GRAPHILER: OPTIMIZING GRAPH NEURAL NETWORKS WITH MESSAGE PASSING DATA FLOW GRAPH

Zhiqiang Xie [1 2] **Minjie Wang** [2] **Zihao Ye** [2 3] **Zheng Zhang** [2] **Rui Fan** [1]

ABSTRACT

Graph neural networks (GNNs) are a new class of powerful machine learning models, but easy programming and efficient computing is often at odds. Current GNN 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). While built-in primitives offer high performance, they are limited in expressiveness; UDFs are flexible, but often have low performance and use excessive memory. In this paper, we propose Graphiler, a compiler stack for GNNs which achieves high performance while offering the flexibility of the UDF programming interface. At the core of Graphiler is a novel abstraction called *Message Passing Data Flow Graph* (MP-DFG), which enables optimizations that substantially reduce computational redundancy and memory footprint, and optimizes both homogeneous and heterogeneous GNNs under a unified framework. Experiments show Graphiler can accelerate UDF GNNs by up to two orders of magnitude, and achieve performance close to or superior to expert implementations, and do so with substantial memory savings.

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 graph propagation and iteratively update node and edge features based on features from neighbors. 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 (Baidu, 2019) and Graph Nets (Battaglia et al., 2018).

However, these frameworks face a challenging trade-off between performance and flexibility. In particular, to enable users to easily construct novel GNN models, several existing frameworks allow the creation of user-defined functions (UDFs) using standard tensor operations which users are familiar with. But while this design permits a high degree of flexibility, straightforward implementations of UDFs are often sub-optimal, and suffer from redundant computations and excessive memory consumption (Huang et al., 2021). Additionally, complex UDFs may include a large number of small tensor operations, leading to excessive function call overhead.

To achieve higher performance, these frameworks also provide a limited set of built-in primitives based on highly efficient sparse matrix operations. These primitives are powerful if effectively leveraged, but require in-depth knowledge of the framework as well as the model at hand, and are tedious to optimize when used in combination.

The performance & flexibility trade-off is even more pronounced for complex models such as heterogeneous graph neural networks (hetero-GNNs). Heterogeneous graphs contain typed nodes and edges, and are prevalent in a number of real-world settings such as knowledge graphs, E-commerce and social networks. As such, hetero-GNNs form an active area of research (Schlichtkrull et al., 2018; Hu et al., 2020d; Wang et al., 2019c; Zhang et al., 2019). However, none of the built-in primitives in current GNN frameworks support hetero-GNN operations, and users must resort to using much less efficient UDF-based implementations. Furthermore, optimizing hetero-GNNs for computational and memory efficiency is even more challenging, as users need to carefully deal with the complex data access patterns associated with heterogeneous graphs. As an example of this difficulty, we show in §5.2.2 that even experienced system

[1]ShanghaiTech University [2]Amazon Web Services [3]University of Washington. Correspondence to: Zhiqiang Xie <xiezhq@shanghaitech.edu.cn>, Minjie Wang <minjiw@amazon.com>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022.

developers often produce sub-optimal hetero-GNN implementations.

In this paper, we propose *Graphiler*, a GNN compiler which automatically compiles GNNs defined using UDFs into efficient execution plans, which allows creating high performance models while retaining the expressiveness of the UDF interface. At the core of Graphiler is a novel abstraction called *message passing data flow graph* (MP-DFG), which enriches traditional DFGs with graph-based message passing semantics. MP-DFG uses semantic information from message passing UDF signatures to deduce *data movement* patterns in the computation, then applies a number of powerful optimizations based on these patterns. For example, MP-DFG allows detecting opportunities for broadcast reordering or fusion, which can significantly reduce redundant computations and memory accesses. Our abstraction also generalizes naturally to hetero-GNNs, allowing them to share data structures and kernels originally designed for homogeneous GNNs, and enabling the automated discovery of previously unexplored optimizations.

We evaluate the effectiveness of Graphiler by using it to compile a number of state-of-the-art GNN and hetero-GNN models written using UDFs. The results show that Graphiler can accelerate the models by up to two orders of magnitude, and achieve performance on par with or sometimes superior to implementations written and tuned by expert users using built-in primitives. On hetero-GNNs, Graphiler can outperform state-of-the-art implementations by up to 7.9×.

2 BACKGROUND

In this section, we provide an overview of the message passing paradigm for GNNs and hetero-GNNs, and introduce two running examples we will refer to in the remainder of the paper.

2.1 Message Passing Paradigm for GNNs

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with node set $\mathcal{V}$ and edge set $\mathcal{E}$. Each node $u \in \mathcal{V}$ is associated with a feature vector $x_u \in \mathbb{R}^{f_v}$, and each edge $(u, e, v) \in \mathcal{E}$[1] is associated with a feature vector $w_e \in \mathbb{R}^{f_e}$, where f_v and f_e are the feature dimensions. 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 its two endpoints.

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.

The preceding model is defined for homogeneous graphs, but can be extended naturally to heterogeneous graphs with multiple node and edge types. An example of a heterogeneous graph is for modeling a citation network. Here we can define two types of nodes, authors and papers, as well as two types of edges, the first between an author node and a paper node representing authorship, and the second between a pair of paper nodes representing a citation. Formally, we define a heterogeneous graph as $\mathcal{HG} = (\mathcal{V}, \mathcal{E}, \mathcal{A}, \mathcal{R})$. $\mathcal{V}$ and $\mathcal{E}$ again represent nodes and edges, and $\mathcal{A}$ and $\mathcal{R}$ are finite sets representing node and edge types, resp. We define functions $\tau : \mathcal{V} \to \mathcal{A}$ and $\varphi : \mathcal{E} \to \mathcal{R}$ mapping each node or edge to its type. In addition, each type $\kappa \in \mathcal{A} \cup \mathcal{R}$ is associated with a weight tensor w_κ. The message passing paradigm for hetero-GNNs can then be defined as follows:

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

2.2 Running Examples: GAT and HGT

We now describe the widely used *graph attention network* (GAT) (Velickovic et al., 2018) GNN, and its hetero-GNN counterpart *heterogeneous graph transformer* (HGT) (Hu et al., 2020d), in order to help illustrate the design and capabilities of Graphiler. The main difference between these two models lies in the message creation stage. Given an edge $j \to i$, the node and edge messages for the edge in GAT are given by:

$$m_{j\to i} = z_j = W h_j, \quad (1)$$

$$e_{j\to i} = LeakyReLU(W^{ATT}(z_i || z_j)), \quad (2)$$

HGT was inspired by the design of *Transformers* (Vaswani et al., 2017), and maps the features of each node i into Query, Key and Value vectors using matrices $W^K_{\tau(i)}$, $W^Q_{\tau(i)}$ and $W^V_{\tau(i)}$ based on the node's type $\tau(i)$ (Equation (3)). To incorporate relational information, it further introduces two weight matrices $W^{ATT}_{\varphi(j\to i)}$ and $W^{MSG}_{\varphi(j\to i)}$ associated edge $j \to i$'s type $\varphi(j \to i)$ (Equations (4) and (5)):

$$K_j, Q_i, V_j = W^K_{\tau(j)} h_j, W^Q_{\tau(i)} h_i, W^V_{\tau(j)} h_j, \quad (3)$$

$$m_{j\to i} = W^{MSG}_{\varphi(j\to i)} V_j, \quad (4)$$

$$e_{j\to i} = K_j W^{ATT}_{\varphi(j\to i)} Q_i^T, \quad (5)$$

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

GAT and HGT share the same message aggregation function, where each node receives messages from incoming edges and produces an aggregated result r:

$$\alpha_{j\to i} = \frac{\exp(e_{j\to i})}{\sum_{k\in\mathcal{N}(i)}\exp(e_{k\to i})}, \quad (6)$$

$$r_i = \sum_{j\in\mathcal{N}(i)} \alpha_{j\to i} m_{j\to i}, \quad (7)$$

In the feature update stage, the features of each node are updated by an activation function σ; for ease of exposition, we omit the complex update performed in HGT.

