uGrapher*.

Graph Op	edge_op	gather_op	A	B	C
Message Creation	copy_lhs	copy_rhs	V	Null	E
	copy_rhs	copy_rhs	Null	V	E
	add/sub/mul/div	copy_rhs	V	V	E
			V	E	E
			E	V	E
Message Aggregation	copy_lhs	sum/max/min/mean	E	Null	V
	copy_rhs	sum/max/min/mean	Null	E	V
Fused Aggregation	add/sub/mul/div	sum/max/min/mean	E	E	V
	add/sub/mul/div	sum/max/min/mean	V	V	V
			V	E	V
			E	V	V

The complete implementation of all graph operation semantics and the corresponding parameter configurations are shown in Tbl. 4. Thus it can be seen that our unified abstraction, for the message creation and message aggregation, along with the graph semantics after fusion, are all supported, which gives us the basis for implementing a unified interface.

4 OPTIMIZATION SPACE ANALYSIS

In this section, we identify the optimization space that is critical to achieve high-performance graph operator execution. Specifically, we explore the tradeoff space for different parallelization strategies of graph operator execution on GPUs, and demonstrate that the optimal strategy varies for different datasets and different graph operators.

4.1 Tradeoff Space

We first describe the tradeoff space that impacts the performance of graph operator on GPUs. Specifically, we focus on the well-understood three-dimensional optimization space: locality, parallelism, and work-efficiency [16, 29, 51].

Locality describes the quantity of spatial and temporal reuse in a program. The better locality increases the cache hit rate and potentially improves the program performance. The GPU contains the per-SM (streaming multiprocessor) L1 cache and shared L2 cache. To improve the locality of graph operators, we can apply tiling or blocking to the nested loop, which restricts the working set of each SM.

Parallelism refers to the amount of computations that can be performed concurrently. Modern GPUs typically contain thousands of computing units, so higher parallelism can improve hardware resource utilization, hide memory access latency, and thus improve program performance. The simplest way for increasing the parallelism of graph operators is to launch more threads, warps, or thread blocks.

Work-efficiency is expressed as the inverse of the amount of overhead. Different execution strategies for the same operator may introduce additional computations such as address calculations. Meanwhile, for executing graph operators in GPUs, atomic instructions are required when write conflict exists, which introduces lock overhead and hence detriments the work-efficency. For example, it is possible to map an edge to each thread. Since different edges can share the same vertices, atomic addition instruction is necessary when performing the accumulation reduction from edge features to vertex features.

Locality, parallelism and work-efficiency form an impossible triangle, meaning there is no single strategy that improves these three metrics simultaneously. In the next subsection, we describe different execution strategies based on our unified abstraction that lets us explore this space systematically.

4.2 Parallelization Strategies Exploration

We now detail the different parallelization strategies and show that they have both positive and negative impacts on various metrics in the aforementioned trade-off space. Given the diversity of graph operators and graph datasets characteristics, we demonstrate that a fixed parallelization strategy only leads to optimal performance in a few cases.

We use the previously explained *aggregation-sum* graph operator in Fig. 6 as a representative example to illustrate the impact of various parallelization strategies on the three trade-off metrics. We first follow two classical parallelization strategies used in existing graph processing systems: vertex-parallel [15] and edge-parallel [40], whose GPU implementation means that one thread handles all the computations of a vertex or an edge. As such, we define them as *thread-vertex* and *thread-edge*, where different threads execute in parallel. Since the edge count in a graph is usually much greater than the vertex count, *thread-vertex* reduces parallelism compared to *thread-edge*, but improves the reuse of output data and hence locality. Meanwhile, *thread-edge* reduces the work-efficiency

Table 5: Comparison of the parallelization strategy space in our work against previous works.

Parallelization Strategy	Warp Vertex	Warp Edge	Thread Vertex	Thread Edge	V/E Grouping	Feature Tiling	Adaptive Pattern
GNNAdvisor		✓			✓	✓	
GE-SpMM	✓						
FeatGraph			✓			✓	
Ours	✓	✓	✓	✓	✓	✓	✓

Table 6: The tradeoff of different parallelization strategies.

Parallelization Strategy	Locality	Parallelism	Work-efficiency
Thread-Edge	→	→	→
Warp-Edge	↓	↑	→
Warp-Vertex	↑	↓	↑
Thread-Vertex	↑↑	↓↓	↑
V/E-Grouping	↑	↓	↓
Feature Tiling	↓	↑	↓

because multiple threads can update the same vertex and thus require atomic update operations.

Meanwhile, since the vertex/edge features in GNNs are vector, while the traditional graph processing algorithms like PageRank use scalar values, we use this GNN-specific feature dimension parallelization strategy called *warp-vertex* and *warp-edge* in Fig. 6(d)(e). In these strategies, each warp (a group of 32 threads in GPU) only processes a single vertex or edge at a time, and different threads (or lanes) in the warp process the different feature elements. Compared to *thread-vertex/edge* strategies, the *warp-vertex/edge* strategies can launch more warps and thus increases the parallelism. However, they also hurt the locality because the per-warp cache capacity is reduced.

