Seastar: Vertex-Centric Programming for Graph Neural Networks

Yidi Wu
The Chinese University of Hong Kong
ydwu@cse.cuhk.edu.hk

Kaihao Ma
The Chinese University of Hong Kong
khma@cse.cuhk.edu.hk

Zhenkun Cai
The Chinese University of Hong Kong
zkcai@cse.cuhk.edu.hk

Tatiana Jin
The Chinese University of Hong Kong
tjin@cse.cuhk.edu.hk

Boyang Li
The Chinese University of Hong Kong
byli@cse.cuhk.edu.hk

Chenguang Zheng
The Chinese University of Hong Kong
cgzheng@cse.cuhk.edu.hk

James Cheng
The Chinese University of Hong Kong
jcheng@cse.cuhk.edu.hk

Fan Yu
Huawei Technologies Co. Ltd
fan.yu@huawei.com

Abstract

Graph neural networks (GNNs) have achieved breakthrough performance in graph analytics such as node classification, link prediction and graph clustering. Many GNN training frameworks have been developed, but they are usually designed as a set of manually written, GNN-specific operators plugged into existing deep learning systems, which incurs high memory consumption, poor data locality, and large semantic gap between algorithm design and implementation. This paper proposes the Seastar system, which presents a vertex-centric programming model for GNN training on GPU and provides idiomatic python constructs to enable easy development of novel homogeneous and heterogeneous GNN models. We also propose novel optimizations to produce highly efficient fused GPU kernels for forward and backward passes in GNN training. Compared with the state-of-the art GNN systems, DGL and PyG, Seastar achieves better usability, up to 2 and 8 times less memory consumption, and 14 and 3 times faster execution, respectively.

Keywords: Graph neural networks, deep learning systems

ACM Reference Format:
Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: Vertex-Centric Programming for Graph Neural Networks. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3447786.3456247

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456247

1 Introduction

Graph neural networks (GNNs) [37, 38, 53, 58] are designed to extract rich information from graph data such as social networks, knowledge graphs and e-commerce networks. In recent years, both academia and industry have presented promising results of using GNNs in important applications such as recommendation systems [48, 64], combinatorial optimization [42], chemistry [23] and physical systems [56].

GNN models are usually composed of 2-3 GNN layers. In its most common form, within each layer, the features of the adjacent vertices of each vertex v in a graph are first transformed using common neural network computations, then aggregated and added to v's own features, and finally passed through activation functions. We refer this process as **graph convolution** in this paper. The output of each layer is fed into the next layer for its computation, except that the output of the final layer is used for downstream tasks such as link prediction and node classification.

Due to the irregularity of graph-structured data, various frameworks such as DGL [59], PyTorch Geometric (PyG) [22], GraphLearn [68], Euler [3], NeuGraph [44] and Roc [33] have been proposed to address the challenges of expressing GNNs' irregular computational patterns and training GNNs efficiently on GPU. From the design perspective, these frameworks can be mostly characterized as GNN operator libraries. Take DGL, a state-of-the-art GNN framework, as an example. DGL provides a graph abstraction, models the graph convolution step using message-passing primitives, and implements the primitives as operators being plugged into an existing DL system (e.g., TensorFlow [2], PyTorch [50], MxNet [16]). At runtime, GNN operators can be treated as normal DL operators by a DL system and the auto-differentiation module of the DL system handles the backward propagation. In this

way, GNN frameworks can design efficient message-passing primitives without modifying the underlying DL system.

However, such a design has two fundamental limitations that are difficult to address. First, existing GNN frameworks use a **whole-graph tensor-centric programming model**. This programming model is designed to implement GNN operators as plug-ins of a DL system, but makes the actual implementation of a GNN model difficult to follow from the conceptual design of the model, especially when graph operations are involved. Second, existing designs expose a trade-off between the generality of operators and performance optimization. Some GNN frameworks prefer generality, for example, [3, 22, 44, 68] decompose GNN computations into graph propagation primitives such as *scatter*/*gather* and connect with other DL operators by materializing the intermediate tensors, which results in *excessive memory consumption* and *massive data movements*. In contrast, DGL [59] provides fused kernels for some commonly used operator combinations, but fused kernels have *limited generality* and DGL's performance is still sub-optimal as we will discuss in §2.

To address these limitations of existing GNN frameworks, we propose **Seastar**, which features a **vertex-centric programming model** with idiomatic Python syntax (§4). Users only need to program a single vertex's logic in familiar Python syntax. To support the efficient execution of various operator combinations, we identify a similar **seastar execution pattern** in training both homogeneous and heterogeneous GNN models (§6.2). Based on the seastar pattern, we design a **generic kernel generator** (§5), which produces highly efficient fused kernels for both forward and backward passes. We propose seastar operator fusion and kernel-level optimizations (§6.3) such as feature-adaptive thread mapping, locality-centric execution, and dynamic load balancing.

We evaluate Seastar (§7) using four widely-used GNN models: GCN [37], GAT [58], APPNP [38] and R-GCN [53] on 12 datasets. Compared with the state-of-the-art GNN frameworks, DGL and PyG, Seastar achieves better usability, up to 2 and 8 times less memory consumption, and 14 and 3 times faster execution, respectively.

Paper outline. In §2, we first give the background of GNN models and discuss the limitations of existing GNN programming models and existing GNN training methods to motivate our work. Then we present an overview of Seastar in §3. In §4, we introduce the vertex-centric GNN programming of Seastar. In §5 and §6, we present the details of code generation and various optimizations in Seastar. We report the performance of Seastar in §7 and discuss related work in §8. Finally, we give an conclusion for our work in §9.

2 Background and Motivation

2.1 Graph Neural Networks

Given a graph G, we denote a directed edge from vertex u to vertex v as uv. Then, each layer of a GNN model can be generally expressed as follows:

$$h_v^{(l+1)} = g(\bigoplus_{u \in \mathcal{N}(v)} (f(h_u^l, h_{uv}^l, h_v^l))), \quad (1)$$

where h_v^l is the feature vector of v in layer l, f and g are customizable neural network modules or functions, and $\oplus$ is a function that aggregates the feature vectors of v's neighbors and edges [67]. The output of a GNN layer is the set of feature vectors of the vertices and/or edges in G. As real-world graphs may have multi-typed vertices and edges, **heterogeneous GNNs** are designed to capture the rich information contained in graph heterogeneity [30, 53, 60, 65]. Usually a hierarchical aggregation scheme is adopted, where the aggregation is first applied on edges of the same type and then on the types to obtain the output embedding of a vertex (to which the edges are incident). We present the formulas of two popular GNN models, GCN and GAT, in Figures 1 and 2.

$$h_v^{(l+1)} = \sigma(b^{(l)} + \sum_{u \in \mathcal{N}(v)} \frac{1}{norm} h_u^{(l)} W^{(l)})$$

Figure 1. The formula of a GCN layer

$$f_u^{(l)} = W^{(l)} h_u^{(l)}, f_v^{(l)} = W^{(l)} h_v^{(l)}$$
$$e_u^{(l)} = (f_u^{(l)})^T attn_u^{(l)}, e_v^{(l)} = (f_v^{(l)})^T attn_v^{(l)}$$
$$e_{uv}^{(l)} = \text{LeakyReLU}\,(e_u + e_v)$$
$$a_{uv}^{(l)} = \frac{exp(e_{uv}^{(l)})}{\sum_{u \in \mathcal{N}(v)} exp(e_{uv}^{(l)})}$$
$$h_v^{(l+1)} = \sum_{u \in \mathcal{N}(v)} a_{uv}^l * f_u^{(l)}$$

Figure 2. The formula of a GAT layer

2.2 Limitations of GNN Programming Models

Although various types of programming models such as message-passing and native dataflow programming are offered by existing GNN training frameworks, they share a common **whole-graph tensor-centric** paradigm as the granularity of operators in these frameworks is in terms of **high-dimensional tensors**. To access a neighbor's features, users explicitly or implicitly create edge tensors using messaging/scattering operation and have to track dimensions carefully to conduct reduction on the right dimension. For example, message-passing systems [22, 31, 59] divide graph convolution into two stages: *message* and *reduce*, where users create messages as edge tensors and conduct aggregation using tensor reduction operations. NeuGraph [44] proposes the Scatter-ApplyEdge-Gather-ApplyNode (SAGA) model, which is similar to the GAS model [26]. However, SAGA shares the same spirit of message-passing, i.e., treating the edge-wise tensors as messages, and its rigid structure may not suit all GNN models. For example, the two 'A's in

SAGA need to be skipped for GCN. Systems adopting native dataflow programming models [3, 68] have a lower-level interface. Users need to care for low-level details such as the IDs of u and v for an edge uv and use operators such as *scatter* and *segment_sum* to implement the *message* and *reduce* functions on their own. Compared with message-passing, even more programming effort is required to implement a GNN model. *The whole-graph tensor-centric design eases the integration with existing DL systems as the granularity of data and operations is consistent with DL systems, but burdens users with the tedious task of translating local computation in Equation 1 to global tensor operations*, which should be handled by GNN training frameworks for better usability and higher user productivity.

2.3 Limitations of Existing Optimizations

Most existing message-passing systems [3, 22, 44, 68] materialize message tensors and the intermediate results generated when creating messages, which results in high memory consumption and data movements between streaming multiprocessors (SMs) and device memory. In these systems, the memory consumption is proportional to the feature size and the number of vertices and edges. This significantly limits their scalability, for example, PyG runs out of memory for a 2-layer GCN with hidden size 16 on the reddit dataset (around 84 million edges) using a NVIDIA GTX 2080Ti with 11 GB device memory.

Operator fusion combines consecutive operators into a single kernel. Within the fused kernel, an operator passes its result to a downstream operator directly using registers without dumping into global memory, thus saving both global memory consumption and execution time. However, it is nontrivial to apply operator fusion to GNN operations. Existing DL systems can only generate simple fused kernels such as fusing several element-wise operations and generally require manual efforts to develop high-performance kernels such as fused conv2d-element-wise [2, 50]. DL compilers such as TVM [17] divide operators into four categories, *injective*, *reduction*, *complex-out-fusible* and *opaque (non-fusible)*, and devise fusion rules for them. Polyhedral model based DL compilers [5, 57] can apply fusion among operators with automatic polyhedron transformation. However, GNN consists of irregular computations due to the irregularity of graph structured data, which requires data-dependent data accessing and aggregations and goes beyond the capability of current DL compilers. In fact, identifying and fusing complex patterns remains to be a difficult problem in general for DL compilers [41].