$$h_i^{new} = \sigma(r_i) \quad (8)$$

3 Performance Problems of Current GNN Systems

A key to the success of GNNs is the flexible choice of the message function ϕ, aggregation function ρ and update function ψ in the message passing paradigm. Current GNN systems allow users to express these functions natively in Python using tensor operators and then invoke them using system provided APIs, a programming paradigm which we call *user-defined functions* (UDFs). Figure 1(a) shows a simplified implementation of GAT and HGT in DGL's UDF interface, where each line of code is associated with an equation from §2.2. Other frameworks such as PyG adopt a similar design. While UDFs is flexible and user-friendly, they can lead to significantly degraded performance. This forces these frameworks to provide efficient but much less expressive primitive operators.

3.1 Redundancy in message creation

The message creation UDF `message_hgt` (lines 8-17) provides users an edge-centric view. Users can access the previously computed representations of an edge's source and destination nodes, the edge itself, and the parameter matrices associated with their types via methods from the `edges` argument. All these methods internally gather data from the corresponding storage tensors and pack them into a message tensor. For example, in line 10, `edges.src['h']` looks up the source node representation for each edge from the node tensor storing all the node representations. Likewise, `edges.srctype['Wk']` gathers the weight matrices based on the source node type of each edge, as illustrated in Figure 1(b). Because all the message tensors have size $|\mathcal{E}|$ in the leading dimension, the subsequent `batch_mm` can perform many matrix multiplications in a batch.

This approach for gathering data leads to a large amount of redundancy. We show this using an example heterogeneous citation network with two types of nodes (author and paper) and two types of edges (authorship and citation). The authorship edges connect author nodes to paper nodes, and the citation edges connect two paper nodes. Figure 1(b) shows the messages produced from different source data using different colors, and illustrates the data redundancy incurred. The redundancy can be severe in real-world graphs, because many nodes may share the same type (*i.e.* $|\mathcal{A}| << |\mathcal{V}|$) and nodes may have high degree (*i.e.* $|\mathcal{V}| << |\mathcal{E}|$).

3.2 Fragmented computation in message aggregation

The message aggregation UDF `aggregate_func` (Figure 1(a), lines 19-24) provides users a node-centric view. Users can access the messages computed by the message creation UDF as well as the node representations using the `node` argument. Since nodes may have different degrees, they can receive different numbers of messages, and this makes batching during aggregation difficult. To deal with this, DGL internally shards the message tensor into multiple pieces, with each piece grouping together nodes receiving equal number of messages. This "bucketing" strategy makes it possible to apply arbitrary tensor operations inside an aggregation UDF. The system then invokes `aggregate_func` on each message tensor shard to obtain the aggregated messages of all the nodes. In a hetero-GNN, if a user wishes to aggregate messages differently for different node types, the system further shards the message tensor based on destination node type and loops over them. However, sharding message tensors fragments the computation, and leads to performance issues such as low GPU utilization, large kernel launch overhead, etc.

3.3 Trading expressiveness for efficiency

To circumvent the previous performance issues, all current GNN frameworks choose to limit their expressiveness to various degrees. For instance, PyG forbids aggregation UDFs and thus avoids the problem of fragmented computation. DGL does, but recommends that users express their models using a set of built-in message creation and aggregation primitives. For example, if an aggregation UDF simply returns `{'r': sum(nodes.mailbox['m'])}`, it can be replaced by `dgl.sum('m', 'r')`, which is executed using an efficient sparse matrix kernel. In addition, if both the message creation and aggregation functions are chosen from the built-in set, DGL fuses the two stages and completely bypasses any message creation. This can also be achieved in PyG using the `message_and_aggregate` function. However, DGL's approach requires a large set of built-in primitives to achieve good model coverage, and the amount of engineering effort needed is often infeasible in practice. For example, the HGT code in Figure 1(a) cannot be constructed using the current DGL primitives.

Moreover, even when it is feasible to program a model us-

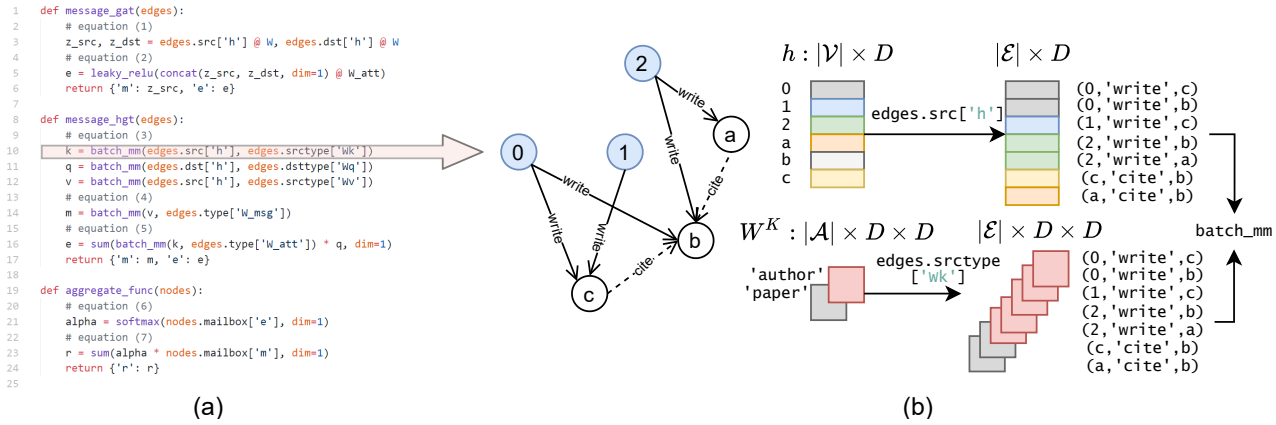

Figure 1: (a) GAT and HGT implementations in DGL UDFs. (b) A Toy heterogeneous graph and an illustration of the message gathering process for code line 10.

```
def gat_dgl_primitives(graph, h):
    # equation (1)
    z_src = z_dst = h @ W
    # equation (2)
    el = z_src @ W_att_l
    er = z_dst @ W_att_r
    graph.srcdata.update({'m': z_src, 'el': el})
    graph.dstdata.update({'er': er})
    graph.apply_edges(dgl.u_add_v('el', 'er', 'e'))
    e = leaky_relu(graph.edata.pop('e'))
    # equation (6)
    e_max = dgl.copy_e_max(graph, e)
    e = exp(dgl.e_sub_v(graph, e, e_max))
    e_sum = dgl.copy_e_sum(graph, e)
    graph.edata['alpha'] = dgl.e_div_v(graph, e, e_sum)
    # equation (7)
    graph.update_all(dgl.u_mul_e('m', 'alpha', 'm'), dgl.sum('m', 'r'))
    return graph.dstdata['r']
```

Figure 2: GAT implementation using DGL primitives.

ing existing primitives, it is challenging for non-experts to understand and make good use of them. Figure 2 shows how DGL developers construct the GAT model without UDFs to achieve better performance. By comparing this with Figure 1(a), we can observe that users have to carefully manage data associated with nodes and edges in the same scope, and additionally decompose familiar tensor operators (e.g. `concat` and `softmax`) into efficient but non-intuitive graph primitives.