For the above four strategies, we introduce two fine-grained parameters to further explore the tradeoff between locality and parallelism. The first parameter, which we call *V/E grouping*, means that we combine multiple edges or vertex into a group. For example, for *thread-edge*, setting this parameter to four means a thread can process four edges instead of the original one edge, which improves the locality but also reduces the parallelism. This also reduces work-efficiency owing to the additional group computation overhead.

The second parameter is *feature tiling*, which launches more threads by exploiting the parallelism from the feature dimension. For example, for the feature size of 64 and warp size of 32, setting the *feature tiling* parameter to two would make a vertex/edge map to two warps instead of one single warp when no feature tiling is applied. In contrast to *V/E grouping*, this strategy increases parallelism but reduces locality. Meanwhile, it also reduces work-efficiency because of the extra address calculation for feature tiling.

Tbl. 5 summarizes the comparison between our parallelization strategy space and the previous manual kernel optimization. It can be seen that *uGrapher* achieves flexible adaptability while providing new parallel strategies, which brings the capability to achieve higher performance for a wide set of graph operators in GNNs.

Tbl. 6 summarizes the impact of different parallelization strategies on locality, parallelism, and work-efficiency. Note that a single fixed parallelization strategy cannot simultaneously improve these three metrics.

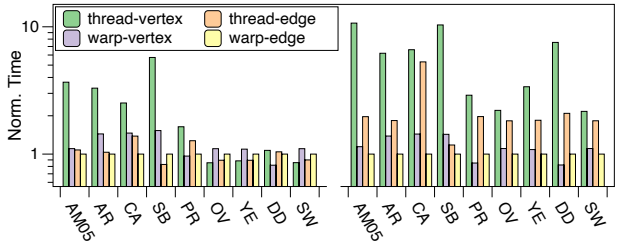


Figure 7: The optimal execution strategy for the *aggregation-sum* operator varies for different feature sizes and graph datasets. The feature size is 8 on the left, and 16 on the right.

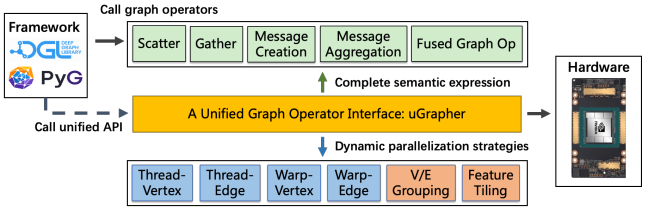


Figure 8: The overview of *uGrapher*.

4.3 Optimal Execution Strategy Analysis

In this subsection, we use the *aggregation-sum* graph operator to demonstrate that the optimal execution strategy for a given graph operator varies across datasets and feature sizes.

We measure the execution time of this graph operator under the four basic strategies (i.e., without V/E grouping nor feature tiling) under the V100 GPU. We set the feature size to 8 and 16 in this experiment. Fig. 7 shows the normalized execution time, confirming that different strategies achieve the optimal results in different cases.

5 DETAILED DESIGN OF *uGrapher*

In this section, we propose *uGrapher*, a high-performance and unified graph operator interface for GNNs, which adopts the unified abstraction discussed in Sec. 3 and incorporates the parallelization strategies described in Sec. 4. Fig. 8 shows its overview with two main features, i.e., the ability to provide complete semantic representation for various graph operators and the ability to achieve efficient execution by automatically exploring the flexible and dynamic parallelization strategies. As a result, *uGrapher* can provide specialized and optimal kernels for all GNN graph operators on different GPU architectures and graph datasets. We also describe how it can be easily integrated into existing frameworks.

5.1 Unified Interface Design

Based on the unified abstraction and various decoupled parallelization strategies, we implement our unified graph operator interface called *uGrapher* as shown in Fig. 9.

The *uGrapher* API contains three arguments: *graph_tensor*, which is the graph data; *op_info*, which passes information about the computation of *edge_op*, *gather_op*, and input tensors; and *parallel_info*, which specifies the parallelization strategy.

```
op_info = [edge_op, gather_op, Tensor_A, A_Type,
    Tensor_B, B_Type, Tensor_C, C_Type]
parallel_info = [parallel_strategy, Grouping_Param,
    Tiling_Param]

uGrapher(Graph_Tensor, op_info, parallel_info)
```

Figure 9: The details of *uGrapher* API.

```
#------------------DGL GCN--------------------#
def gcn_forward(self, graph, h, edge_weight):
    h = torch.mm(h, self.W)
    graph.srcdata['h'] = h
    graph.edata['_edge_weight'] = edge_weight
    graph.update_all(fn.u_mul_e('h', '_edge_weight', 'm'),
        fn.sum(msg='m', out='rst'))
    rst = graph.dstdata['rst']
    return torch.relu(rst + self.bias)
```

Figure 10: DGL's implementation of GCN.

The above API separates the operator computation, the graph data, and the parallelization strategy so that users can come up with their own heuristics to identify the optimal strategy for different operators and graph structures. Meanwhile, when users do not specify any parallelization strategy, our interface would currently perform an automatic tuning to find the optimal parallelization strategy, which we detail later.

