- AliGraph
  - 数据模型
  - 算法框架
  - 系统框架
    - 图存储
    - 图采样
    - 计算
- NeuGraph
  - 编程模型
  - 系统实现
    - Graph-Aware Dataflow Translation
    - Streaming Processing out of GPU core
    - Parallel Multi-GPU Processing
    - Propagation Engine
    - 其他技巧
  - 实验评估
- DGL
- Architectural Implications of Graph Neural Networks
- Characterizing and Understanding GCNs on GPU [Yan-2020]
- 参考文献

AliGraph

数据模型

AliGraph面向的数据模型为Attributed Heterogeneous Graph (AHG)。

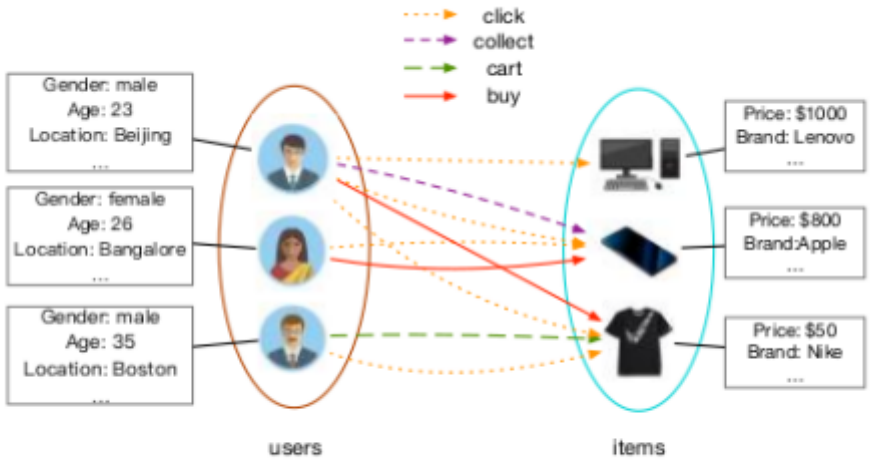

Figure 2: Illustrative example of AHG with multiple types of edges, nodes and rich attributes.

算法框架

AliGraph所支持的通用GNN框架。在该框架中，每一层的GNN被拆解为三个基本算子：Sample, Aggregate和Combine。其中Aggregate进行边计算，而Combine进行点计算。

Algorithm 1: GNN Framework

Input: network $\mathcal{G}$, embedding dimension $d \in \mathbb{N}$, a vertex feature $\mathbf{x}_v$ for each vertex $v \in \mathcal{V}$ and the maximum hops of neighbors $k_{max} \in \mathbb{N}$.
Output: embedding result $\mathbf{h}_v$ of each vertex $v \in \mathcal{V}$

$\mathbf{h}_v^{(0)} \leftarrow \mathbf{x}_v$
for $k \leftarrow 1$ *to* k_{max} **do**
  for *each vertex* $v \in \mathcal{V}$ **do**
    $S_v \leftarrow \text{SAMPLE}(Nb(v))$
    $\mathbf{h}'_v \leftarrow \text{AGGREGATE}(\mathbf{h}_u^{(k-1)}, \forall u \in S)$
    $\mathbf{h}_v^{(k)} \leftarrow \text{COMBINE}(\mathbf{h}_v^{(k-1)}, \mathbf{h}'_v)$
  normalize all embedding vectors $\mathbf{h}_v^{(k)}$ for all $v \in \mathcal{V}$
$\mathbf{h}_v \leftarrow \mathbf{h}_v^{(k_{max})}$ for all $v \in \mathcal{V}$ return $\mathbf{h}_v$ as the embedding result for all $v \in \mathcal{V}$

> 注意：在该算法框架中，每一层（hop）都采用了相同的Sample/Aggregate/Combine算子，其不允许不同层采用不同的算子进行组合。

系统框架

AliGraph的系统架构如下。最上面是Aggregate和Combine算子的实现，中间是Sampling方法，最下面是图存储。目前的AliGraph主要运行在CPU环境中。