3.4 Summary

We observe that UDFs offer a high degree of expressiveness and work well as a user-facing interface. At the same time, built-in primitives are a good low-level interface targeting system efficiency. Therefore, we have built Graphiler as a compiler stack to build a bridge so as to offer the benefits of both. A key insight in Graphiler's design is that the most computationally demanding parts of a GNN computation, namely message gathering and aggregation, can be associated with changes in data storage type, which we term *data residency*. For instance, `edges.src['h']` converts representations which conceptually reside at nodes to messages which reside on edges, while `edges.srctype['Wk']` converts weights which reside with node types to messages on edges. Similarly, message aggregation converts messages on edges to new node representations. Therefore, the main algorithmic problems Graphiler focuses on are detecting data residency changes and deducing optimizations based on the changes.

4 Design of Graphiler

Graphiler centers around a novel intermediate representation (IR) that we call *Message Passing Data Flow Graph* (MP-DFG), which augments a classic DFG with residency information about tensors and the movement pattern induced by operators. In this section, we first define *data residency* and *data movement* in §4.1, then describe how Graphiler builds an MP-DFG for a GNN program, in §4.2 which enables two effective GNN optimizations in §4.3.

4.1 Message Passing Data Flow Graph

A DFG is an acyclic graph with nodes and edges representing operators and the dependencies between them, respectively. Existing DNN compilers, including XLA (Google, 2017), NNVM (DMLC, 2017) and JAX (Bradbury et al., 2018), have successfully leveraged this abstraction. Annotations such as data types and the shapes of intermediate tensors to enables optimization such as kernel fusion (Chen et al., 2018b), subgraph substitution (Jia et al., 2019), etc.

The key insight of Graphiler is to extend DFG with *message passing semantics* that captures changes in data storage type as the computation moves along, summarized in Figure 3:

Data residency is an annotation that associates a variable with the element of a graph, and there are five of them: node data $\mathcal{D}_{\mathcal{V}}$, edge data $\mathcal{D}_{\mathcal{E}}$, node type data $\mathcal{D}_{\mathcal{A}}$, edge type

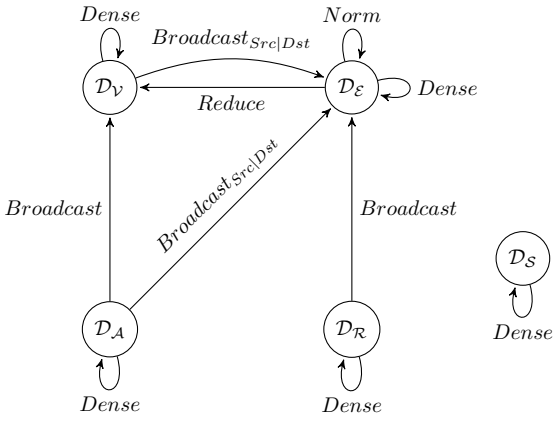

Figure 3: Transition graph for data residency in an MP-DFG. Edges represent data movement; edge direction indicates the residency of input and output data.

data $\mathcal{D}_\mathcal{R}$ and shared data $\mathcal{D}_\mathcal{S}$. The tensors of the first four residency types have $|\mathcal{V}|$, $|\mathcal{E}|$, $|\mathcal{A}|$ and $|\mathcal{R}|$ as the size of their leading dimension, while the tensors of shared residency type (e.g., GNN model weight matrices) can have arbitrary shapes since they are not directly related to the graph.

Data movement is associated with each operator, indicating how the operator transforms the data residency of its input. Under the message passing paradigm, we categorize each operator into one of four classes:

- *Broadcast* operators convert data from residency type $\mathcal{D}_x$ to another residency type $\mathcal{D}_y$ by index look-up. For example, `edges.srctype['Wk']` (Figure 1, line 10) converts node type weights into an edge data tensor by looking up the weight matrices of the source node type of each edge. Because broadcasting node data or node type data into edge data can be based on either the source or destination nodes of edges, we further distinguish them with $Broadcast_{Src}$ and $Broadcast_{Dst}$. Broadcast operators typically induce data duplication if $|\mathcal{D}_y| >> |\mathcal{D}_x|$.
- $Reduce$ operators are the reverse, as for example in line 23 in Figure 1, where the `sum` operator transforms the input data of residency $\mathcal{D}_\mathcal{E}$ to output data of $\mathcal{D}_\mathcal{V}$ by aggregating edge data to a node.
- $Norm$ operators work on edge data, combining two steps in one by first aggregating edge data belonging to the same destination node and then broadcasting the aggregated result back to the edges. Although the data residency of the output remains the same as that of the input, norm operators still perform data movements between nodes and edges internally.
- $Dense$ operators are independent of the graph structure and generate no data movement. In other words, for input data with residency $\mathcal{D}_\mathcal{V}, \mathcal{D}_\mathcal{E}, \mathcal{D}_\mathcal{A}$, or $\mathcal{D}_\mathcal{R}$, the operator will not compute across the first dimension. The data movement of unary dense operators is illustrated in Figure 3. General dense operators take N inputs $\{x_1, ..., x_N\}$, where each x_i has data residency $\mathcal{D}_i \in \{\mathcal{D}_\mathcal{S}, \alpha\}$, for $\alpha \in \{\mathcal{D}_\mathcal{V}, \mathcal{D}_\mathcal{E}, \mathcal{D}_\mathcal{A}, \mathcal{D}_\mathcal{R}\}$. It produces output data with residency $\mathcal{D}_\mathcal{S}$ if all $\mathcal{D}_i = \mathcal{D}_\mathcal{S}$, and otherwise is α.

Table 1: Inference rules of data residency.

Op/API	Loc of Call	Data Movement
`edges.data`	Msg Creation	$Fetch\ data : \mathcal{D}_\mathcal{E}$
`edges.src` `edges.dst`	Msg Creation Msg Creation	$Broadcast_{Src} : \mathcal{D}_\mathcal{V} \rightarrow \mathcal{D}_\mathcal{E}$ $Broadcast_{Dst} : \mathcal{D}_\mathcal{V} \rightarrow \mathcal{D}_\mathcal{E}$
`edges.srctype` `edges.dsttype`	Msg Creation Msg Creation	$Broadcast_{Src} : \mathcal{D}_\mathcal{A} \rightarrow \mathcal{D}_\mathcal{E}$ $Broadcast_{Dst} : \mathcal{D}_\mathcal{A} \rightarrow \mathcal{D}_\mathcal{E}$
`edges.type`	Msg Creation	$Broadcast : \mathcal{D}_\mathcal{R} \rightarrow \mathcal{D}_\mathcal{E}$
`nodes.mailbox`	Msg Aggr.	$Fetch\ data : \mathcal{D}_\mathcal{E}$
`nodes.data`	Msg Aggr. , Feat. Update	$Fetch\ data : \mathcal{D}_\mathcal{V}$
`nodes.type`	Msg Aggr. , Feat. Update	$Broadcast : \mathcal{D}_\mathcal{A} \rightarrow \mathcal{D}_\mathcal{V}$
`sum`, `max`, ... with `dim=1`	Msg Aggr.	$Reduce : \mathcal{D}_\mathcal{E} \rightarrow \mathcal{D}_\mathcal{V}$
`softmax`, ... with `dim=1`	Msg Aggr.	$Norm : \mathcal{D}_\mathcal{E} \rightarrow \mathcal{D}_\mathcal{E}$
Other tensor ops	Anywhere	$Dense$: §4.1

4.2 MP-DFG Builder

Graphiler first extracts a standard DFG from each UDF using the tools such as TorchScript available for PyTorch (Paszke et al., 2019). Graphiler first determines the data residencies for the root variables. This information is explicitly or implicitly specified through interfaces provided by existing GNN frameworks (e.g., `edges.src` for broadcasting node data to edge data, `edges.srctype` for broadcasting node type data to edge data, etc.). Graphiler detects these operations and annotates the input and output data residencies as well as their data movement types.