5.2 Implementation of *uGrapher*

We now describe how we implement the *uGrapher* interface. Specifically, we focus on how to generate the CUDA kernels for operators defined via *uGrapher* API. At a high level, our CUDA code generator also follows the *uGrapher* design principle that fully decouples the operator's scheduling strategy from its computation.

Decoupling Scheduling and Computation. To provide full scheduling support for various graph operators, we leverage the template-based programming. We first manually implement the CUDA kernel templates for each of the parallelization strategies described in Sec. 4. We then reserve a device function interface in each of these templates to support various graph operators.

Code Generation Implementation. We implement an automated end-to-end code generation process that ensures the correctness and performs optimizations of generated CUDA kernels for different graph operators. The entire process consists of two passes and is also flexible and extensible to support future operators. The first pass fuses the two innermost code statements when the members of *op_info* such as *edge_op* or *gather_op* are NULL, thus reducing register usage and read/write overhead. The second pass generates the final device function code, where it may choose to use atomic operations by analyzing whether different threads would compete for the same data or not.

```
from uGrapher import update_all

#------------------uGrapher GCN------------------#
def gcn_forward(self, graph, h, edge_weight):
    h = torch.mm(h, self.W)
    graph.srcdata['h'] = h
    graph.edata['_edge_weight'] = edge_weight
    uGrapher.update_all(graph, fn.u_mul_e('h', '_edge_weight',
        'm'), fn.sum(msg='m', out='rst'))
    # op_info = ['mul', 'sum', h, 'Src_V', edge_weight,
        'Edge', rst, 'Dst_V']
    # parallel_info = ['warp-edge', 8, 4]
    # uGrapher(graph, op_info, parallel_info)

    rst = graph.dstdata['rst']
    return torch.relu(rst + self.bias)
```

Figure 11: Our implementation of GCN

The above design leads to the flexible and efficient implementation for different operators by freely combining global functions with device functions. The former provides support for different parallelization strategies, and the latter provides support for different arithmetics in the graph operator.

5.3 Integration with Existing Frameworks

Since existing frameworks all adopt Python-based programming interfaces [11, 44], we use pybind11 [21] to implement the Python invocation interface of CUDA kernels generated by our code generator. Meanwhile, we take DGL as an example of implementing automatic interface replacement. As such, we can keep the existing framework's code base unchanged, thus minimizing the user's burden of using *uGrapher*.

DGL implements its own specific interfaces for graph operators: *apply_edge* and *update_all*. The former is mainly used to implement message creation operators, and the latter is mainly used to implement the message aggregation or fused aggregation operators. DGL passes in the corresponding built-in function name by string, and we can recognize the passing of the graph operator interface of DGL. The program development burden for the integration with existing frameworks is limited only to the implementation of pattern recognition and switching table. Thus, we can implement *uGrapher* as the underlying interface to be called without changing the user code. The code snippets in Fig. 10 and Fig. 11 show how GCN is implemented in DGL and *uGrapher*. As we can see from the code, *uGrapher* requires only a simple substitution to achieve the functionality accomplished by the different interfaces for graph operators used by existing GNN frameworks.

5.4 Adaptive Predict Optimal Parallelization Strategies

Finding the optimal parallelization strategy can be challenging and time-consuming because there is a total number of 10^4 valid strategies for a graph operator in *uGrapher*. The exhaustive grid search would require days of time. Therefore, we leverage the gradient

Table 7: Extracted features in predict strategy.

	Graph Info			Operator Info		
Feature	#Vertex	#Edge	std_nnz	Edge_op	Gather_op	A/B/C Type

Table 8: Detailed experimental setup.

GPU	NVIDIA Tesla V100(80 SMs)	Nvidia Ampere A100 (108 SMs)
CPU	Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz
OS	Ubuntu 18.04.5 (kernel 5.4.0)	Ubuntu 20.04.2 (kernel 5.11.0)
Software	GPU Driver Version: 470.57; CUDA Version: 11.1; Pytorch Version: 1.8	

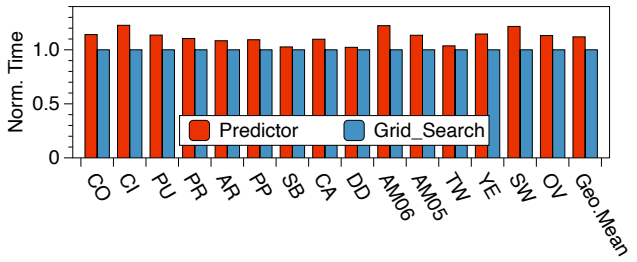


Figure 12: Comparison of the performance results obtained by the predictor and grid search for the first layer of GCN on V100.

boosting framework LightGBM [23] to train a prediction model to select the optimal strategy in parallelization space. We synthetically construct the training dataset by randomly selecting 128 graphs in the network graph dataset [39], and use the features of graph data and operator information for model training as shown in Tbl. 7. We further analyze the selection choices of the optimal strategies in different scenarios in Sec. 7.

6 METHODOLOGY

Experiments. To evaluate *uGrapher*, we use two different GPUs as our hardware platforms: Tesla V100 [37] and Ampere A100 [35]. Tbl. 8 details our experimental setup.