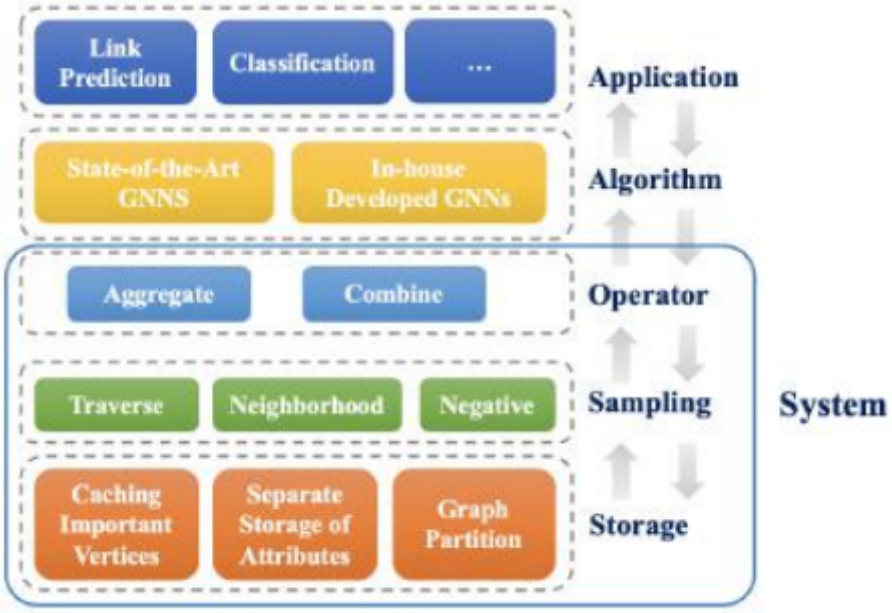

Figure 3: Architecture of the *AliGraph* system.

图存储

AliGraph采用的是vertex-cut的划分方案，即不同的边被分到不同的机器上。

Algorithm 2: Partition and Caching

Input: graph $\mathcal{G}$, partition number p, cache depth h, threshold $\tau_1, \tau_2, \ldots, \tau_h$
Output: p subgraphs

Initialize p graph servers
for *each edge* $e = (u, v) \in \mathcal{E}$ **do**
    $j = \text{ASSIGN}(u)$
    Send edge e to the j-th partition
for *each vertex* $v \in V$ **do**
    for $k \leftarrow 1$ *to* h **do**
        Compute $D_i^{(k)}(v)$ and $D_o^{(k)}(v)$
        if $\frac{D_i^{(k)}(v)}{D_o^{(k)}(v)} \geq \tau_k$ **then**
            Cache the 1 to k-hop out-neighbors of v on each partition where v exists

本文采用的是vertex-cut的划分方案

图中顶点和边的属性与图的邻接表**分开存储**，见图AliGraph-Fig4。原因有二：

1. 属性信息更占据存储空间。
2. 不同顶点/边的属性有很大的重叠。例如大量顶点都具有共同的标签，例如“男性”、“洗漱用品”等。

通过为顶点属性和边属性建立Index，将图的拓扑信息与图的属性信息建立关联。为了减少对属性信息的访问开销，在每台机器上会对Index中的属性条目建立cache，cache采用LRU替换策略。