DGL [59] proposes tailored optimizations for common patterns. DGL merges one edge-wise operation (e.g., adding the features of the two end vertices of an edge) with an aggregation (e.g., sum) on edges into a fused *BinaryReduce* kernel. By avoiding materializing the result of binary operation (which is usually a tensor whose size is proportional to the number of edges), it achieves significant memory saving. However, BinaryReduce only represents a very small subset of operator combinations in the whole GNN design space and delivers poor performance due to its kernel design as we will show in §7. A recent work [31] proposes generalized SpMM (spare-dense matrix multiplication) and SDDMM (sample dense-dense matrix multiplication) and generates kernels using TVM [17]. However, users are required to program using TVM's *compute* and *schedule* primitives to develop new GNN layers, which requires good understanding of GPU's architecture and low-level graph data representation details.

3 Overview

We first give an overview of Seastar in this section, which also provides an outline of the presentation in §4-§6. Seastar differs from existing GNN training frameworks mainly in its vertex-centric programming model (for better usability) and execution plan generation (for higher performance). Seastar's programming model allows users to easily program GNN models with vertex-centric user-defined functions (UDFs). In §4, we present this vertex-centric programming model and compare it with DGL's tensor-centric programming model using GCN and GAT as examples to highlight its benefits. Given a vertex-centric UDF, Seastar generates an efficient execution plan (§5) and applies operator fusion and kernel-level optimizations (§6). Seastar first translates the vertex-centric logic into tensor operations by a tracer (§5) and then identifies operator fusion opportunities by the notion of Seastar computation pattern (§6.2). Seastar effectively utilizes the massive parallelism of GPU by feature-adaptive thread grouping (§6.3.1) and locality-centric execution (§6.3.2), which delineate the key differences (in terms of SIMT parallelization strategy) between Seastar and other solutions such as DGL. Seastar also supports dynamic load balancing to handle skewed loading (§6.3.3).

4 Vertex-Centric GNN Programming

To address the limitations discussed in §2.2, we propose a new vertex-centric GNN programming model, which is inspired by Pregel's vertex-centric model [45] for programming distributed graph algorithms such as PageRank. Our objective is more natural GNN programming so that users' learning curve is flattened. Our key observation is that *Equation 1 takes a form of vertex-centric computation, i.e., it computes the features of a center vertex v by aggregating the features of its neighbors.*

To support vertex-centric programming, we introduce a **function decorator**, *compile*. A function f decorated by *compile* has a single argument v and users only need to program the operation on v. Users only need to include the line "*@Seastar.compile()*" before the decorated function *vc_compute(v)*, and Seastar will compile *vc_compute(v)* and operate on each vertex in the input graph. Users may use

```
#-----------------Seastar GCN-------------------#
def gcn_forward(self, h, norm):
  @Seastar.compile(v_feature={'norm':norm,'h':h})
  def vc_compute(v):
    return sum([torch.mm(u.h, self.W) * u.norm
                for u in v.innbs])
return torch.sigmoid(vc_compute() + self.bias)

#-----------------Seastar GAT-------------------#
def gat_forward(self, h):
  h = torch.mm(self.W, h)
  eu = h * self.attn_u.sum(dim=-1)
  ev = h * self.attn_v.sum(dim=-1)
  @Seastar.compile(
    v_feature={"eu" : eu, "ev" : ev, "h" : h})
  def vc_compute(v):
    e = [torch.exp(self.leakyRelu(u.eu+u.ev))
          for u in v.innbs]
    a = [c/sum(e) for c in e]
    hu = [u.hu for u in v.innbs]
    return sum([a[i]*hu[i]
                for i in range(len(v.innbs))])
return vc_compute()
```

Figure 3. Seastar's implementation of GCN and GAT

```
#-----------------DGL GCN-------------------#
def gcn_forward(self, graph, h, norm):
  h = torch.mm(h, self.W)
  h = h * norm
  graph.srcdata['h'] = h
  graph.update_all(fn.copy_src(src='h', out='m'),
    fn.sum(msg='m', out='h'))
  h = graph.dstdata['h']
return torch.sigmoid(h + self.bias)

#-----------------DGL GAT-------------------#
def gat_forward(self, graph, h):
  h = torch.mm(self.W, h)
  eu = h * self.attn_u.sum(dim=-1)
  ev = h * self.attn_v.sum(dim=-1)
  graph.srcdata.update({'h': h, 'eu': eu})
  graph.dstdata.update({'ev': ev})
  graph.apply_edges(fn.u_add_v('eu', 'ev', 'e'))
  e = self.leaky_relu(graph.edata.pop('e'))
  graph.edata['a'] = self.attn_drop(
          fn.edge_softmax(graph, e))
  graph.update_all(fn.u_mul_e('h', 'a', 'm'),
    fn.sum('m', 'h'))
  rst = graph.dstdata['h']
return rst
```

Figure 4. DGL's implementation of GCN and GAT

native Python syntax in a decorated function, e.g., using the 'dot' operator to access the features of neighbors, and using list comprehension and reduction to compute and aggregate on the neighbor features.

The code snippet in Figure 3 shows how GCN and GAT are implemented in Seastar. The key step of GCN can be implemented succinctly with only one line of code (Line 5). More importantly, we can see a clear correspondence between the GNN formulas in Figure 1 and Figure 2 and the vertex-centric implementations (i.e., *vc_compute(v)*) in Figure 3. The benefit is bi-directional: users can implement a GNN model easily and users can learn the GNN model by directly checking its implementation.

In Figure 4, we also present the implementation of GCN and GAT using DGL, which is one of the most popular message-passing GNN training frameworks. Consider the GCN implementation of DGL in Figure 4. Line 5 stores a tensor h in *graph.srcdata* with key $'h'$. Line 6 executes the message passing step for the whole graph, where $fn.copy\_src$ is the message step, which copies the tensor stored in *graph.srcdata* with key $'h'$ and produces a message tensor with key $'m'$. Then the reduce step $fn.sum$ aggregates the $'m'$ tensor and stores the output in *graph.dstdata* with key $'h'$. Line 8 retrieves the aggregated result from *graph.dstdata*.

Compared with Seastar's GCN implementation in Figure 3, the above GCN implementation in DGL is more complicated due to the extensive use of specialized functions (e.g., $fn.copy\_src$, $graph.update\_all$, $fn.edge\_softmax$). Moreover, referring to the formula of GCN in Figure 1, it is hard to find a correspondence between the GCN formula and DGL's GCN code in Figure 4.

Table 1. Seastar API

	Attribute	Explanation
Seastar	compile	Vetex-centric decorator
v	innbs	list of in-neighbors
	inedges	list of in-edges
	key_name	value of vertex features
e	src	src vertex
	dst	dst vertex
	type	edge type
	key_name	value of edge features

In addition, DGL imperatively executes the operators one by one and materializes intermediate results between operators. In contrast, Seastar separates the operator definition from the operator execution, which allows us to identify patterns (from the UDF, *vc_compute(v)*) for kernel-level optimizations that significantly increase the training throughput (§6).

We summarize some of API of Seastar in Table 1. The API only includes the *compile* decorator and the attributes of a vertex and edge, as users may simply use Python syntax in their code. With this simple API, we can already implement most of the homogeneous and heterogeneous GNN models that are supported by PYG and DGL.

5 Seastar Code Generation

This section describes the code generation process of Seastar, i.e., how Seastar translates a vertex-centric program into a computational graph composed of tensor operations and how

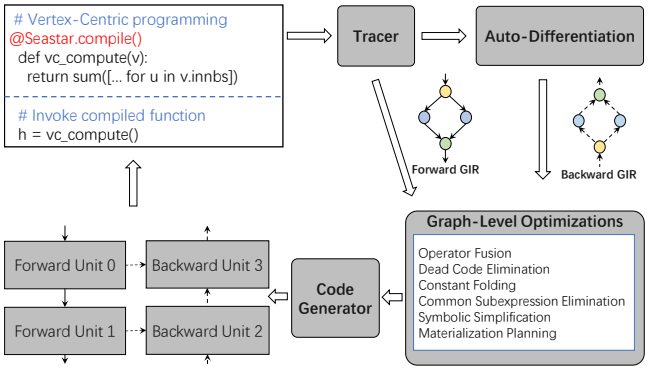

Figure 5. The architecture of Seastar

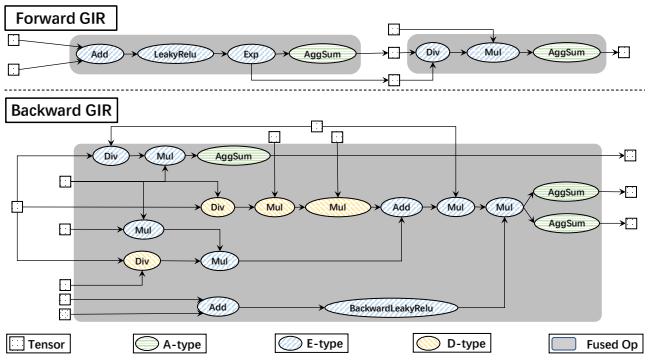

Figure 6. The computational graph of GAT

Seastar integrates with a DL backend's runtime. Seastar's primary DL backend is MindSpore [32]. MindSpore features auto-parallel training that simplifies and accelerates distributed model training and leverages tensor-compilation techniques for optimal performance, which in turn also improves the performance of GNN training using Seastar. However, Seastar decouples its code generation and compilation process from the backend DL system, and it can also be integrated with other popular DL backends. This design is driven by following reasons. First, we find that merging with a specific DL backend such as MindSpore or PyTorch would require many changes in its intermediate representation and code optimization process, which is a non-trivial task (for both merging and update maintenance) as popular DL systems are being actively upgraded. More importantly, a decoupled design allows us to easily use the vertex-centric programming model in different DL backends by writing a light-weight interfacing code. For example, in this paper we used PyTorch as the DL backend for Seastar in our experiments for fair comparison with DGL and PyG, as they both use PyTorch as the DL backend.