Baselines. We choose several baseline implementations for comparison: 1) Deep Graph Library (DGL) is the state-of-the-art GNN framework that works for multiple DL frameworks, and in our experiments, we choose PyTorch for the fair comparison; 2) Pytorch-Geometric (PyG) is another GNN framework which is built upon PyTorch; 3) GNNAdvisor by Wang et al. presents a system with the handwritten optimized kernels that aims to systematically accelerate GNNs on GPUs; In our experiments with GNNAdvisor, we keep its default configuration and disable the node renumbering optimization for a fair comparison.

Benchmarks. We choose the four representative GNN models widely that are also used by previous works: GCN [26], GIN [48], GAT [43], GraphSage [14]. We evaluate three different aggregators, max, sum, and mean in the GraphSage model. For all benchmarks, the layer, head, and hidden feature parameters follow the default configuration in the original paper for all baselines and *uGrapher*. Note that GNNAdvisor only supports the GCN and GIN models.

Batchsize. Our experiments target full-graph inference so the number of vertices in the input graph dataset is the batchsize. There is another GNN inference scenario called mini-batch inference, which performs the inference for a set of vertices in the graph. The typical execution flow is to perform sampling preprocessing first, and then execute the graph operator. As such, this falls back to full-graph inference in our case.

Datasets. We use 15 datasets that have also been used in many previous GNN-related works. The total count, sparsity, and distribution characteristics of edges vary significantly among these datasets. As such, our chosen datasets are sufficient to represent the graph in real-world scenarios. Tbl. 3 provides the detailed information for these datasets.

7 EVALUATION

7.1 Prediction Validation

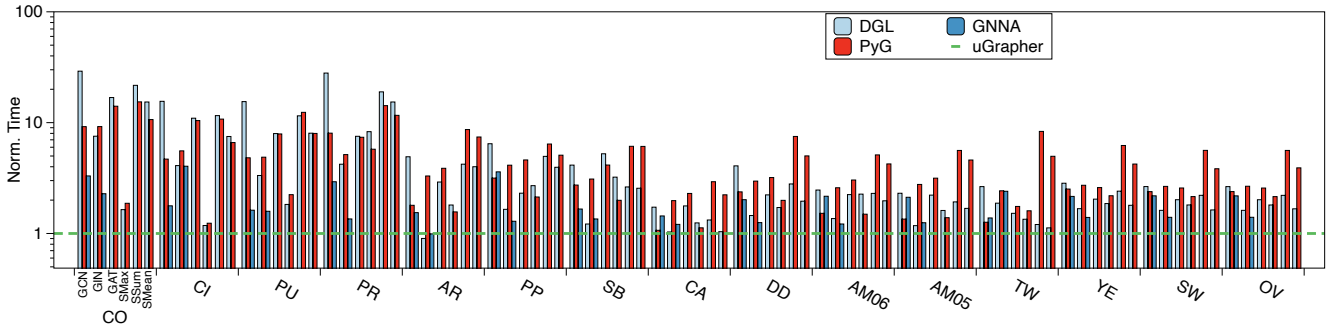
Given the rich set of parallelization strategy space, we use lightgbm [23] to train a prediction model to select the optimal strategy in parallelization space. Thus, the overhead of searching for optimal scheduling can be eliminated almost completely. To verify the effectiveness of the prediction, we compare the predicted optimal strategies with the optimal strategies found by grid search. As shown in Fig. 12, the predictor is able to achieve performance results close to those of grid search. As a result, we chose lightGBM as the default prediction method in our following experiments.

7.2 Comparisons with State-of-the-Art Frameworks

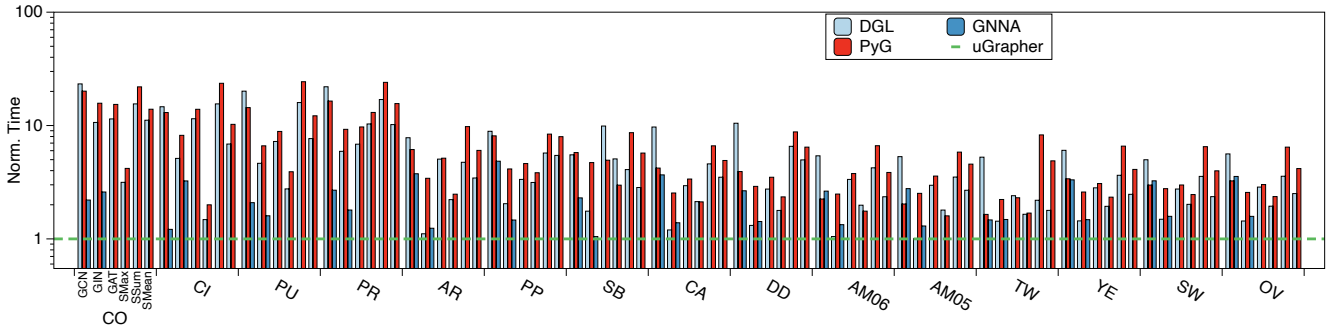
Fig. 13 shows the normalized end-to-end execution time on two GPUs. On both GPUs, *uGrapher* obtains a significant performance improvement over baselines, which comes from the ability in avoiding redundant computations while enabling flexible exploration of parallelization strategies. The geometric averaged speedup values *uGrapher* over the DGL, PyG, GNNAdvisor are 3.04, 3.75, and 1.76 on V100 and 4.07, 5.13, and 2.04 on A100, respectively.