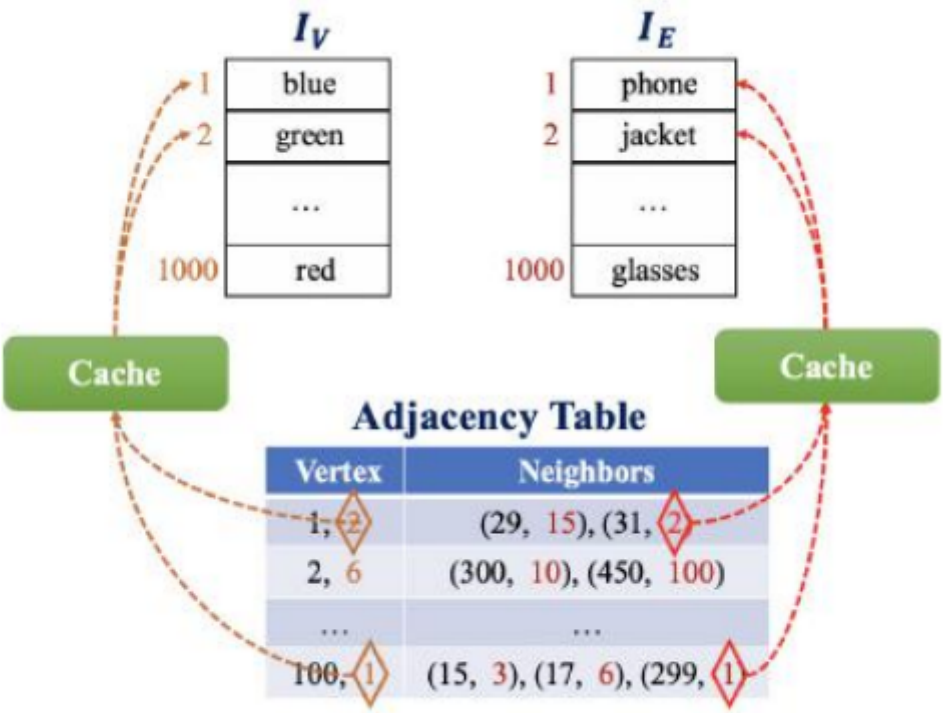

Figure 4: Index structure of graph storage.

同时，每台机器会缓存重要顶点的邻接表。采用如下的公式为每个顶点v，确定其k-重要性（k-th importance），其中$D_i^{(k)}(v)$和$D_o^{(k)}(v)$表示顶点v的k跳出/入邻域的大小。每台机器只缓存重要性大于阈值τ_k的顶点v的出边邻接表。实际实践表明考虑至多2跳邻域就足够了，阈值τ_k设置为0.2就效果很好。

$$Imp^{(k)}(v) = \frac{D_i^{(k)}(v)}{D_o^{(k)}(v)}$$

与重要性指标相关的一个定理是：如果顶点的度数服从幂率分布，则其k跳邻域的规模也服从幂率分布，从而k-重要性也服从*幂率分布*。

> 对于无向图这个指标无法适用。

实验表明因为Importance指标遵从Power-law分布，因此较低的threshold就能够cache足够数量的顶点。同时缓存替换策略的实验基于importance指标的cache策略比随机替换和LRU替换都有效，更适合图神经网络。Importance策略和随机替换策略都是静态策略，其会预先cache相应的顶点邻接表。而LRU策略因为其动态特性，会经常剔除、替换已经cache的邻接表，导致额外开销。