Figure 5 shows the major components of Seastar and the overall code generation process. In the vertex-centric function, all operations conducted on each center node and its neighbors are traced and recorded using a *graph-aware intermediate representation* (*GIR*) in §5.1. We embed an automatic differentiation engine for GIR in §5.2. The generated code will then be compiled and wrapped into a generic kernel executed by a DL backend as described in §5.3. We illustrate each component with the code snippet of GAT in Figure 3 as a running example.

5.1 Tracer and GIR

Operator tracing. Similar to PyTorch's JIT tracer [50], we use operator overloading to record the operations executed in the vertex-centric function. The features of each vertex and edge can be accessed using the keys in the vertex_feature and edge_feature dictionary with the 'dot' operator. Each feature vector is a symbolic tensor that inherits various attributes (e.g., data type, shape, device information, whether requiring gradient) from the corresponding tensor given in the dictionary. For the shape attribute, since the corresponding tensor batches vertex/edge feature vectors along the first dimension, we strip away the first dimension and assign the remaining dimensions as the shape. All operators in a DL backend and the methods of the tensors will be monkey-patched with a new version inside the decorator. The patched version records the operation together with the features it operates on and produces new features. The patched version also creates tensors with the attributes of the input features and executes the operators with the original version to enjoy the type inference provided by the DL backend. The traced program is a DAG since tracing essentially unrolls loops and branches. Traced DAGs are then compiled and cached after the first run so that later on it can be executed directly.

As shown in the vertex-centric implementation of GAT in Figure 3, users provide a $v\_feature$ dictionary as an argument of the compile decorator. The values in the dictionary (i.e., eu, ev and h) are tensors and each row in them is to be interpreted as the embedding feature of a vertex by Seastar. When the decorated function (or UDF for short) is invoked for the first time, Seastar will create an attribute for the center vertex v and its neighbor u for each key-value pair in the dictionary, where the name of the attribute is the key (e.g., $u.eu$ and $v.ev$) and the content of the attribute is a sliced version of the original tensor (e.g., the first row of the tensor) to reflect the fact that the attribute belongs to a single vertex. Note that the sliced tensor is only used in Tracer and will be discarded after the first run. Then, the operators in the UDF are to be invoked one by one. Besides carrying out the computation on the sliced tensors and producing the results as normal operators, they additionally create a node in a global DAG using their signatures, input and output information. The resulting DAG of GAT is shown in the forward GIR in Figure 6.

GIR. The DAG produced by Tracer follows the computational graph representation that is commonly adopted in DNN systems [2, 16, 50]. The nodes in a computational graph represent operations, which take one or more tensors as input and produce new tensors as output. Operations can also

have attributes such as the slope of the *leakyRelu* operation. The DAG specifies the computation for each vertex v. This DAG can be separately executed for each vertex, but this approach will severely damage the performance of GPU due to limited parallelism. Seastar groups the vertex-centric operations for all vertices and executes them together. To this end, Seastar annotates the tensor in the vertex-centric DAG with a graph type S (*source*), D (*destination*), E (*edge*), or P (*parameter*). S, D and E mean that the tensor corresponds to a row of full tensor that is a source-wise, destination-wise, and edge-wise embedding, respectively. P means that the tensor is a parameter shared by all vertices (e.g., the slope of leakyRelu). A tensor's graph type can be determined from the way in which the vertex-centric program accesses it. In the GAT example, eu is accessed by source vertex u and thus its graph type is S, while ev is accessed by destination vertex v and thus has a graph type D. Knowing the graph type of eu and ev, we can then batch all $u.eu + v.ev$ operation in GAT into one kernel by mapping a thread on GPU to each edge and use the source or destination vertex id of each edge as an index to query the corresponding rows in the original tensor, i.e., the tensors eu and ev in the $v\_feature$ dictionary. We will discuss more usage of graph types in automatic differentiation and optimizations in §5.2 and §6.

Graph type inference. We infer the graph type of intermediate results according to the following rules:

1. For operators conducting edge-wise aggregation (e.g., the sum operator in GAT), if the input type is S (or D), its return type is the opposite type, i.e., D (or S). If the input is of type E, the return type is determined by the direction of training. By default, for forward (or backward) pass of GNN training, the return type is D (or S).
2. For operators taking single graph-type inputs, its output tensor has the same type as the input.
3. For operators having more than two types from S, D, E, its output is of type E.
4. When used as one of the inputs together with other graph types, type P does not have effect on the outputs' graph type.

The graph type of operators. We define an operator's graph type based on their output's graph type. Specifically, an operator is defined as type S, D or E if its output is of type S, D or E, respectively. The graph type of operator summarizes what kind of index is required to carry out the computation of this operator. For examples, for an S-type or D-type operator, vertex ids are used as indices to access the embedding. E-type operator requires both vertex id and/or edge ids in order to access all required embedding since its input can be of type S, D and E. Additionally, we define the graph type of aggregation operators (e.g., the AggSum operator in GAT) as a new type A to differentiate its unique data access and execution pattern.

We show the DAG of GAT with operator-type annotation in Figure 6 and explain the forward GIR in details to illustrate the concept. Initially, the operands of the Add operator are of S type (eu) and D type (ev). According to graph type inference Rule 3, its output will be E-type. Thus, the graph type of the Add operator is also E-type. The type of LeakyRelu is E since it has only one E-type input (Rule 2). Similarly, Exp is also E-type and produce an E-type output (Rule 2). For the AggSum operator, it is A-type by definition and it returns a D-type input since the GIR is for forward pass. Similarly, we can determine the graph type of the remaining operators.

5.2 Auto-Differentiation on GIRs

At compilation time, we do not have the actual value of gradient, we use placeholders to represent it and the actual value will be either provided by the auto-differentiation system of the DL backend or computed during backward execution at runtime. We start from the output of the whole vertex-centric computation. We find its producer and generate the backward operators to compute gradients for producer's inputs using the output and its gradient. If an input already has gradient, an additional Add operator will be generated to accumulate the newly computed gradient. The process then goes recursively for each input. We make sure that an operator's all downstream operators are differentiated before itself to avoid propagating back partially aggregated gradient by tracking its downstream dependencies. Besides, we need to pay attention to the graph type of the operators and follow the graph type inference rule when implementing the backward logic of the operators to obtain the correct gradient. For example, for E-type operators, we need to ingest edge-wise aggregation operators to compute the gradient of S-type or D-type input. We depict the backward GIR of GAT in Figure 6.

5.3 Code Generation and Execution

Generating execution unit. For GIR, we run various graph-level optimizations to generate an optimized computational graph (§6). Among them, Seastar operator fusion combines multiple operators of the computational graph into a single operator.

We then divide the computational graph into fused and un-fused execution units, where a fused/un-fused unit consists of a set of fused/un-fused operators. For un-fused execution units, we directly emit the operators in the DL backend. For each fused unit, we generate a kernel according to the Seastar fusion template and compile it. We determine the dependency among the execution units according to the data dependency between the operators in the units. At runtime, we follow the dependency to execute the units one by one.

Runtime execution. At runtime, we wrap each compiled execution unit into a user-defined function to be plugged into a DL backend (e.g., by inheriting the function *autograd* in PyTorch). When users call the compiled vertex-centric

function. Seastar dispatches the execution unit following the dependency among the units. For un-compiled kernels, Seastar simply invokes the DL backend's implementation. The output is recorded in a tensor map if it is required by a subsequent program. For compiled kernels, we look for their required inputs in the tensor map, invoke the compiled kernels and record their outputs. During backward propagation, the DL backend invokes the backward logic of an execution unit and provides the gradient as input. Seastar then takes control and invokes the corresponding backward execution unit. We clear the tensors in the state map eagerly once there is no dependency on them to avoid memory leakage.

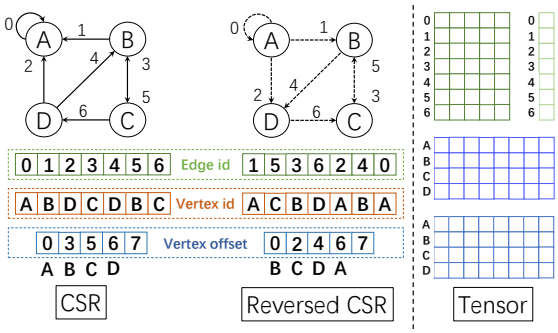

Figure 7. Graph and data representation in Seastar

6 Seastar Optimizations

The computational graphs obtained from the tracer and auto-differentiation engine usually consist of redundant or useless computations and, most importantly, plenty of operator fusion opportunities that existing GNN frameworks fail to exploit (§2.3). To address these problems, we present Seastar operator fusion and its kernel-level designs. We also remove redundant/useless computations using classic optimizations such as common sub-expression elimination, constant folding, and mathematical simplification.

6.1 Graph and Data Representation

We first introduce the graph and data representation in Seastar. Datasets used in GNN training usually consist of two parts: a sparse directed graph and the feature vectors for vertices and edges. Vertex/edge features are represented as tensors, whose first dimension is indexed by the vertex/edge id (starting from zero). For graph representation, we adopt the widely used Compressed Sparse Row (CSR) format because of its low memory consumption and access efficiency for sparse graphs. We also store a *reverse* CSR format for the backward pass. Figure 7 shows the data layout for an example graph. We sort vertices in descending order of (out-)in-degrees for generating the (reverse) CSR format. The sorting is necessary for kernel-level optimizations in §6.3. We create a separate array for edge ids following DGL [59], which is of the same length as the vertex id array for accessing E-typed tensor. It may be tempting to directly use the index of the vertex id array as edge ids to save memory consumption, but the mapping between the index and edge ids is invalidated in the reverse CSR format. We will discuss this issue in greater details in §6.3.4. At the start of a GNN program, the graph and feature tensors are constructed from the input dataset and then moved to GPU memory together with the parameter tensors in the GNN model to be trained.