Graphiler then infers the data movements of operators and the residencies of subsequent output variables of the DFG in a topological order. In each step of this process, Graphiler uses rules based on operator description, location of invocation (i.e., stage of message passing) and input data residency to perform inference. These inference rules are listed in Table 1. Inference can fail if a user invokes either operators or APIs on data with invalid residency types, or invokes customized operators whose data movements are unknown. In these cases, Graphiler will abort and fall back. Graphiler also supports expanding the rule set for customized operators and preparing these rules is onetime and lightweight.

Once inference is completed, all variables and operators will have been populated with their respective residency and movement types, and an MP-DFG is created. Figure 4 shows the MP-DFG built by Graphiler from the GAT code sample. Using the MP-DFG formalism, Graphiler is able

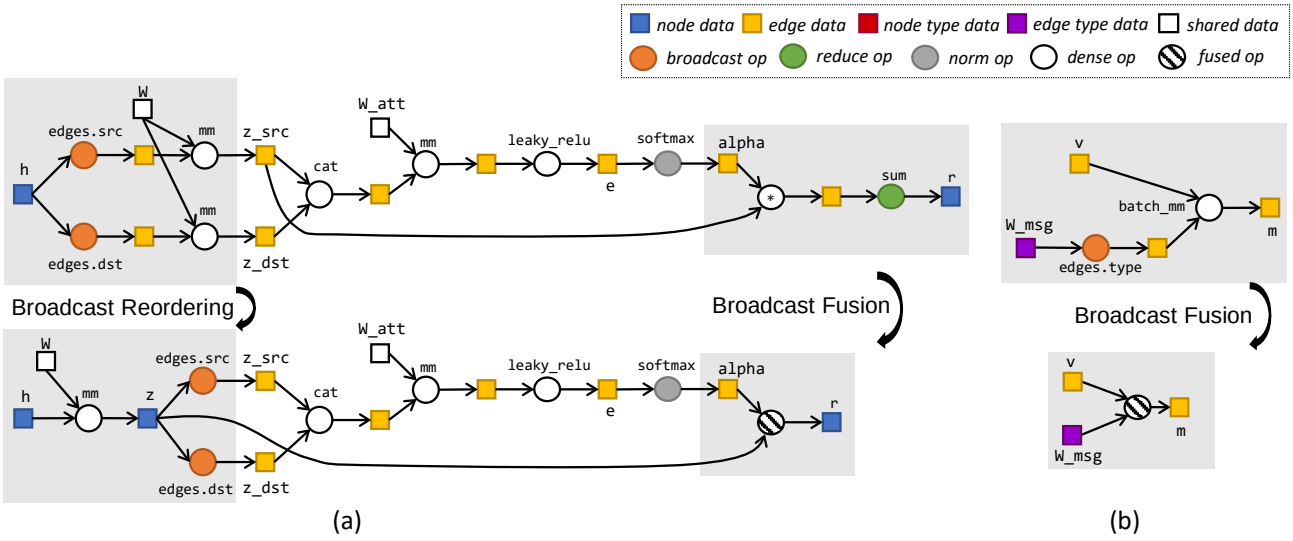

Figure 4: (a) Transformation on the MP-DFG of GAT. (b) Transformation on part of the MP-DFG of HGT

to effectively model irregular data accesses in aggregation UDFs containing $Reduce$ and $Norm$ operators, and automatically replace regular tensor operators in UDFs with efficient primitives to remove the fragmented computations described in §3.2.

4.3 MP-DFG Optimizer

Graphiler adopts a *pattern matching* approach to optimize MP-DFGs. It iteratively traverses an MP-DFG to match subgraphs with predefined patterns and replace them with optimized ones. By capturing the semantics of message passing computations, MP-DFGs enable several optimizations which can substantially accelerate GNN models. Here we discuss two important optimization patterns.

4.3.1 Broadcast Reordering

As discussed in §3.1, broadcast operations can introduce redundant computations. For instance, consider the two matrix multiplications at line 3 in the GAT example, where node data gets scattered first and then multiplied. Suppose the node feature size is d_1 and the weight matrix `w` has shape $d_1 \times d_2$, this multiplication is $O(|\mathcal{E}| + |\mathcal{E}|d_1d_2)$, where the first term is the cost of broadcasting node features onto edges while the second term is for the matrix multiplication. The redundancy can be eliminated by changing the order of the computation sequence as shown in Figure 4 (a), with the same result but drop the complexity to $O(|\mathcal{V}|d_1d_2 + |\mathcal{E}|)$ (since $|\mathcal{E}| >> |\mathcal{V}|$ for most graphs): computing the matrix multiplication first then broadcasting the outcome to edges.

Formally, we define the following sub-graph substitution rule:
Source Pattern: $y = f(g(x))$, where f and g are $Dense$ and $Broadcast$ operators respectively[2].
Substitute Pattern: $y = g(f(x))$

Note that the substitution rule is applicable to both homogeneous and heterogeneous GNNs. For heterogeneous GNNs, it can appear when node and edge type data are broadcasted to edges. For example, the Simple-HGN model (Lv et al., 2021) computes a message of an edge by multiplying the edge type embedding with a weight matrix. This can be captured by matching x as an edge type data, g as an operator of movement type $Broadcast : \mathcal{D}_{\mathcal{R}} \rightarrow \mathcal{D}_{\mathcal{E}}$ and f as the matrix multiplication.

4.3.2 Broadcast Fusion

In DNNs, the tensor produced by one operator is often immediately consumed by a subsequent operator. Fusing these two operators into one can greatly reduce memory bandwidth use and kernel launch overhead. This optimization applies to GNNs with even greater benefit because of the prevalence of broadcast operations that produces large amount of data that are soon consumed by operators that follow. Fusing can often reduce the amount of memory access and the memory footprint by $O(|E|)$. We separate this optimization into two computation patterns:

Broadcast-compute. An example can be found in Figure 4(b), which corresponds to the line 14 of HGT code in Figure 1(a). The intermediate edge data produced by the broadcast operator `edge.type` soon gets consumed by the following `batch_mm`. Suppose `W_msg` has shape $(|R| \times d_1 \times d_2)$. By fusing the broadcast operator with

[2]For simplicity, we omit the arguments of shared data residency for all the $Dense$ operators in the pattern description.