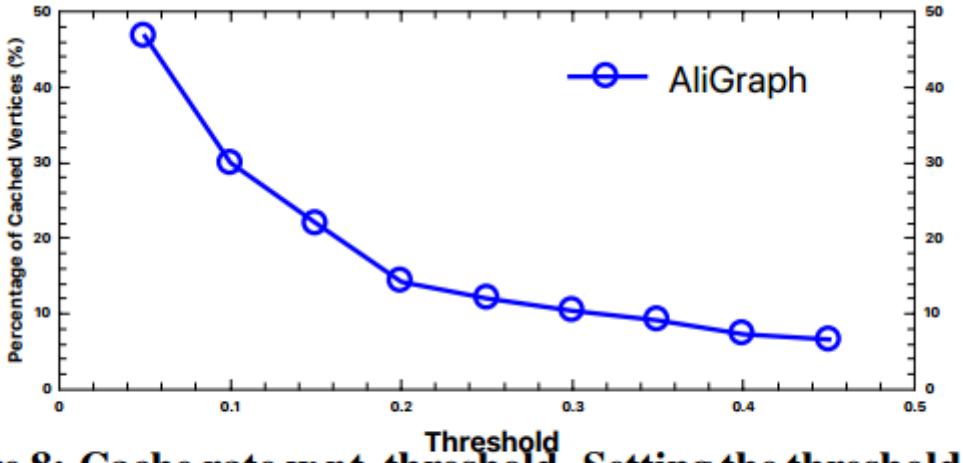

Figure 8: Cache rate w.r.t. threshold. Setting the threshold near 0.15 makes the best trade-off.

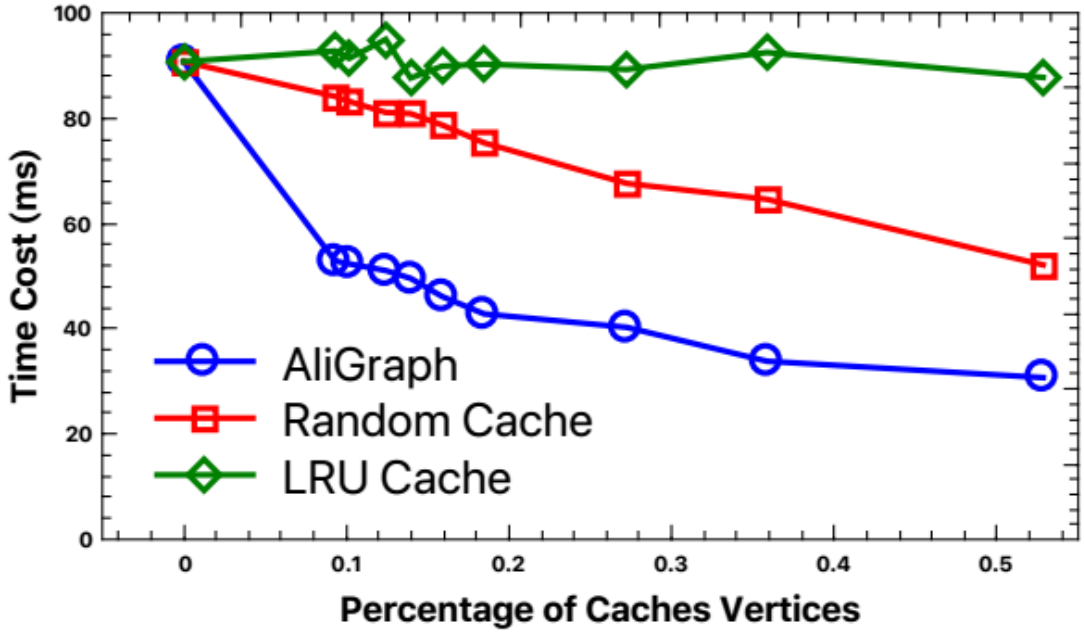

Figure 9: Cost time w.r.t. percentage of cached vertices. Our method saves much time than other cache strategies.

> 如何制定适合图分析的cache策略也是研究方向之一。

在实现时，将边按照source vertex划分成不同的组，每一个组绑定到一个core上。对于该组顶点邻接表的访问与更新操作被组织到一个request-flow桶中，该桶由**lock-free的队列实现**。