6.2 Seastar Operator Fusion

Two operators in a producer-consumer relationship can be fused into a single operator if and only if the result of the producer can be directly pipelined to the consumer without writing intermediate results to the global memory of GPU. TVM [17] categorizes fusible operators into three categories: *injective*, *reduction* and *complex*. Then it exploits the following fusion opportunities: *injective-injective*, *injective-reduction*, and *complex-injective*. But graph convolution does not belong to any of the categories due to its graph-dependent execution.

Seastar pattern. To devise a generic operator fusion scheme for GNN workloads, we characterize GNN models and observe that despite the variety of GNN models, they share a similar execution pattern, which we name as the **Source-Edge-Aggregation star** (**seastar**) pattern. The pattern is of a *star* shape as the computation is vertex-centric with edges around a center vertex. We explain the seastar pattern with the graph and data layout presented in §6.1 and show its connection with Equation 1. GNNs usually start with processing vertex-wise features of type S. Operators access the S-type inputs according to the vertex's id and produce outputs that correspond to h_u^l or h_v^l. Then the outputs are processed by E-type operators, where we conduct the computation on each edge following the destination, source, and edge ids retrieved from the CSR indexes. Specifically, the destination vertex's id is assigned to be its location in the vertex offset array. The source and edge ids for the k-th vertex can be accessed using the offset range stored at the k and $(k+1)$-th position of the vertex offset array. We then use these ids to access the embedding stored in the corresponding row of the tensors and then apply the E-type operator on these rows (i.e., function f in Equation 1). Finally, values produced by edge-wise computations are aggregated according to the destination vertices' ids with A-type operators and written to the corresponding row in the output tensor (i.e., function $\oplus$). The result can be further processed using a Dropout layer or activation layer such as *LeakyRelu* (i.e., function g).

Fusion opportunities in Seastar. We describe three kinds of fusion opportunities: *S-E*, *E-E* and *E-A*. *S-E* fusion combines a producer in the *Source* stage and a consumer in the *Edge* stage into one operator. In the *Edge* stage, we have access to the edge id as well as the source and destination vertex ids. Thus, we can defer the *Source* stage computation

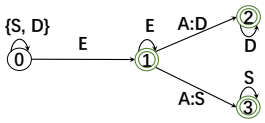

Figure 8. Finite state machine for operator fusion

to the *Edge* stage. Specifically, in the *Edge* stage, we query the corresponding row of the tensor by the vertex's id and execute the *Source* stage computation using the retrieved vector. Now each edge will have the resultant vector of the *Source* stage. We can then feed the output of the *Source* stage directly to the downstream operator in the *Edge* stage. *E-E* fusion is more straightforward, as they require the same set of indices to access the embedding. The LeakyRelu and Exp operation in the forward pass of GAT can be fused following the *E-E* fusion. *E-A* fusion is done by first conducting the computation on the *Edge* stage and then using the destination vertex's id to carry out the aggregation (e.g., the Exp and AggSum operators in the forward pass of GAT can be fused.).

Identifying seastar patterns. To identify seastar patterns in a computational graph automatically, we summarize Seastar's valid fusions as a finite state machine in Figure 8. We associate each operator with a state that is determined by its own graph type and its upstream operator. For example, if an upstream operator is in State 3, a fusible downstream operator must be of type *S*. We differentiate two types of aggregation operators $A:D$ and $A:S$. $A:D$ returns a *D*-typed tensor and $A:S$ returns an *S*-type tensor. The differentiation is necessary as their fusible downstream operators are of different graph types. To match seastar patterns, we visit each operator in the computational graph following the topological order. We set each root's state by transiting from State 0 according to its graph type. Then for each visited operator, we set the state of their children. If the graph type of a child is a valid transition from its parent's state, its state will be set following the transition. The parent operator is added to the the child's transition upstream set. Otherwise, the transition upstream set of the child operator is set to be null and its state is set by transiting from 0 according to its graph type. Operators having multiple upstream operators will be set more than once. To break ties, we adopt the last-write-wins scheme. The rationale is that due to the topological order, operators will try to be fused with its "nearest" parent to avoid the scenario where an operator *Q* is fused with a parent operator *P*, but *P* has a downstream operator that cannot be fused with *Q* (e.g., we encounter such a case in the forward pass of GAT). After all operators are visited, for each operator in States 1, 2 and 3, we trace back to its transition upstream recursively and merge them into a fused operator.

We use the forward pass of GAT in Figure 6 to illustrate the fusion algorithm. Initially, the Add operator is in State 0 and of type *E*, and thus it moves to State 1 after transitioning according to type *E*. Its child operator LeakyRelu is also of type *E*, which is a valid transition from State 1 itself. Thus, LeakyRelu stays in State 1 and it can be fused with its parent Add operator. Similarly, Exp and AggSum are transitioned to State 1 and State 2, respectively. They can be fused with all the previous operators. For the Div operator, its nearest parent in topologically sorted order is AggSum, which is in State 2. However, Div is of type E and the only valid transition from State 2 is by type *D*. Thus, Div is not fused with its parent and the FSM is re-started.

6.3 Kernel-Level Optimizations

In order to support efficient execution of fused kernels, we need to carefully map the massive parallelism of a GPU to the computation and data. We find three types of parallel execution opportunities: *feature-wise parallelism* (*FP*), *vertex-wise* (*VP*) and *edge-wise* (*EP*). FP is the most notable difference of GNNs from traditional graph processing workloads such as PageRank and is not exploited by GPU graph processing systems. In GNN training, each vertex is associated with an embedding vector (or tensor) with up to hundreds or thousands of elements. In contrast, in PageRank there is only a single PageRank value. Massive performance gain can thus be achieved by exploiting FP.

DGL delegates execution to minigun, which is a GPU execution engine for graph operations. Its execution strategy is heavily influenced by GPU graph processing systems [20, 61]. Minigun exploits EP and FP by first assigning thread blocks to edges and then mapping threads in a block to the embedding dimension. One key step is to determine the id of the destination vertex of an edge. Threads assigned to an edge conducts a binary search on the vertex offset array to look for an offset range that this edge belongs to. For example, for the sample graph shown on the left of Figure 7, threads that are assigned edge id 4 will find that range [3, 5) contains the edge and thus return *B* as the target destination vertex.

By exploiting EP, DGL balances the loads among threads and blocks regardless of vertex degrees skewness. However, there are two key limitations of this approach. First, the binary searches require $O(log(N))$ search instructions, where N is the number of vertices in the graph. It incurs a significant overhead when the number of vertices is large as we will show in §7. Second, the data locality is poor. Edges incident to the same vertex may be assigned to different blocks. Thus, atomic instructions are required to avoid data race when writing to the same destination vertex, e.g., in the case of aggregation. Besides, the embedding for the same destination vertex may be loaded by different thread blocks, which further increases the number of load instructions.

Seastar proposes feature-adaptive group (§ 6.3.1) to increase the occupancy of GPUs and exploits FP. Seastar exploits VP with the locality-centric execution strategy (§ 6.3.2) and leverage dynamic load balancing (§ 6.3.3) to address the

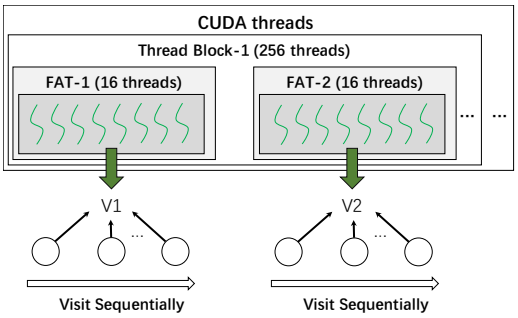

Figure 9. An illustration of Seastar's kernel execution when the feature size is 16 and the thread block size is 256

vertex degree skewness in real-world graphs. We provide an overview of Seastars' design in Figure 9 when the feature size is set to 16.

6.3.1 Feature-Adaptive Groups. The original feature of vertices may have dimension up to a few thousands but as the GNN layers stack up, the hidden features shrinks to a more compact representation, e.g., one or a few scalars. When the feature dimension is large, we can directly assign one or more thread blocks to work on the features (denote its dimension as D). Each block consists of 2^k threads, where k is the largest integer satisfying $2^k \leq D$. Threads in a block focus on the computation for a consecutive range of values in a feature vector, resulting in coalesced memory access and SIMT execution (except a small amount of leftovers when the feature size is not divisible by the number of threads).

However, each block will consist a very small number of threads if D is small, which leads to severe low GPU occupancy due to the hardware limit: the number of blocks that can concurrently run on a streaming multi-processor. For example, 16 threads per block will reduce the theoretical upper-bound occupancy to 25% on a 1080Ti GPU. This motivates us to only reduce the number of threads assigned to features based on the feature dimension while keeping the block size large enough. We propose the abstraction of **feature-adaptive thread (FAT) group**. A FAT group consists of 2^k threads as defined previously to best adapt to the change of feature dimensions. The block size is fixed to a constant (e.g., a block size of 256 is commonly adopted). Physically, a FAT group can share a block with other groups or be mapped to one or more blocks. For each thread, its group id can be calculated from dividing global thread id by block size and its thread id in the FAT group is the remainder.

6.3.2 Locality-Centric Execution. Seastar exploits VP, which emphasizes on data locality. Seastar assigns each vertex to a FAT group and for each vertex, the FAT group carries out edge-wise computation edge by edge — a pattern we name as **vertex-parallel edge-sequential**. This looks undesirable as we ignore edge-wise parallelism, but it is actually preferable for the following reasons. First, parallel edge-wise computation results in either hierarchical or atomic operations for aggregation, introducing frequent synchronization among threads explicitly or implicitly. In contrast, when threads visit edges sequentially, they can accumulate the partial results in registers without any form of synchronization. Second, the destination vertices' features can be loaded only once and stored in registers, thus resulting in excellent data locality. This approach basically reduces the number of load instructions for the destination vertices from the number of edges to the number of vertices, as the former is usually much larger.