`batch_mm`, the fused kernel is able to directly read from `W_msg`, eliminating an intermediate memory buffer of size $|E|d_1d_2$ as well as the memory traffic to access it. We can formulate this sub-graph substitution rule as follows:
Source Pattern: $z = f(g_1(x), y)$ or $z = f(x, g_2(y))$ or $z = f(g_1(x), g_2(y))$, where f is an $Dense$ operator, g_1 and g_2 are $Broadcast$ operators.
Substitute Pattern: $z = FusedOp(x, y)$

Broadcast-compute-reduce. The pattern is common in many message passing GNNs. GAT, for example, first looks up source node features onto edges, scales it by the edge attention value and aggregates them into new node features. Expert developers usually rewrite it with framework provided primitives that fuse the three steps into one to avoid instantiating the intermediate data of size $O(E)$ (Figure 2, line 17). Graphiler automatically does so with the following rule:
Source Pattern: $z = \rho(f(g_1(x), g_2(y)))$ or $z = \rho(f(x, g_2(y)))$ or $z = \rho(f(g_1(x), y))$, where ρ and f are $Reduce$ and $Dense$ operators respectively; g_1 and g_2 are $Broadcast$ operators.
Substitute Pattern: $z = FusedOp(x, y)$

While recent works (Wang et al., 2019b; Huang et al., 2021; Wu et al., 2021) have explored these two types of fusion patterns for homogeneous GNNs, they failed to identify the fusion opportunities in hetero-GNNs. With the help of MP-DFG, we observe that all explored fusion patterns in prior works are special cases of our rules, which allows Graphiler to seamlessly apply broadcast fusion to both homogeneous and heterogeneous GNNs.

Fused Operator Implementation. Graphiler leverages a rich set of primitives provided by DGL to serve as kernels of fused operators. Therefore, Graphiler adopts sparse matrix representations used in DGL as well, namely CSR and COO. As a result, load balancing is also largely handled by these sparse computation kernels. In addition to that, we only manually implement a few commonly used kernels, e.g., segmented matrix multiplication, for better coverage. We leave efficient sparse matrix computation kernel generation as a future work.

Besides the optimizations tailored to the message passing paradigm of GNNs, traditional DFG optimizations such as dead-code elimination, common sub-expression elimination, pattern matching and substitution, etc. are applicable to MP-DFG as well.

5 Evaluation

5.1 Experimental Setup and Methodology

Benchmark models and datasets. Our benchmarks are based on a broad range of GNN models designed for node classification tasks. These include the widely used Graph Convolutional Network (GCN) (Kipf & Welling, 2017), the attention-based model GAT (Velickovic et al., 2018), and an improved version of GAT called Constrained GAT (C-GAT) (Wang et al., 2019a). All three models are standard GNNs consisting of a single node and edge type. For hetero-GNNs, we evaluated the popular R-GCN (Schlichtkrull et al., 2018) model, and the transformer-based HGT (Hu et al., 2020d) model, which achieves state-of-the-art accuracy. For all models, we follow standard practices and use two layers with 64 dimensions in the hidden layer.

We compare Graphiler with the following four baselines.

- *DGL-UDF*: A set of non-expert implementations written by DGL users using the UDF interface. These were collected largely from the DGL forum.
- *DGL-Primitives* and *PyG-Primitives*: A set of expert implementations carefully engineered by DGL and PyG framework developers using primitive operators similar to the ones shown in Figure 2. These were collected from the official DGL and PyG repositories.
- *Seastar* (Wu et al., 2021) implementations: Seastar is a recent GNN framework with a similar pipeline to Graphiler, but based on a vertex-centric programming interface and a DFG-based IR.

We validated our experimental results on ten graphs containing thousands to millions of nodes and drawn from a variety of domains, as shown in Table 2. For homogeneous graphs, we considered Pubmed (Sen et al., 2008), ogbn-arxiv (Hu et al., 2020a), PPI (Zitnik & Leskovec, 2017), and Reddit (Hamilton et al., 2017). For heterogeneous graphs, we considered MUTAG, BGS and AM from the Semantic Web Dataset (Ristoski et al., 2016), and the ogbn-biokg dataset from the Open Graph Benchmark (Hu et al., 2020b).

Machine environment. We conducted our experiments on an AWS p3.2xlarge instance equipped with an NVIDIA Tesla V100 GPU (16GB version) and Intel(R) Xeon(R) E5-2686 v4@2.30GHz CPUs. The software environment included Ubuntu 20.04 and CUDA 11.1.

We prototyped Graphiler using DGL v0.6.1 with a PyTorch 1.8.2 backend. Our baselines consist of expert and non-expert implementations in DGL v0.6.1 and PyTorch Geometric (PyG) 2.0.1. All reported performance numbers are averages over 1,000 runs and exhibited low variance. The correctness of programs compiled using Graphiler was verified by comparing their outputs with those from the original UDF implementations. Each program was only compiled once for different input graphs and the compilation only took seconds to complete.

Table 2: Graph datasets.

Dataset	PubMed	PPI	ogbn-arxiv	Reddit	MUTAG	BGS	ogbn-biokg	AM
#vType	1	1	1	1	5	27	5	7
#eType	1	1	1	1	46	96	51	96
#Vertex	19,717	56,944	169,343	232,965	27,163	94,806	93,773	1,885,136
#Edge	88,651	1,644,208	1,166,243	114,615,892	148,100	672,884	4,762,678	5,668,682
#Feature	500	50	128	602	-	-	-	-
Density	0.0228	0.0507	0.0041	0.2112	0.0201	0.0075	0.0542	0.00015

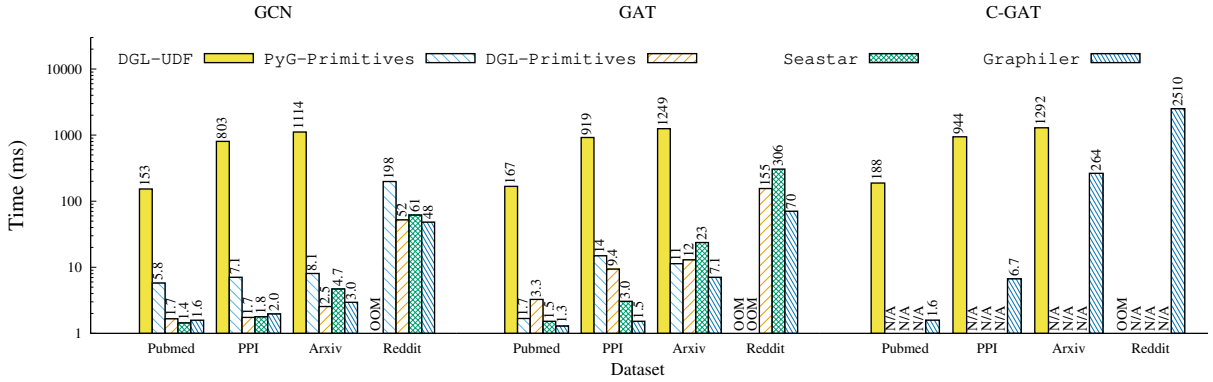

Figure 5: Inference time for homogeneous GNN models on different graph datasets.

We focus our evaluation on a single GPU setting. This was done because many important GNN applications today fit in a single GPU. In addition, Graphiler focuses on computation graph level optimizations, which is orthogonal to the cross device communication based optimizations considered in multi-GPU GNN systems. Finally, we believe our reported speedups will largely carry over to a distributed setting. In particular, distributed GNN computations often break a large graph into smaller subgraphs and compute on each subgraph using individual GPUs. Thus, the performance of these systems largely depends on the performance of the single GPU computations.

5.2 End-to-end Comparison

5.2.1 Homogeneous GNNs

We first demonstrate the overall efficiency of Graphiler for homogeneous GNNs. Note that none of DGL, PyG and Seastar can express C-GAT using framework provided primitives due to its edge pair-wise normalization. By contrast, Graphiler can successfully compile and accelerate the UDF implementation.