Fig. 14 summarizes the per-model speedups on two GPUs. We find that the different speedup values for different models are mainly owing to the different execution time ratios of the graph operators in different models. As *uGrapher* only targets graph operators, we are able to achieve a greater performance improvement in models that spend a large proportion of time on graph operators, such as GCN and SageMean. In contrast, SageMax has a larger proportion of general matrix multiplication (GEMM), and its speedup is smaller.

Fig. 15 summarizes the per-dataset speedups on two GPUs. Also comparing Fig. 15a and Fig. 15b, *uGrapher* achieves a higher speedup value on the A100 GPU. The reason is that the A100 GPU provides tensor core support for the float data type, so its GEMM performance is faster than the V100 GPU. We obtain different speedup values for different datasets. It also reveals that these baselines can only achieve relatively high performance for partially limited datasets due to the lack of adaptive parallelization.

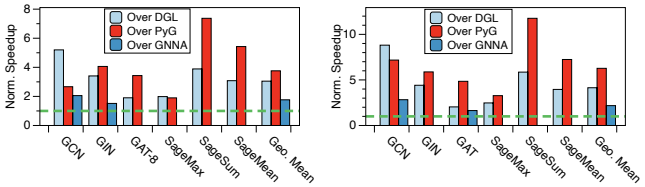


(a) V100.



(b) A100.

Figure 13: End-to-end inference time result on two GPUs. The first "S" in "SMax", "SSum" and "SMean" denotes "GraphSage". GNNAdvisor only supports GCN and GIN, so certain places have missing data.



(a) V100.

(b) A100.

Figure 14: Per-model speedup averaged on all datasets.

We further use NVIDIA profiling tool nvprof [36] to collect the GPU performance metrics to inspect the source of improvements in *uGrapher*. Fig. 16 shows the collected results for a typical operator on V100. *uGrapher* significantly improves the GPU's SM utilization, L2 cache hit rate, and occupancy.

7.3 Execution Strategy Analysis

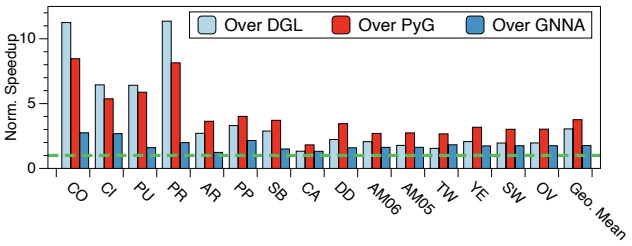
We also analyze the impact of various parallelization strategies on performance improvement. Tbl. 9 shows the optimal parallelization strategy of typical graph operators for different datasets on V100 and A100. The results show that different graph operators tend to choose different optimal strategies on different datasets and different hardware platforms.

Generally, the *thread-edge* strategy is used most often as the optimal parallelization strategy. For example, for the message creation graph operator for the first layer of GAT (GAT_L1_MsgC), *thread-edge* is the optimal strategy for all datasets and two GPUs. We also find that large graph datasets favor better locality over parallelism. For example, the first two layers of SageMax both choose the *thread-vertex* strategy with better locality but worse parallelism on OVACR-8H dataset. Comparing the two GPUs, we observe that they tend to choose similar optimal strategies but often different fine-grained parameters (i.e., V/E grouping and feature tiling). Meanwhile, another difference between the two GPUs is that the overall number of using vertex-mapping as the optimal strategy on V100 is more than that of A100. The reason is that V100 has fewer SMs so it favors less parallelism.

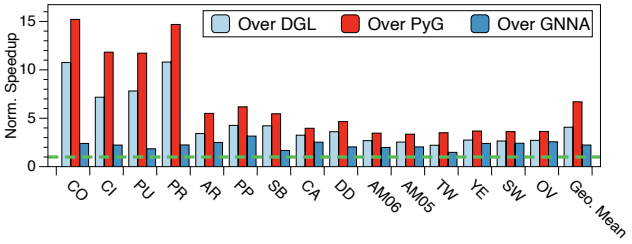
The above results confirm the necessity of the four basic parallelization strategies as each of them would be chosen as the optimal under different operators and datasets. We further study the performance impact of the two additional parameters, i.e., V/E grouping and feature tiling. Fig. 17 compares the normalized execution time of the basic strategies (without V/E grouping and feature tiling) for two operators against their optimal strategies on V100. The performance has a large gap from the optimal results when only using the basic strategy. As such, choosing these fine-grained control knobs is also critical for achieving the optimal performance for graph operators.

Table 9: Optimal strategies of different graph operators for different datasets on V100 and A100. The graph operators are denoted as model-layer-type. '*MsgC*' denotes message creation operator and '*Aggr*' denotes aggregation. The optimal strategy is denoted as (parallelization strategy)-(V/E grouping)-(feature tiling). '*TE*', '*WE*', '*TV*', '*WV*' are '*thread-edge*', '*warp-edge*', '*thread-vertex*', '*warp-vertex*' parallelization strategies, respectively. They are also highlighted with red, yellow, green, and gray backgrounds, respectively.