6.3.3 Dynamic Load Balancing. Different FAT groups may end up with skewed loading in the locality-centric scheme of Seastar since real-world graphs usually follow a power-law degree distribution. We propose dynamic load balancing to address the problem of skewed loading as follows.

Dynamic scheduling. Load stealing is a commonly used strategy for load balancing [7, 10]. A simple and straightforward design is to use the trick of "persistent threads", where threads in a FAT group run in an indefinite loop and atomically increase a global vertex counter once they finish the current execution until no vertices are left. However, the atomic operation on GPU memory incurs an overhead, which can be large when the amount of vertices is large. To solve this, we leverage the block schedulers on GPU hardware to help us with the scheduling since the overhead of block scheduler hardware is negligible. Specifically, we launch as many blocks as it requires to cover all vertices and rely on the block scheduler of GPU to dynamically launch and retire the blocks (hence the name dynamic scheduling). With dynamic scheduling, the time-consuming computation for high-degree vertices can be overlapped with short ones, thus reducing the effect of skewed workloads.

Degree sorting. To further reduce the overhead caused by unbalanced load, we propose to sort vertices according to their degrees. The benefits for degree sorting are two-folds. First, even though high-degree vertices still exist, we know that they are ordered in the front of vertex array and we can process them earlier so that their computation can be better overlapped with the execution of a large number of low-degree vertices. Second, a consecutive range of vertices in sorted vertex offset array have similar (if not identical) degrees. When the feature dimension D is small, the FAT groups assigned to consecutive vertices will be from the same thread block. A uniform workload within this small range helps eliminate intra-block load imbalancing.

Though sorting incurs overhead, our key insight is that for GNNs, the graph structure is either fixed (for full graph training) or can be prepared to be independent of the current training iteration (e.g., sampling the mini-batches in background). We could sort the vertices without slowing down the current training.

Algorithm 1: Seastar CUDA template

```
   Input: group_size: size of each group,
   csr: graph stored in csr format
1  thread_id = threadIdx.x + blockIdx.x * blockDim.x;
2  vid = thread_id / group_size;
3  tid = thread_id % group_size;
4  if (vid < num_vertices) then
5      beg = __ldg(csr.vertex_offset + vid);
6      end = __ldg(csr.vertex_offset + vid);
7      Emit Aggregation initialization;
8      for (;beg < end; ++beg) do
9          eid = __ldg(csr.edge_ids + beg);
10         uid = __ldg(csr.vertex_ids + beg);
11         // S and E stage
12         Emit Edge-wise instructions;
13         Emit Aggregation computation;
14     end
15     // A stage
16     Emit Aggregation output;
17     Emit Vertex-wise instructions;
18 end
```

Block scheduling. Given the fact that high-degree vertices are clustered in the front of the array, Seastar uses the order of block scheduling as the order of processing vertices: earlier scheduled blocks process vertices in the front. However, the built-in block id available in each kernel does not provide the guarantee directly, i.e., blocks with smaller ids do not necessarily get scheduled earlier. Luckily, based on the results obtained by others [35, 40] as well as in our own benchmarking, there does exist a strong correlation between block id and the time it gets scheduled for 1-D grid, which means that we can simply use the block id and avoid atomic instruction. Note that violation of the observed correlation will slightly affect the processing order but have no effect on the correctness of processing.

To summarize, we start from the feature dimension and assign a FAT group to perform data parallel operations on the features. The remaining parallelism (if any) in a block is then distributed to vertices for edge-wise computation and aggregation. The vertices are sorted according to their degrees and we can use atomic blockId or built-in blockId to schedule the execution on the vertices. For example, if the feature dimension D is 16, we fix the group size to 16 and suppose that we set the block size to be 128 (a tunable parameter B) then the block will be assigned 8 vertices with similar degrees to process. Effectively, each warp of 32 threads will be divided to work on 2 vertices. We launch as many blocks as the number of vertices divided by 8 so that when the previous block retires, subsequent blocks can fill the SM dynamically. Such a strategy achieves load balancing within blocks and warps, coalesced memory access on features, and fast aggregation using registers. We design a micro-benchmark of accessing neighbors' feature vectors to verify the effectiveness of this design and compare with DGL's load-balancing method. Seastar outperforms DGL significantly, with up to 946 times speedup for the largest reddit dataset. Detailed decomposition of the contributions of individual designs are presented in §7.

We show the CUDA template of Seastar in Algorithm 1. For each thread, Lines 1-3 compute its group id and thread id within the group. Lines 8-14 conduct the edge-sequential execution. Lines starting with *Emit* are instruction placeholders, which will be replaced with the specific implementation of operators during the code generation process. To make the implementation of the operators easier, we abstract away the data-accessing step of operators so that developers only need to write sequential code for scalar inputs.

6.3.4 Backward Propagation of Seastar.

We discuss how we conduct the backward execution of Seastar. Forward and backward training present a perfect symmetry: if the forward computation follows the Seastar pattern, so will the backward pass. This can be reasoned using the flow of data. In the forward training, each center vertex receives data from its neighbors. In the backward pass, the gradient of the center vertex needs to be sent to all its neighbors following the reverse direction of edges. But from its neighbors' point view, we find that in the backward pass it aggregates from all out-going edges. It is also easy to verify that for source-wise (destination-wise) operations in the *Source* (*Aggregation*) stage of the forward pass is now *Aggregation* (*Source*) stage of the backward pass. This means that we can reuse all the designs except that we should flip the in-edges to out-edges and sort the vertices again (which can be done in a pre-processing stage).

But we need to carefully handle the edge ids. As after we flip the direction of the CSR representation, the same edge index does not correspond to the original edge anymore. Thus, we need to remember the edge ids in the forward pass and sort/flip them together with the vertex index array.

6.3.5 Heterogeneous Seastar.

Heterogeneous graphs bring a new type of parallelism: *edge type parallelism* (*ETP*). The aggregation operation among **edges** may be different with the aggregation on **edge types**. Existing systems either execute for each edge type one by one or use batched operations such as *batached_matrix_multiplication* (*BMM*). The latter falls back to dataflow programming.

A generic hierarchical aggregation scheme first aggregates over edges of the same type and stores the results, and then aggregates over edge types. Due to the same reasoning for sequentially executing the edge-wise computation, it may be preferred to conduct aggregations sequentially. Our key observation here is that common hierarchical aggregation operations such as *max*, *sum* and *min* permit a sequential algorithm. For example, *sum* aggregation over edges then *max* aggregation over edge types can be conducted by two for-loops: the outer loop using *max* and iterating over edge

types, while the inner loop iterating over edges. This effectively transforms the problem of heterogeneous training to the homogeneous case.

Implementing this design requires us to introduce a secondary sorting key, the edge type, to sort the edges of each vertex in addition to vertices' degree. Then we can unroll the outer loop and detect changes of edge types, and use it as a signal to conduct the original outer loop accumulation. Edge types are stored alongside with edge ids and can be indexed using edge ids. We have also considered a more compressed way of storing edge types using a similar scheme as CSR. Basically, we add one more layer of indirection between the vertex offset array and the vertex id array by introducing a type offset array. Intuitively, this scheme saves memory by sharing the edge type value among edges of the same type. We do not choose this scheme for two reasons: type offsets need to be stored for both forward and backward passes, while storing edge types alongside with edge ids can be shared for both passes. In fact, the compressed format is only useful when the dataset satisfies the condition $Ne/Nt > 2$, where Ne is the number of edges of the graph and Nt is the summation of the unique types of all vertices. For the four popular datasets used in our experiments, the highest and lowest ratios are 1.923 and 1.385, indicating that using the edge-type array is a better option.

7 Experimental Evaluation

We ran Seastar using PyTorch 1.6.0 as the DL backend and compared with DGL-0.4 [59] and PyG-1.6.0 [22]. Both DGL and PyG also use PyTorch as the backend. We used DGL and PyG as the baselines as they are the most popular and state-of-the-art GNN training frameworks.

We used three hardware platforms. The first one is equipped with an Intel(R) Xeon(R) E5-2660 v4 CPU (56 logical CPUs, 2.00GHz), 256 GB of memory and NVIDIA GTX 1080Ti GPUs with 11 GB device memory. The second one is equipped with NVIDIA GTX 2080Ti with 11 GB device memory. The third one is equipped with NVIDIA Tesla V100 with 16 GB device memory. We compiled generated kernels using CUDA 10.1 with the highest optimization level. Note that Seastar only uses a single GPU.

We evaluated the systems with four models: GAT [58], GCN [37], APPNP [38] and R-GCN [53], since they are widely adopted in academia and industry. For all models, we used their default configurations in DGL's implementation. We created unit tests for each model to verify the correctness of generated kernels by making sure that they produced the same results as DGL does. For all the models, we trained them for 200 epochs and report the average training time per epoch. We discarded the time of first 3 epochs for GPU to warm up. We also report the peak memory usage of the systems.

Table 2. Datasets

Dataset	#vertices	#edges	#feature	#relation
cora [54]	2,709	10,556	1,433	1
citeseer [54]	3,328	9,228	3,703	1
pubmed [49]	19,718	88,651	500	1
coraFull [8]	19,794	130,622	8,710	1
ca_cs [29]	18,334	327,576	6,805	1
ca_physics [29]	34,494	991,848	8,415	1
amz_photo [46]	7,651	287,326	745	1
amz_comp [46]	13,753	574,418	767	1
reddit [28]	198,021	84,120,742	602	1
aifb [53]	8,285	58,086	-	90
mutag [53]	23,644	148,454	-	46
bgs [53]	333,845	1,832,398	-	206

Table 2 lists some statistics of the datasets used in the experiments. GAT, GCN and APPNP are homogeneous models and we trained them using the first 9 datasets (with #relation to be 1). For RGCN, we used the last 3 heterogeneous datasets. We used 85% of the vertices in the reddit graph so that at least one system did not run OOM.