As shown in Figure 5, Graphiler significantly outperforms DGL-UDF on all the models. For example, on the PPI dataset it achieves a $407\times$, $604\times$ and $145\times$ speedup for GCN, GAT and C-GAT, respectively. Our speedup also scales with graph size because of the skewed degree distribution which severely fragments the computation for message aggregation in DGL-UDF. An exception is the C-GAT model, where the relative speedup drops on a larger graph due to the load imbalance problem with the underlying kernel for the $Norm$-type operator. We leave the kernel optimization as a future work.

Compared with DGL-Primitives, PyG-Primitives and Seastar, Graphiler-compiled programs demonstrate competitive performance in all cases and are faster in many cases. For GCN, Graphiler achieves almost the same speed as DGL-primitive, slightly faster than Seastar and $3.4\times$ faster than PyG-primitive on average over all datasets. For GAT, the average speedup over DGL-primitive and Seastar are $3.1\times$ and $2.7\times$ respectively, which is mainly due to the implementation of the underlying fused kernels (e.g., edge softmax) implementation.

We also measured GPU memory usage for intermediate data. From Figure 6 we can see that Graphiler, DGL-primitive and Seastar consume similar amounts of memory, while the memory usage of PyG-primitive is much higher on large graphs like Reddit due to lack of broadcast fusion. This also explains why PyG-primitive is slower and runs out of memory for GAT on Reddit.

5.2.2 Heterogeneous GNNs

Due to the complexity of hetero-GNNs, current GNN frameworks like DGL and PyG provide multiple implementations. The recommended implementation resembles our HGT running example (Figure 1) due to its simplicity and efficiency on small graphs. However, its memory footprint is quite

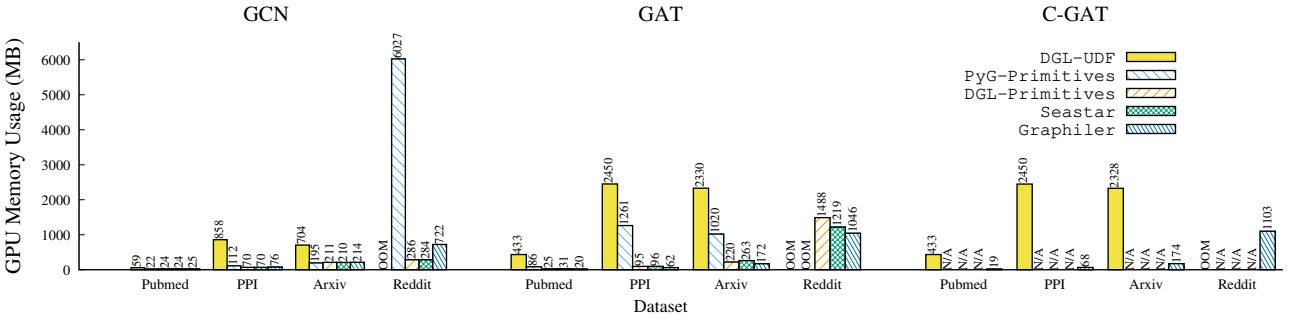

Figure 6: GPU memory usage for homogeneous GNN models on different graph datasets.

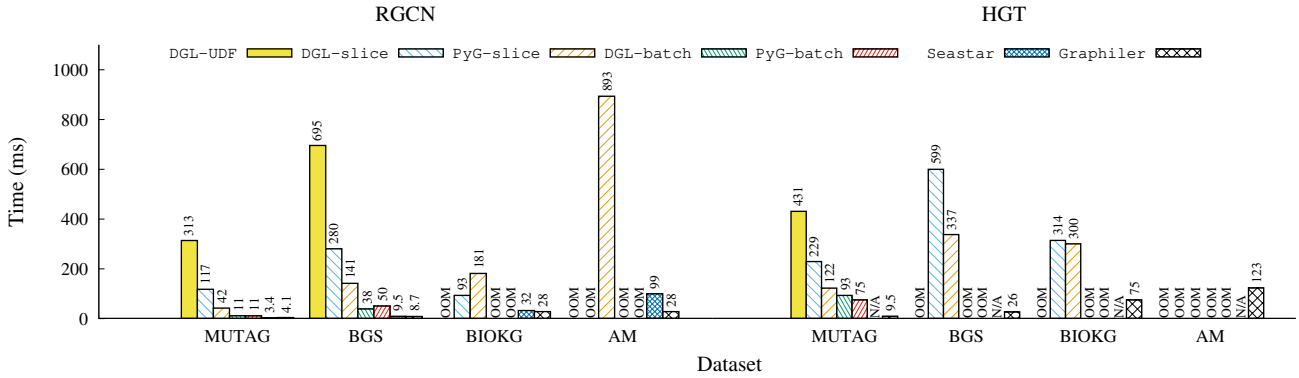

Figure 7: Inference time of hetero-GNN models on different graph datasets.

large due to the broadcasting operations. Another implementation breaks down the overall computation into message passing on each homogeneous subgraph *slice* and aggregates the slice-wise results afterwards. Although it workarounds the memory issue, the computation is fragmented, leading to device under-utilization.

We compare Graphiler with both implementations, which we name DGL-batch and PyG-batch for the first kind and DGL-slice and PyG-slice for the second kind. The Seastar baseline does not naturally support hetero-GNNs, and only provides a hard-coded kernel for R-GCN. It is not clear how to implement other hetero-GNNs using Seastar.

Figure 7 shows that Graphiler significantly outperforms all other baselines for R-GCN and HGT on most datasets. For R-GCN, Graphiler is on average $78\times$, $21\times$, $16\times$, $3.6\times$ and $4.2\times$ faster than DGL-UDF, DGL-slice, PyG-slice, DGL-batch and PyG-batch, respectively, across all datasets. Graphiler achieves comparable performance to Seastar on small graphs, but shows better scalability and is $3.5\times$ faster than Seastar on the largest graph AM.

While the batch baselines are fast on small graphs, they consume much more memory than the slice baselines as shown in Figure 8. Even though the slice baselines are memory efficient, they can still run out of memory on large graphs. Only Graphiler, PyG-slice and Seastar are capable of running R-GCN on all four datasets, and only Graphiler can do so for HGT, while all the other baselines run out of memory; Seastar does not support HGT. This demonstrates that Graphiler is able to automatically find optimizations for hetero-GNNs even when experts fail.

5.3 Breakdown Analysis

We pick GAT and HGT to study the contribution of each optimization in Graphiler.

5.3.1 Performance Improvement

As shown in Figure 9, by simply compiling GAT's message aggregation UDF into MP-DFG to eliminate degree-bucket-based executions, Graphiler already achieves significant speedups, by $55\times$ on average across different graphs. In addition, broadcast reordering produces another $2.4\times$ speedup on average by reducing redundant computations. After applying broadcast fusion, which reduces the number of memory accesses, Graphiler improves GAT's speed by a further $2.2\times$ on average. Fusion is especially useful on dense graphs such as PPI, with a $4.3\times$ speedup.