Dataset	GPU	GAT_L1_MsgC	GAT_L1_Aggr	GIN_L1_Aggr	GIN_L2_Aggr	GIN_L5_Aggr	SageMax_L1_Aggr	SageMax_L2_Aggr
CO	V100	TE_G1_T16	TE_G2_T32	WE_G4_T16	TE_G4_T32	TE_G4_T32	WE_G4_T16	TE_G1_T32
	A100	TE_G1_T32	WE_G2_T1	WE_G8_T64	WE_G2_T1	TE_G2_T32	WE_G4_T16	WE_G2_T1
CI	V100	TE_G1_T32	TE_G2_T32	WV_G2_T64	WE_G4_T16	TE_G4_T64	WV_G4_T64	WE_G2_T2
	A100	TE_G1_T4	TE_G2_T32	WV_G16_T64	WE_G2_T1	WE_G2_T1	WV_G4_T64	WE_G2_T8
PR	V100	TE_G2_T32	TE_G4_T32	WE_G8_T1	TE_G16_T64	WE_G8_T1	WV_G2_T8	TV_G1_T8
	A100	TE_G2_T64	WE_G8_T1	WE_G4_T8	TE_G8_T32	TE_G8_T32	WV_G1_T2	TV_G1_T64
AR	V100	TE_G4_T32	TE_G32_T32	TE_G16_T32	TE_G64_T32	TE_G64_T32	TE_G8_T32	TE_G8_T8
	A100	TE_G8_T64	WE_G64_T1	WE_G32_T1	TE_G64_T32	TE_G64_T32	TE_G8_T32	TE_G16_T16
SB	V100	TE_G4_T16	WE_G64_T1	WE_G64_T1	TE_G64_T32	TE_G64_T32	TE_G8_T32	TE_G8_T8
	A100	TE_G8_T16	WE_G64_T1	WE_G16_T1	WE_G64_T1	TE_G64_T32	TE_G8_T32	TE_G16_T64
DD	V100	TE_G4_T8	WE_G8_T1	TE_G4_T32	WV_G8_T32	WV_G8_T2	TV_G2_T32	TV_G1_T8
	A100	TE_G4_T8	TE_G16_T32	TE_G16_T32	TE_G64_T32	WE_G64_T8	TV_G4_T32	TV_G1_T8
TW	V100	TE_G4_T8	WE_G4_T1	WV_G16_T8	WV_G16_T8	WV_G16_T2	WV_G4_T8	TV_G2_T8
	A100	TE_G2_T4	TE_G16_T32	WV_G64_T8	WE_G8_T1	TE_G8_T32	WV_G32_T16	TV_G1_T4
YE	V100	TE_G4_T64	WV_G16_T64	TE_G4_T32	WV_G16_T16	WV_G16_T32	WV_G2_T1	TV_G1_T4
	A100	TE_G1_T4	TE_G2_T16	WE_G8_T1	TE_G8_T32	WV_G32_T4	TV_G4_T32	TV_G1_T4
OV	V100	TE_G4_T32	TE_G4_T32	TE_G4_T32	WV_G8_T4	WV_G8_T4	TV_G4_T32	TV_G1_T4
	A100	TE_G8_T8	WE_G16_T1	TE_G4_T32	WE_G16_T8	TE_G8_T32	TV_G4_T32	TV_G1_T4



(a) V100.



(b) A100.

Figure 15: Per-dataset speedup averaged on all models.

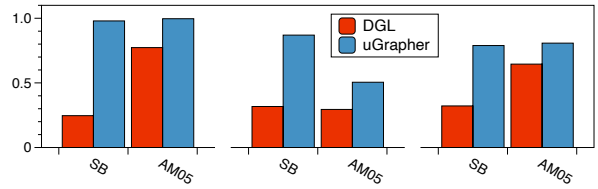


Figure 16: GPU performance metrics collected by nvprof for the second layer of SageMax. The left metric is SM utilization, the middle is L2 cache hit rate, and the right is achieved occupancy.

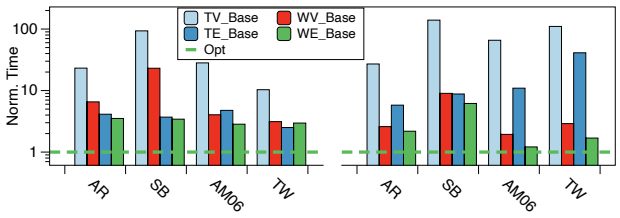


Figure 17: The normalized time of different base strategies compared to the optimal. The left figure shows the *message creation* operator for the first layer of GAT, and the right figure shows the *aggregation* operator for the first layer of GIN.