7.1 Comparison with Existing Systems

Per-epoch time. We report the per-epoch time of Seastar, DGL and PyG for the homogeneous models in Figure 10. Seastar outperforms DGL and PyG on all the datasets. Compared with DGL, Seastar achieves up to 14 times speed-up. Compared with PyG, the speed-up is up to 3 times. The performance gain comes from two aspects. First, by applying the Seastar fusion, intermediate results can be directly pipelined to downstream operators without dumping into GPU memory, which leads to better data locality. Second, the kernel design in §6 allows us to achieve much better performance compared with DGL when conducting graph convolution operations. Seastar performs better when a dataset has higher average vertex degree (e.g., amz_comp and reddit), thanks to the efficient locality-centric aggregation. In addition, Seastar also achieves better performance when models are more complicated (e.g., GAT) because there are more opportunities of applying Seastar operator fusion.

Memory consumption. We report the memory consumption in Figure 11. We omit the results for the small datasets as for them the peak memory usage is mostly below 100MB. Seastar is the only system that can run all the three GNN models on all datasets. Our materialization planning optimizes all intermediate results within execution units. Most of the intermediate results are resided on edges, whose memory consumption can be large when a graph consists of many edges. In contrast, PyG has a much higher peak memory usage and fails to run any of the models on the reddit dataset. This is due to the fact that it explicitly materializes edge-wise tensors. DGL has similar peak memory usage as Seastar since it can avoid producing large edge-wise tensors using

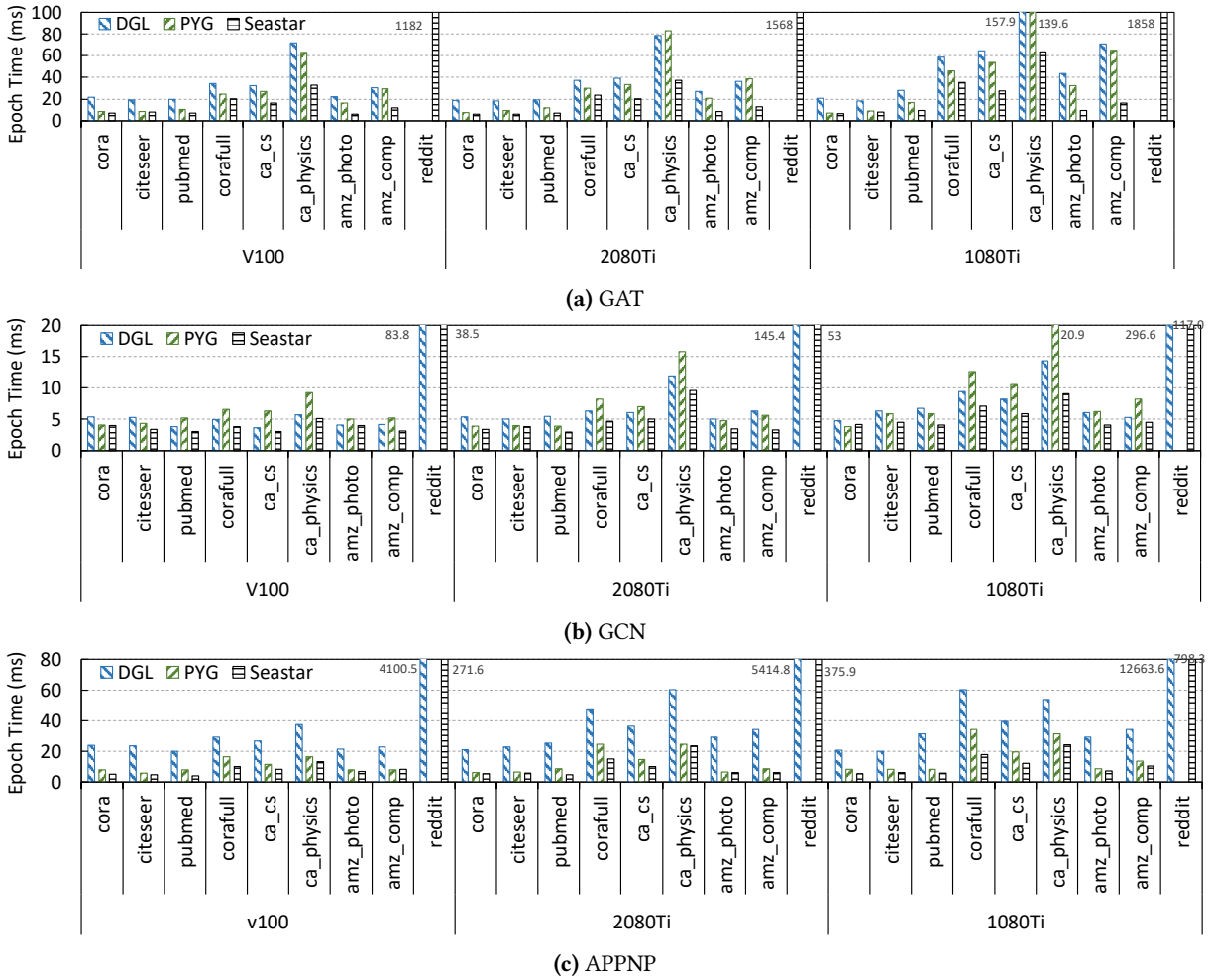

Figure 10. Per-epoch time (ms) of training homogeneous GNN models using DGL, PyG, and Seastar

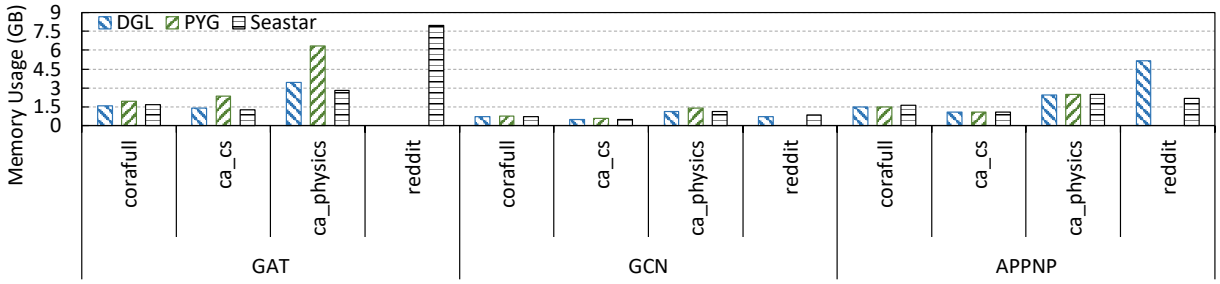

Figure 11. Peak memory consumption (GB) of training homogeneous GNN models using DGL, PyG, and Seastar

the *binaryReduce* primitive. But Seastar uses only 40% of the memory used by DGL for training APPNP on reddit.

Training heterogeneous model. DGL and PyG have two versions of implementations for RGCN. The first version of both systems uses their native heterogeneous programming model, which processes different edge types sequentially and then aggregates among edge types. The second version is manually optimized using batched matrix multiplication (bmm), denoted as **DGL-bmm** and **PyG-bmm**. We report the results of training RGCN in Tables 3 and 4. Seastar is orders of magnitude faster than PyG and DGL. Even compared with the manually optimized DGL-bmm and PyG-bmm, Seastar can still be 2.3 and 3.4 times faster, respectively. Both PyG and PyG-bmm ran OOM on the bgs dataset as they explicitly materialize edge-wise tensors. Seastar is

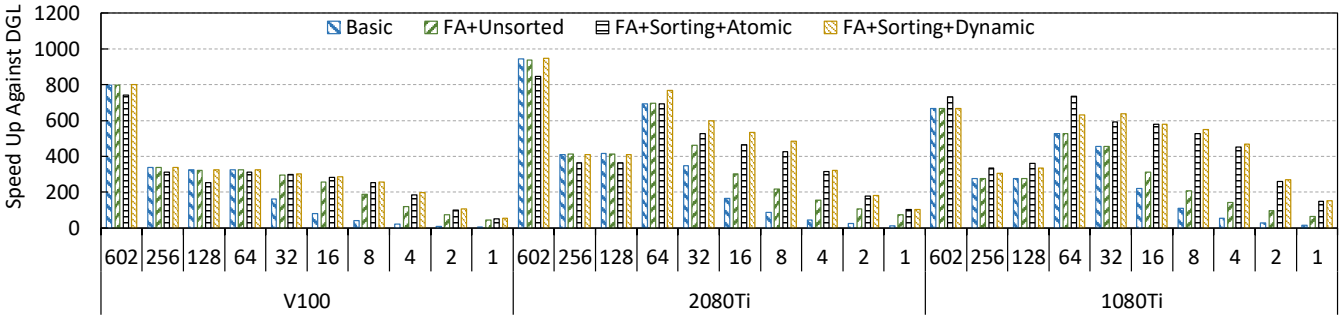

Figure 12. Neighbor access speed-up compared with DGL

Table 3. Per-epoch time (ms) of training RGCN

Dataset	GPU	Seastar	PyG-bmm	PyG	DGL-bmm	DGL
aifb	V100	**3.1**	10.8	106.2	7.3	624.5
	2080Ti	**3.8**	11.4	122.2	8.7	646.8
	1080Ti	**5.2**	10.6	211.2	14.8	932.7
mutag	V100	**8.4**	23.2	63.3	8.4	334.9
	2080Ti	**8.5**	22.8	72.8	10.5	350.4
	1080Ti	**13.3**	15.9	136.0	31.4	858.2
bgs	V100	141.7	-	-	**104.3**	-
	2080Ti	**138.1**	-	-	144.2	-
	1080Ti	**289.9**	-	-	309.6	-

Table 4. Peak memory consumption (MB) of training RGCN

Dataset	Seastar	PyG-bmm	PyG	DGL-bmm	DGL
aifb	148	318	323	**146**	197
mutag	361	475	483	354	**282**
bgs	8262	-	-	**8159**	-

slower than DGL-bmm on the bgs dataset using V100 GPU, because DGL-bmm was implemented using cuBLAS which has been optimized by leveraging architecture-specific designs (e.g., tensor cores in V100). Currently, Seastar has not implemented such optimizations. In terms of peak memory usage, only DGL-bmm and Seastar finished training on bgs.