Table 3 shows the performance improvement breakdown

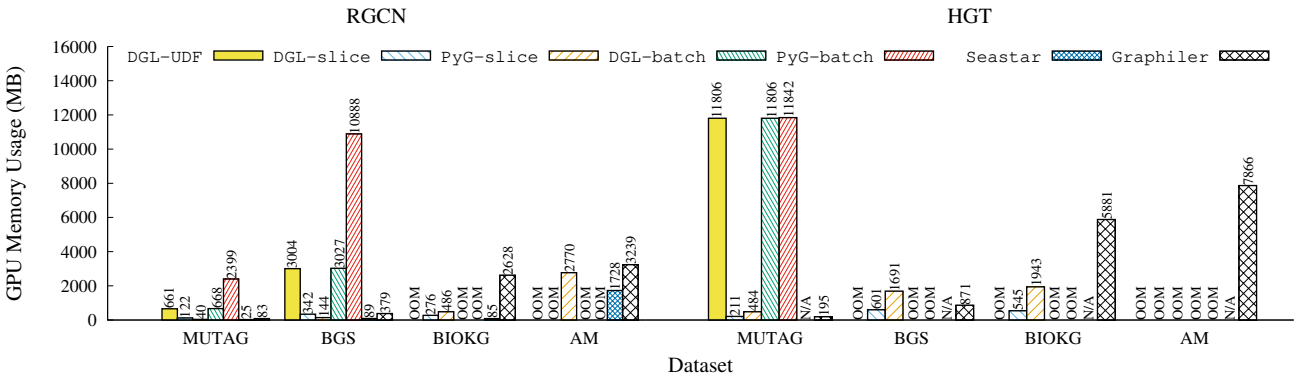

Figure 8: GPU memory usage for homogeneous GNN models on different graph datasets.

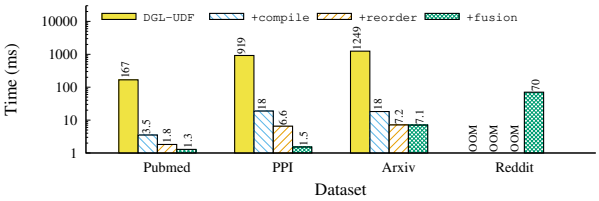

Figure 9: Speedup Breakdown of GAT.

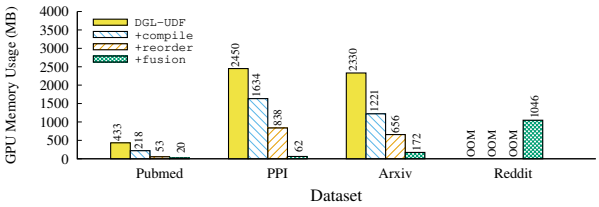

Figure 10: Memory Consumption Breakdown of GAT.

for HGT. Graphiler achieves a $4.9\times$ speedup on MUTAG by compiling UDF into kernels, which is already close to the state-of-the-art PyG-batch baseline. The HGT model does not have the opportunity for broadcast reordering while broadcast fusion boosts the speed by another $9.2\times$.

5.3.2 *Memory Saving*

Figure 10 and Table 4 show memory savings. By removing fragmented computations, Graphiler not only accelerates GNNs as shown in Figure 9 and Table 3, but also noticeably reduces memory consumption, by 43% on average for GAT and 37% for HGT. Broadcast reordering reduces the feature size of nodes, which saves 56% memory on average and also enables more opportunities for broadcast fusion. By combining broadcast reordering and fusion, Graphiler dramatically reduces memory usage, by 76% on average for GAT and 97% for HGT.

Table 3: Speedup Breakdown of HGT.

Datasets	UDF	+compile	+reorder	+fusion
MUTAG	431	88	88	9.5
BGS	OOM	OOM	OOM	26.5
ogbn-biokg	OOM	OOM	OOM	75.2
AM	OOM	OOM	OOM	123

Table 4: Memory Consumption (MB) Breakdown of HGT.

Datasets	UDF	+compile	+reorder	+fusion
MUTAG	11822	7087	7087	220
BGS	OOM	OOM	OOM	957
ogbn-biokg	OOM	OOM	OOM	6095
AM	OOM	OOM	OOM	8034

6 Related Work

GNN Frameworks. DGL and PyG support message aggregation UDFs using primitives, as discussed in §3.3. Another approach, used by PGL and Graph Nets, is *ragged tensors*. However, operator support for ragged tensors is currently limited, as implementing high performance kernels is sometimes challenging. In addition, users need to manage both ragged and dense tensors, which complicates code maintenance. In contrast, Graphiler can directly compile UDFs into efficient execution plans which utilize sparse matrix computation primitives.

Optimizing GNN Computation Graphs. HAG (Jia et al., 2020b) reuses the aggregation result of messages produced from the same node to eliminate redundant computation. While this approach works nearly optimally for nodes, it is not applicable to graphs with edge features, which occur frequently in practice, and are necessary for attention based GNNs. In contrast, Graphiler can eliminate redundancy on both nodes and edges.

(Huang et al., 2021) analyzed performance problems in GNNs and proposed to combine sparse lookup of messages with neural operations to eliminate redundancy. This technique is a special case of Graphiler's broadcast fusion optimization. Seastar (Wu et al., 2021) proposed vertex-centric GNN programming interface using a DFG-based IR with graph type annotations, fusion patterns and kernel optimizations. As shown in §5, Graphiler has several advantages compared to Seastar. First, Graphiler directly compiles highly expressive UDFs, while Seastar requires GNNs to be reprogrammed using a custom and less expressive vertex-centric interface. Second, Graphiler naturally supports hetero-GNNs using its proposed MP-DFG IR, whereas Seastar's GIR can only optimize homogenous GNNs. Lastly, Graphiler has comparable and in many cases superior performance to Seastar.

Efficient Sparse Matrix Computations are core to graph processing and are used to implement a number of important primitives in GNNs. One line of work, including FeatGraph (Hu et al., 2020c), GeSpMM (Huang et al., 2020), FusedMM (Rahman et al., 2020) and GNNAdvisor (Wang et al., 2021b), optimizes GNN kernels using graph specific information such as feature dimensions, node degrees, etc.

Graphiler is compatible with all these methods and can utilize their optimized kernels. None of the previous works studied hetero-GNNs. Graphiler shows that the hetero-GNN computations can be decomposed into traditional sparse operators using different data mappings, which allows combining existing sparse matrix algorithms with Graphiler to further improve performance.

Distributed GNNs. The line of work for training GNNs at scale such as distributed sampling (Zheng et al., 2020; Alibaba, 2019; Park et al., 2020; Zhu et al., 2019), intelligent graph partitioning (Jia et al., 2020a; Lin et al., 2020), efficient communication patterns (Ma et al., 2019; Wang et al., 2021a; Cai et al., 2021; Gandhi & Iyer, 2021), serverless computing paradigm (Thorpe et al., 2021), etc. is complementary to our work: Graphiler can be used within a distributed GNN context.

Graph Processing Systems. Message passing is also widely used in traditional graph processing systems, but their computation patterns and optimization methodology are substantially different to Graphiler. For example, while Gemini (Zhu et al., 2016) uses message combining to reduce network communication, Graphiler applies broadcast fusion to eliminate intermediate data materialization. Similar to Seastar, Ligra (Shun & Blelloch, 2013) and Medusa (Zhong & He, 2013) ask users to rewrite their programs using a fined grained vertex or edge-centric programming model. However, Graphiler directly compiles programs written in the more widely used UDF interface into efficient primitives.

Deep Learning Compilers. Compilers for optimizing deep learning models can be largely divided into two classes. High-level compilers, including XLA (Google, 2017), TorchScript (Contributors, 2018), JAX (Bradbury et al., 2018), Relay (Roesch et al., 2018) perform program-level optimizations. Low-level compilers, including TVM (Chen et al., 2018b), TC (Vasilache et al., 2018), etc. aim to generate efficient kernels. Graphiler belongs to the first family, and uses TorchScript to extract DFG IR from user programs written in PyTorch before performing follow-up IR transformations. An interesting future direction is to combine Graphiler with low-level kernel generation.