Fig. 18 shows the results obtained with varying V/E grouping and feature tiling parameters for the first layer of GIN with the TWITTER-Partial dataset on V100. It can be seen that the change of grouping and tiling parameters under different basic parallelization strategies will bring different effects on the performance. Thus, it is

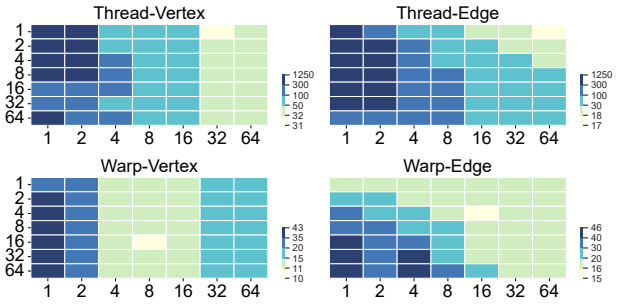


Figure 18: Execution time obtained by varying the grouping (y-axis) and tiling parameters (x-axis) in basic strategies for the first layer of GIN model with the TWITTER-Partial dataset on V100.

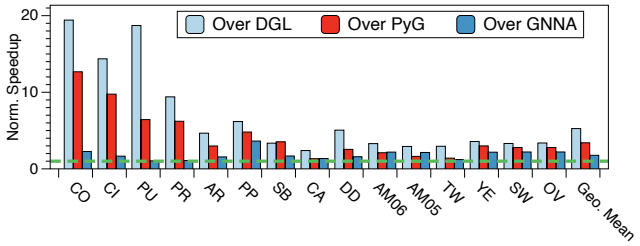


Figure 19: Comparison of the performance results obtained with node renumbering of GCN on V100.

necessary to fine-tune the V/E grouping and feature tiling parameters in different scenarios to achieve the optimal performance.

7.4 Additional Studies

Graph data preprocessing Data preprocessing, as a typical optimization method in graph algorithms, has also been explored in some research work on GNN [7, 33, 46, 49]. The *uGrapher*'s design does not restrict the optimizations of graph data preprocessing (e.g. node renumbering in GNNAdvisor), as they are orthogonal to our parallelization-centric optimizations. Fig. 19 shows the performance results compared to baselines with Rabbit node renumbering [2]. *uGrapher* can still obtain substantial performance improvement.

Overhead Analysis The scheduling overhead of *uGrapher* mainly comes from prediction model inference of LightGBM. However, the prediction only needs to be executed once before the GNN model inference, and the time is less than 0.2 ms for a single prediction, which has a negligible impact on performance.

8 RELATED WORK

GNN Frameworks. DGL [44] and PyG [11] are two popular GNN frameworks, which both use a message-passing programming interface based on DNN frameworks. Other frameworks including Roc [22], NeuGraph [31], and AliGraph [53] target large-scale distributed GNN processing. G^3 [30] focuses on using graph processing frameworks to train GNNs on GPUs. GNNAdvisor [46] presents a runtime system for accelerating GNNs on GPUs, but only supports GCN and GIN.

Existing works do not pay attention to the performance bottleneck of the graph operators, therefore cannot achieve optimal performance on GPUs.

Graph Processing on GPUs. Numerous graph processing systems [13, 24, 25, 28, 34, 40, 45] have been proposed to accelerate traditional graph algorithms on GPUs. Some of these research works have also tried to explore different parallelization strategies including vertex parallelism, edge parallelism, etc. There are also efforts to explore dynamic parallelization strategies through domain-specific language (DSL) [4, 5, 17, 50, 51].

However, GNNs differ from traditional graph algorithms in terms of graph operation characteristics and feature embedding dimension, for which the parallelization strategy space is beyond the capability of traditional graph processing systems.

Graph Kernel Optimization. There are also some works that try to explore the graph operators in GNNs to optimize GNNs. GE-SpMM [19] focuses on optimizing SPMM-like graph operators in GNNs. FeatGraph [18] extends TVM [6], and designs an execution mode on GPU for SPMM-like and SDDMM-like graph operators. However, these graph operator optimization efforts still take a manual optimization approach to optimize graph operators. FusedMM [38] provides an abstraction for SDDMM-SpMM kernels. However, this abstraction does not have the ability to decouple computation and scheduling, and thus cannot explore adaptive parallelization strategies.

Without providing an independent abstraction for graph operators and without exploring different parallelization strategies, there are limitations in the aspects of providing complete support for different operators and achieving optimal performance for these operators.

9 CONCLUSION

In this work, we propose *uGrapher*, a unified and high-performance interface that can support various graph operators in GNNs. Our design can decouple the computation and scheduling of graph operators by introducing a GNN-specific operator abstraction and exploring various parallelization strategies. Experimental results demonstrate that *uGrapher* is able to achieve an average speedup of 3.5× compared to existing SOTA frameworks, which can easily adopt our design owing to its simple and unified API.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China under Grant 2021ZD0110104, the National Natural Science Foundation of China (NSFC) grant (62222210, U21B2017, and 620722-97), Alibaba Group through Alibaba Innovative Research Program and Alibaba Research Intern Program. We would like to thank the anonymous reviewers for their constructive feedback for improving the work. We also thank Wencong Xiao, Qing Wang, Hao Lv, Zhihui Zhang, and Zihan Liu for their technical support and beneficial discussions. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2021. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys (CSUR)* 54, 9 (2021), 1–38.

[2] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 22–31.