7.2 The Effects of Various Designs

Next we evaluate our kernel-level designs in §6 by designing a neighbor access benchmark. The benchmark tests the time taken to access the neighbors' features for all vertices, which is a necessary step for GNN training. The benchmark is intentionally designed to be simple in order to reveal the bottleneck in kernel designs. For this experiment, we used reddit as the evaluation dataset. We used reddit's original feature and created synthetic features. We set synthetic features' sizes to the exponent of 2, ranging from 2^1 to 2^8.

We use DGL's binary-search based approach as the **baseline**. We denote the *vertex-parallel edge-sequential* execution as **Basic**. For Basic, we fix its block size to 256 and assign a single vertex to each block. The second alternative is **FA+Unsorted**, which changes the static configuration of blocks to feature-adaptive. Then, **FA+Sorting+Atomic** uses atomic instructions to ensure that earlier launched blocks are assigned with high-degree vertices. **FA+Sorting+Dynamic** removes the atomic instructions and uses built-in block id.

Figure 12 presents the speed-up ratio with respect to the baseline. The result shows that our designs outperform the baseline by a large margin (up to 946 times faster). When the feature size is 602 (i.e., the original feature size), the baseline takes 1.57 sec while FA only takes 2.36 msec on 1080 Ti. When the feature size is 1, the baseline takes 5.92 msec while FA+Sorting+Dynamic only takes 38.5 μsec. Profiling shows that binary-search-based approach executes 830 times more instructions than the Basic version.

FA+Unsorted improves upon Basic by incorporating the feature-adaptive optimization. When the feature size is less than 64, FA+Unsorted achieves up to 8.2 times speedup compared with Basic. When the feature size is large, FA+Unsorted has similar performance with Basic since there is enough feature-level parallelism for Basic to exploit. FA+Sorting+Atomic improves upon feature-adaptive optimization by sorting vertices according to their degrees and relying on atomic instructions to achieve load balancing. But FA+Sorting+Atomic can sometimes be slower than FA+Unsorted due to the overhead of atomic instructions. We observe that when the feature size is large (over 64), sorting does not improve much over FA+Unsorted. This is because the vertex degree distribution in the reddit dataset is not skewed enough. Thus, even if the high-degree vertices are processed late, they will not become serious stragglers. Another reason is that in the original dataset, high-degree vertices are distributed rather randomly. Thus, they may have the chance of being processed earlier even if the vertices are not sorted by their degrees. Sorting leads to more benefits when the feature size is small. When vertices are sorted by their degrees, FAT groups in the same block have more balanced loads than when vertices are not sorted. FA+Sorting+Dynamic achieves better performance compared with using atomic instructions in all cases, indicating that dynamic block ids are indeed useful.

8 Related Work

GNN frameworks. DGL [59] is one of the most popular GNN frameworks. DGL proposes a message-passing style programming model, and it optimizes training by fusing some commonly used operators and executes the fused operator by exploiting edge-wise parallelism. In comparison, Seastar adopts a vertex-centric programming model and demonstrates better performance thanks to its kernel-level optimizations presented in §6. FeatGraph [31] is an acceleration engine targeted for GNN systems such as DGL. Implementing new GNN operators for the FeatGraph's kernel templates requires programming in low-level TVM primitives. In contrast, Seastar allows users to program UDFs in native Python expressions and automatically generates a high-performance kernel for both forward and backward training. In terms of kernel design, Seastar proposes novel designs such as FAT group, locality-centric execution and dynamic load balancing to maximize data locality and improve load balancing, while FeatGraph puts more focus on the efficient execution of feature-wise computation through partitioning and tiling. Roc [33] and NueGraph [44] mainly focus on efficient training of GNNs on very large graphs using several GPUs. Their techniques can be combined with Seastar to support distributed training. Euler [3] and AliGraph [68] scale GNN training to large graphs using sampling-based training. Seastar can be used as their GNN training engine to improve their single GPU training performance. For distributed GNN training, we also developed an efficient distributed graph communication library (DGCL) [11], which enables us to run Seastar on distributed GPUs for scalable GNN training.

Graph computing systems. Many graph computing systems [4, 13, 14, 27, 34, 45, 63] and graph database systems [1, 12, 15, 21] have been developed in recent years. These systems are mainly based on CPUs and their designs are very different from Seastar. Existing GPU-based graph computing systems optimize graph workloads such as breadth first search [47], betweenness centrality [9] and single source shortest path [20] on GPUs. General purpose GPU graph processing engines provide different types of programming models. For example, MapGraph [24] and Cusha [36] adopt the vertex-centric programming model for GPU graph processing. Medusa [66] proposes the Edge-Message-Vertex programming model. Gunrock [61] divides graph processing into advance-compute-filter. In comparison, Seastar does not introduce stages explicitly and enables users to use idiomatic python syntax to program GNN models.

To handle the irregularity of graph data, the CTA+Warp+Scan scheme [24, 43, 47] partitions vertices into three categories, larger, medium and small, based on their degrees and vertex arbitrates for processing using blocks, warps and threads, respectively. Gunrock [61] switches between this scheme and the load-balancing approach in [20] according to a graph's structure. Groute [6] designs asynchronous multi-GPU constructs to speed up multi-GPU processing. XBFS [25] leverages various runtime optimizations and uses dynamic scheduling using atomics. CuSha [36] proposes G-Shards and Concatenated Windows (CW) to allow fully coalesced memory access and concurrent execution of different shards. Tigr [52] proposes to transform the unbalanced edge distribution into regular one by vertex splitting. Digraph [55] exploits the idea of applying GPU graph processing on a compressed graph to address the bottleneck in memory access. Contrary to these graph workloads, GNN training is iterative and vertex features are dense vectors. Based on these observations, Seastar proposes feature-adaptive thread mapping, degree sorting and dynamic load balancing to train GNNs efficiently.

Deep learning compilers. Many DL compilers have been proposed in recent years [5, 17, 19, 39, 51, 57, 62]. TVM [17] combines graph-level optimization and automatic operator-level optimization to generate kernels that are comparable with highly optimized libraries such as cuDNN [18]. Tensor-Comprehension [57] and Tiramisu [5] represent a computational graph using the polyhedral model, which supports various combinations of affine transformation such as tiling, fusion and shifting. Seastar is different from these works in the following aspects. First, Seastar provides a user-friendly vertex-centric programming model for programming GNNs. Existing DL compilers usually use the computational graph generated from popular DL frameworks and do not directly interact with model developers. Second, for the code generation process, Seastar introduces graph type into a computational graph to make sure that the optimizations work correctly. Third, Seastar designs high-performance template for GNN workloads by leveraging domain knowledge of GNN training. The techniques we propose may also be adopted in DL compilers.

9 Conclusions

We presented the design of a novel GNN training framework, called Seastar. Seastar offers a vertex-centric programming model so that users can focus on the logic of a single vertex and program GNNs with idiomatic Python syntax. Seastar identifies abundant operator fusion opportunities in the computational graphs of GNN training. With novel designs such feature-adaptive groups, locality-centric execution and dynamic load balancing, Seastar generates high-performance fused kernels for forward and backward passes. Seastar achieves significant performance improvements over popular systems such as DGL and PyG.

Acknowledgments. We thank the reviewers and the shepherd of our paper, Marco Serafini, for their constructive comments that have helped greatly improve the quality of the paper. This work was partially supported by GRF 14208318 from the RGC of HKSAR.

References

[1] 2018. *Neo4J*. https://neo4j.com/.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*, Kimberly Keeton and Timothy Roscoe (Eds.). USENIX Association, 265–283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[3] Alibaba. 2020. Euler. https://github.com/alibaba/euler.

[4] Ching Avery. 2011. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara* 11 (2011).

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, Mahmut Taylan Kandemir, Alexandra Jimborean, and Tipp Moseley (Eds.). IEEE, 193–205. https://doi.org/10.1109/CGO.2019.8661197

[6] Tal Ben-Nun, Michael Sutton, Sreepathi Pai, and Keshav Pingali. 2017. Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, Vivek Sarkar and Lawrence Rauchwerger (Eds.). ACM, 235–248. http://dl.acm.org/citation.cfm?id=3018756

[7] Robert D. Blumofe. 1994. Scheduling Multithreaded Computations by Work Stealing. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 356–368. https://doi.org/10.1109/SFCS.1994.365680

[8] Aleksandar Bojchevski and Stephan Günnemann. 2018. Deep Gaussian Embedding of Graphs: Unsupervised Inductive Learning via Ranking. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=r1ZdKJ-0W

[9] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality. *The Journal of Mathematical Sociology* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249 arXiv:https://doi.org/10.1080/0022250X.2001.9990249

[10] F. Warren Burton and M. Ronan Sleep. 1981. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 conference on Functional programming languages and computer architecture, FPCA 1981, Wentworth, New Hampshire, USA, October 1981*, Arvind and Jack B. Dennis (Eds.). ACM, 187–194. https://doi.org/10.1145/800223.806778

[11] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. 2021. DGCL: An Efficient Communication Library for Distributed GNN Training. In *Proceedings of the Fourteenth EuroSys Conference 2021, April 26-28, 2021*. ACM.