TASO (Jia et al., 2019) is a DNN computation graph optimizer which automatically searches for pattern substitutions using a cost-based algorithm. However, it can only be applied to DNNs modeled using standard DFGs, and cannot detect GNN-specific optimizations. As Graphiler's MP-DFG IR annotates a traditional DFG with message passing semantics, it enlarges the search space for pattern substitutions. We leave automatic pattern discovery for GNNs as potential future work. Rammer (Ma et al., 2020) is a DNN compiler which generates efficient kernels to exploit both the inter- and intra-operator parallelism in a DNN model. In contrast, Graphiler targets the problem of computational redundancy in GNN workloads.

7 Conclusion

Graphiler is a compiler stack to optimize GNN workloads. Graphiler tackles a key challenge faced by current GNN frameworks, to achieve both high performance and expressiveness. Graphiler translates GNNs written in flexible and expressive UDFs into a Message Passing Data Flow Graph, enabling automatic discovery of important and GNN-specific optimizations. Experiments show that Graphiler can accelerate homogeneous and heterogeneous GNN models programmed in UDFs by up to two orders of magnitude and achieve performance comparable or superior to expert-optimized implementations, while substantially saves memory consumption.

8 Acknowledgements

We thank anonymous reviewers for their valuable suggestions. Zhiqiang Xie thanks Prof. Jonathan Mace's generous support during his stay at MPI-SWS. Rui Fan is affiliated with the Shanghai Engineering Research Center of Intelligent Vision and Imaging at ShanghaiTech University. This work was supported in part by ShanghaiTech University under grant F-0203-17-005.

REFERENCES

Alibaba. Euler. https://github.com/alibaba/euler, 2019.

Baidu. 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.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

Cai, Z., Yan, X., Wu, Y., Ma, K., Cheng, J., and Yu, F. Dgcl: an efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 130–144, 2021.

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.

Contributors, P. TorchScript. https://pytorch.org/docs/stable/jit.html, 2018.

DMLC. Nnvm compiler: Open compiler for ai frameworks. https://tvm.apache.org/2017/10/06/nnvm-compiler-announcement, 2017.

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

Gandhi, S. and Iyer, A. P. P3: Distributed deep graph learning at scale. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 551–568, 2021.

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.

Google. Xla: Optimizing compiler for machine learning. https://www.tensorflow.org/xla, 2017.

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, 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. F., and Lin, H. (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 22118–22133. Curran Associates, Inc., 2020b. URL https://proceedings.neurips.cc/paper/2020/file/fb60d411a5c5b72b2e7d3527cfc84fd0-Paper.pdf.

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, 2020c.

Hu, Z., Dong, Y., Wang, K., and Sun, Y. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, pp. 2704–2710, 2020d.

Huang, G., Dai, G., Wang, Y., and Yang, H. Ge-spmm: General-purpose sparse matrix-matrix multiplication on gpus for graph neural networks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20. IEEE Press, 2020. ISBN 9781728199986.

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.

Jia, Z., Lin, S., Gao, M., Zaharia, M., and Aiken, A. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems*, 2:187–198, 2020a.

Jia, Z., Lin, S., Ying, R., You, J., Leskovec, J., and Aiken, A. Redundancy-free computation for graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 997–1005, 2020b.

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.

Lin, Z., Li, C., Miao, Y., Liu, Y., and Xu, Y. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pp. 401–415, 2020.

Lv, Q., Ding, M., Liu, Q., Chen, Y., Feng, W., He, S., Zhou, C., Jiang, J., Dong, Y., and Tang, J. Are we really making much progress? revisiting, benchmarking and refining heterogeneous graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pp. 1150–1160, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383325. doi: 10.1145/3447548.3467350. URL https://doi.org/10.1145/3447548.3467350.

Ma, L., Yang, Z., Miao, Y., Xue, J., Wu, M., Zhou, L., and Dai, Y. Neugraph: parallel deep neural network computation on large graphs. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pp. 443–458, 2019.

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.

Park, C., Park, H.-M., and Kang, U. Flexgraph: Flexible partitioning and storage for scalable graph mining. *Plos one*, 15(1):e0227032, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pp. 8024–8035, 2019.

Rahman, M., Sujon, M. H., Azad, A., et al. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. *arXiv preprint arXiv:2011.06391*, 2020.

Ristoski, P., de Vries, G. K. D., and Paulheim, H. A collection of benchmark datasets for systematic evaluations of machine learning on the semantic web. In Groth, P., Simperl, E., Gray, A., Sabou, M., Krötzsch, M., Lecue, F., Flöck, F., and Gil, Y. (eds.), *The Semantic Web – ISWC 2016*, pp. 186–194, Cham, 2016. Springer International Publishing. ISBN 978-3-319-46547-0.

Roesch, J., Lyubomirsky, S., Weber, L., Pollock, J., Kirisame, M., Chen, T., and Tatlock, Z. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, pp. 58–68, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358347. doi: 10.1145/3211346.3211348. URL https://doi.org/10.1145/3211346.3211348.

Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *European semantic web conference*, pp. 593–607. Springer, 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.

Shun, J. and Blelloch, G. E. Ligra: a lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 135–146, 2013.

Thorpe, J., Qiao, Y., Eyolfson, J., Teng, S., Hu, G., Jia, Z., Wei, J., Vora, K., Netravali, R., Kim, M., et al. Dorylus: affordable, scalable, and accurate gnn training with distributed cpu servers and serverless threads. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 495–514, 2021.

Vasilache, N., Zinenko, O., Theodoridis, T., Goyal, P., DeVito, Z., Moses, W. S., Verdoolaege, S., Adams, A., and Cohen, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

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, L., Yin, Q., Tian, C., Yang, J., Chen, R., Yu, W., Yao, Z., and Zhou, J. Flexgraph: a flexible and efficient distributed framework for gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 67–82, 2021a.

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, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., and Yu, P. S. Heterogeneous graph attention network. In *The World Wide Web Conference*, pp. 2022–2032, 2019c.

Wang, Y., Feng, B., Li, G., Li, S., Deng, L., Xie, Y., and Ding, Y. Gnnadvisor: An adaptive and efficient runtime system for {GNN} acceleration on gpus. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pp. 515–531, 2021b.

Wu, Y., Ma, K., Cai, Z., Jin, T., Li, B., Zheng, C., Cheng, J., and Yu, F. Seastar: vertex-centric programming for graph neural networks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pp. 359–375, 2021.

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.

Zhang, C., Song, D., Huang, C., Swami, A., and Chawla, N. V. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 793–803, 2019.

Zheng, D., Ma, C., Wang, M., Zhou, J., Su, Q., Song, X., Gan, Q., Zhang, Z., and Karypis, G. Distdgl: Distributed graph neural network training for billion-scale graphs. *arXiv preprint arXiv:2010.05337*, 2020.

Zhong, J. and He, B. Medusa: Simplified graph processing on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1543–1552, 2013.

Zhu, R., Zhao, K., Yang, H., Lin, W., Zhou, C., Ai, B., Li, Y., and Zhou, J. Aligraph: A comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

Zhu, X., Chen, W., Zheng, W., and Ma, X. Gemini: A {Computation-Centric} distributed graph processing system. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 301–316, 2016.

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