[3] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charlie Nash, Victoria Langston, Chris Dyer, Nicolas Manfred Otto Heess, Daan Wierstra, Pushmeet Kohli, Matthew M. Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. 2018. Relational inductive biases, deep learning, and graph networks. *ArXiv* abs/1806.01261 (2018).

[4] Mario Luca Bernardi, Marta Cimitile, and Giuseppe Di Lucca. 2014. Design pattern detection using a DSL-driven graph matching approach. *Journal of Software: Evolution and Process* 26, 12 (2014), 1233–1266.

[5] Ajay Brahmakshatriya, Yunming Zhang, Changwan Hong, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2021. Compiling Graph Applications for GPU s with GraphIt. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 248–261.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An Automated {End-to-End} Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[7] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. 2021. Rubik: A hierarchical architecture for efficient graph neural network training. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 4 (2021), 936–949.

[8] Edward Choi, Cao Xiao, Walter Stewart, and Jimeng Sun. 2018. Mime: Multilevel medical embedding of electronic health records for predictive healthcare. *Advances in neural information processing systems* 31 (2018).

[9] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The world wide web conference*. 417–426.

[10] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Yihong Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (Eds.). ACM, 417–426. https://doi.org/10.1145/3308558.3313488

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

[12] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.

[13] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} Computation on Natural Graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[15] Pawan Harish, Vibhav Vineet, and PJ Narayanan. 2009. Large graph algorithms for massively multithreaded architectures. *International Institute of Information Technology Hyderabad, Tech. Rep. IIIT/TR/2009/74* (2009).

[16] Muhammad Amber Hassaan, Martin Burtscher, and Keshav Pingali. 2011. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. *Acm Sigplan Notices* 46, 8 (2011), 3–12.

[17] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for easy and efficient graph analysis. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. 349–362.

[18] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[19] Guyue Huang, Guohao Dai, Yu Wang, and Huazhong Yang. 2020. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.

[20] Kezhao Huang, Jidong Zhai, Zhen Zheng, Youngmin Yi, and Xipeng Shen. 2021. Understanding and bridging the gaps in current GNN performance optimizations. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 119–132.

[21] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2017. pybind11–Seamless operability between C++ 11 and Python. *URL: https://github. com/pybind/pybind11* (2017).

[22] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[23] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[24] Zuhair Khayyat, Karim Awara, Amani Alonazi, Hani Jamjoom, Dan Williams, and Panos Kalnis. 2013. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European conference on computer systems*. 169–182.

[25] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. 239–252.

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

[27] Sofia Ira Ktena, Sarah Parisot, Enzo Ferrante, Martin Rajchl, Matthew Lee, Ben Glocker, and Daniel Rueckert. 2018. Metric learning with spectral graph convolutions on brain connectivity networks. *NeuroImage* 169 (2018), 431–442.

[28] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. {GraphChi}:{Large-Scale} Graph Computation on Just a {PC}. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 31–46.

[29] Charles E Leiserson and Tao B Schardl. 2010. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*. 303–314.

[30] Husong Liu, Shengliang Lu, Xinyu Chen, and Bingsheng He. 2020. G3: when graph neural networks meet parallel graph processing systems on GPUs. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2813–2816.

[31] 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.

[32] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU graph traversal. *ACM Sigplan Notices* 47, 8 (2012), 117–128.

[33] Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, and Sachin S Sapatnekar. 2022. GNNIE: GNN inference engine with load-balancing and graph-specific caching. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 565–570.

[34] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming irregular graphs for GPU-friendly graph processing. *ACM SIGPLAN Notices* 53, 2 (2018), 622–636.

[35] NVIDIA. 2021. NVIDIA A100 Tensor Core GPU Architecture. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[36] NVIDIA. 2021. Profiler User's Guide. https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[37] Tesla NVIDIA. 2017. Nvidia tesla v100 gpu architecture.

[38] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. 2021. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 256–266.

[39] Ryan Rossi and Nesreen Ahmed. 2015. The network data repository with interactive graph analytics and visualization. In *Twenty-ninth AAAI conference on artificial intelligence*.

[40] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 472–488.

[41] Julian Shun and Guy E Blelloch. 2013. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

[42] Dave Steinkraus, Ian Buck, and PY Simard. 2005. Using GPUs for machine learning algorithms. In *Eighth International Conference on Document Analysis and Recognition (ICDAR'05)*. IEEE, 1115–1120.

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

[44] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, et al. 2019. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315* (2019).

[45] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. 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*. 1–12.

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

[47] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

[48] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).

[49] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. 2020. Hardware acceleration of large scale gcn inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 61–68.

[50] Yunming Zhang, Ajay Brahmakshatriya, Xinyi Chen, Laxman Dhulipala, Shoaib Kamil, Saman Amarasinghe, and Julian Shun. 2020. Optimizing Ordered Graph Algorithms with GraphIt. In *International Symposium on Code Generation and Optimization*.

[51] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–30.

[52] Zhihui Zhang, Jingwen Leng, Lingxiao Ma, Youshan Miao, Chao Li, and Minyi Guo. 2020. Architectural implications of graph neural networks. *IEEE Computer architecture letters* 19, 1 (2020), 59–62.

[53] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. Aligraph: a comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730* (2019).

Received 2022-07-07; accepted 2022-09-22