[12] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 87–100. https://doi.org/10.1145/3357223.3362715

[13] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an efficient task-oriented graph mining system. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, Rui Oliveira, Pascal Felber, and Y. Charlie Hu (Eds.). ACM, 32:1–32:12. https://doi.org/10.1145/3190508.3190545

[14] Hongzhi Chen, Xiaoxi Wang, Chenghuan Huang, Juncheng Fang, Yifan Hou, Changji Li, and James Cheng. 2019. Large Scale Graph Mining with G-Miner. In *SIGMOD*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1881–1884. https://doi.org/10.1145/3299869.3320219

[15] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. 2020. High Performance Distributed OLAP on Property Graphs with Grasper. In *SIGMOD*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 2705–2708. https://doi.org/10.1145/3318464.3384685

[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

[18] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[19] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, William Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar Vijay, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. 2018. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. *CoRR* abs/1801.08058 (2018). arXiv:1801.08058 http://arxiv.org/abs/1801.08058

[20] Andrew A. Davidson, Sean Baxter, Michael Garland, and John D. Owens. 2014. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*. IEEE Computer Society, 349–359. https://doi.org/10.1109/IPDPS.2014.45

[21] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[22] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[23] Alex Fout, Jonathon Byrd, Basir Shariat, and Asa Ben-Hur. 2017. Protein Interface Prediction using Graph Convolutional Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 6530–6539. http://papers.nips.cc/paper/7231-protein-interface-prediction-using-graph-convolutional-networks

[24] Zhisong Fu, Bryan B. Thompson, and Michael Personick. 2014. MapGraph: A High Level API for Fast Development of High Performance Graph Analytics on GPUs. In *Second International Workshop on Graph Data Management Experiences and Systems, GRADES 2014, co-loated with SIGMOD/PODS 2014, Snowbird, Utah, USA, June 22, 2014*, Peter A. Boncz and Josep-Lluís Larriba-Pey (Eds.). CWI/ACM, 2:1–2:6. https://doi.org/10.1145/2621934.2621936

[25] Anil Gaihre, Zhenlin Wu, Fan Yao, and Hang Liu. 2019. XBFS: EXploring Runtime Optimizations for Breadth-First Search on GPUs. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing* (Phoenix, AZ, USA) *(HPDC '19)*. Association for Computing Machinery, New York, NY, USA, 121–131. https://doi.org/10.1145/3307681.3326606

[26] 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 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[27] 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 2012, Hollywood, CA, USA, October 8-10, 2012*. 17–30. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

[28] William L. Hamilton, Justine Zhang, Cristian Danescu-Niculescu-Mizil, Dan Jurafsky, and Jure Leskovec. 2017. Loyalty in Online Communities. In *Proceedings of the Eleventh International Conference on Web and Social Media, ICWSM 2017, Montréal, Québec, Canada, May 15-18, 2017*. AAAI Press, 540–543. https://aaai.org/ocs/index.php/ICWSM/ICWSM17/paper/view/15710

[29] Drahomira Herrmannova and Petr Knoth. 2016. An Analysis of the Microsoft Academic Graph. *D Lib Mag.* 22, 9/10 (2016). https://doi.org/10.1045/september2016-herrmannova

[30] Huiting Hong, Hantao Guo, Yucheng Lin, Xiaoqing Yang, Zang Li, and Jieping Ye. 2020. An Attention-Based Graph Neural Network for Heterogeneous Structural Learning. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 4132–4139. https://aaai.org/ojs/index.php/AAAI/article/view/5833

[31] 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. *CoRR* abs/2008.11359 (2020). arXiv:2008.11359 https://arxiv.org/abs/2008.11359

[32] Huawei. 2020. MindSpore. https://e.huawei.com/us/products/cloud-computing-dc/atlas/mindspore.

[33] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*, Inderjit S. Dhillon, Dimitris S. Papailiopoulos, and Vivienne Sze (Eds.). mlsys.org. https://proceedings.mlsys.org/book/300.pdf

[34] Tatiana Jin, Zhenkun Cai, Boyang Li, Chengguang Zheng, Guanxian Jiang, and James Cheng. 2020. Improving resource utilization by timely fine-grained scheduling. In *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020*, Angelos Bilas, Kostas Magoutis, Evangelos P. Markatos, Dejan Kostic, and Margo I. Seltzer (Eds.). ACM, 20:1–20:16. https://doi.org/10.1145/3342195.3387551

[35] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar R. Iyer, and Chita R. Das. 2013. OWL: cooperative thread array aware scheduling techniques for improving GPGPU performance. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*, Vivek Sarkar and Rastislav Bodík (Eds.). ACM, 395–406. https://doi.org/10.1145/2451116.2451158

[36] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *The 23rd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'14, Vancouver, BC, Canada - June 23 - 27, 2014*, Beth Plale, Matei Ripeanu, Franck Cappello, and Dongyan Xu (Eds.). ACM, 239–252. https://doi.org/10.1145/2600212.2600227

[37] Thomas N. Kipf and Max Welling. 2017. 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. https://openreview.net/forum?id=SJU4ayYgl

[38] Johannes Klicpera, Aleksandar Bojchevski, and Stephan Günnemann. 2019. Predict then Propagate: Graph Neural Networks meet Personalized PageRank. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=H1gL-2A9Ym

[39] Chris Leary and Todd Wang. 2017. TensorFlow XLA, TensorFlow Dev Summit 2017.

[40] Minseok Lee, Seokwoo Song, Joosik Moon, John Kim, Woong Seo, Yeon-Gon Cho, and Soojung Ryu. 2014. Improving GPGPU resource utilization through alternative thread block scheduling. In *20th IEEE International Symposium on High Performance Computer Architecture, HPCA 2014, Orlando, FL, USA, February 15-19, 2014*. IEEE Computer Society, 260–271. https://doi.org/10.1109/HPCA.2014.6835937

[41] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, and Depei Qian. 2020. The Deep Learning Compiler: A Comprehensive Survey. *CoRR* abs/2002.03794 (2020). arXiv:2002.03794 https://arxiv.org/abs/2002.03794

[42] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. 2018. Combinatorial Optimization with Graph Convolutional Networks and Guided Tree Search. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 537–546. http://papers.nips.cc/paper/7335-combinatorial-optimization-with-graph-convolutional-networks-and-guided-tree-search

[43] Hang Liu and H. Howie Huang. 2019. SIMD-X: Programming and Processing of Graph Algorithms on GPUs. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 411–428. https://www.usenix.org/conference/atc19/presentation/liu-hang

[44] 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 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 443–458. https://www.usenix.org/conference/atc19/presentation/ma

[45] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146. https://doi.org/10.1145/1807167.1807184

[46] Julian J. McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015*, Ricardo Baeza-Yates, Mounia Lalmas, Alistair Moffat, and Berthier A. Ribeiro-Neto (Eds.). ACM, 43–52. https://doi.org/10.1145/2766462.2767755

[47] Duane Merrill, Michael Garland, and Andrew S. Grimshaw. 2012. Scalable GPU graph traversal. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, J. Ramanujam and P. Sadayappan (Eds.). ACM, 117–128. https://doi.org/10.1145/2145816.2145832

[48] Federico Monti, Michael M. Bronstein, and Xavier Bresson. 2017. Geometric Matrix Completion with Recurrent Multi-Graph Neural Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 3697–3707. http://papers.nips.cc/paper/6960-geometric-matrix-completion-with-recurrent-multi-graph-neural-networks

[49] Galileo Namata, Ben London, Lise Getoor, and Bert Huang. 2012. Query-driven Active Surveying for Collective Classification. In *10th International Workshop on Mining and Learning with Graphs*, Vol. 8.

[50] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy andF Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

[51] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018). arXiv:1805.00907 http://arxiv.org/abs/1805.00907

[52] Amir Hossein Nodehi Sabet, Junqiao Qiu, and Zhijia Zhao. 2018. Tigr: Transforming Irregular Graphs for GPU-Friendly Graph Processing. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2018, Williamsburg, VA, USA, March 24-28, 2018*, Xipeng Shen, James Tuck, Ricardo Bianchini, and Vivek Sarkar (Eds.). ACM, 622–636. https://doi.org/10.1145/3173162.3173180

[53] Michael Sejr Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling Relational Data with Graph Convolutional Networks. In *The Semantic Web - 15th International Conference, ESWC 2018, Heraklion, Crete, Greece, June 3-7, 2018, Proceedings (Lecture Notes in Computer Science)*, Aldo Gangemi, Roberto Navigli, Maria-Esther Vidal, Pascal Hitzler, Raphaël Troncy, Laura Hollink, Anna Tordai, and Mehwish Alam (Eds.), Vol. 10843. Springer, 593–607. https://doi.org/10.1007/978-3-319-93417-4_38

[54] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. 2008. Collective Classification in Network Data. *AI Mag.* 29, 3 (2008), 93–106. https://doi.org/10.1609/aimag.v29i3.2157

[55] Mo Sha, Yuchen Li, and Kian-Lee Tan. 2019. GPU-based Graph Traversal on Compressed Graphs. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 775–792. https://doi.org/10.1145/3299869.3319871

[56] Sainbayar Sukhbaatar, Arthur Szlam, and Rob Fergus. 2016. Learning Multiagent Communication with Backpropagation. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 2244–2252. http://papers.nips.cc/paper/6398-learning-multiagent-communication-with-backpropagation

[57] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. *CoRR* abs/1802.04730 (2018). arXiv:1802.04730 http://arxiv.org/abs/1802.04730

[58] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2017. Graph Attention Networks. *CoRR* abs/1710.10903 (2017). arXiv:1710.10903 http://arxiv.org/abs/1710.10903

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

[60] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. 2019. Heterogeneous Graph Attention Network. 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, 2022–2032. https://doi.org/10.1145/3308558.3313562

[61] Yangzihao Wang, Andrew A. 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, PPoPP 2016, Barcelona, Spain, March 12-16, 2016*, Rafael Asenjo and Tim Harris (Eds.). ACM, 11:1–11:12. https://doi.org/10.1145/2851141.2851145

[62] Richard Wei, Lane Schwartz, and Vikram S. Adve. 2018. DLVM: A modern compiler infrastructure for deep learning systems. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HJxPq4ywz

[63] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A Block-Centric Framework for Distributed Computation on Real-World Graphs. *Proc. VLDB Endow.* 7, 14 (2014), 1981–1992. https://doi.org/10.14778/2733085.2733103

[64] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 974–983. https://doi.org/10.1145/3219819.3219890

[65] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V. Chawla. 2019. Heterogeneous Graph Neural Network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 793–803. https://doi.org/10.1145/3292500.3330961

[66] Jianlong Zhong and Bingsheng He. 2014. Medusa: Simplified Graph Processing on GPUs. *IEEE Trans. Parallel Distributed Syst.* 25, 6 (2014), 1543–1552. https://doi.org/10.1109/TPDS.2013.111

[67] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2018. Graph Neural Networks: A Review of Methods and Applications. *CoRR* abs/1812.08434 (2018). arXiv:1812.08434 http://arxiv.org/abs/1812.08434

[68] 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. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.