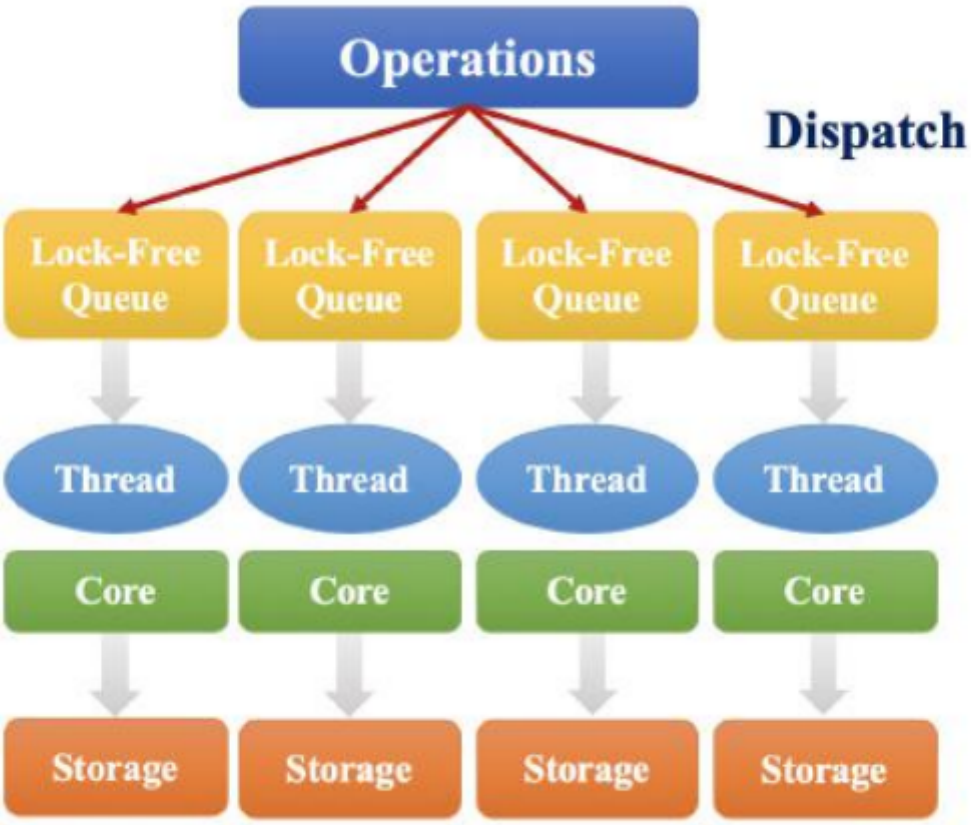

Figure 6: Lock-free graph operations

图采样

AliGraph中支持3种图采样策略，同时也运行以plugin的形式扩展。

1. Traverse：从一个图分区中采样一批顶点。可以直接从当前服务器的分区中生成。
2. Neighborhood：采样某个顶点的1跳或多跳邻域。如果顶点邻域跨服务器，则分批地从其他图服务器获取邻接表。
3. Negative：生成负采样样本，加速收敛。通常可在本地服务器上完成，按需地查询远程服务器以获得邻接表。

Sampler中的权重也允许根据梯度进行更新。

经过采样，每个顶点的邻域大小被对齐，使其可以很容易地被处理。

实验表明通过分布式采样（Batch size=512，Cache size=20%），即使是很大的图，也能非常快地采样完毕。

Neighborhood采样因为要涉及服务器之间的通讯，速度会比另外两个采样慢很多。

采样技术的性能对数据规模不敏感，及时图规模增大6倍，采样时间的变化也不大。

Table 4: Effects of optimized *Sampling*. All sampling methods can finish in no more than 60ms.

Dataset	Setting		Time (ms)		
	# of workers	Cache Rate	TRAVERSE	NEIGHBORHOOD	NEGATIVE
Taobao-small	25	18.46%	2.59	45.31	6.22
Taobao-large	100	17.68%	2.62	52.53	7.52

计算

编程模型中与计算相关的是Aggregate和Combine。AliGraph也允许以plugin的形式扩展实现这两个算子。需要注意的是，这两个算子需要同时实现其forward计算和backward计算的逻辑。

在计算的过程中，会保存每个顶点v在当前mini-batch中的最新的中间特征向量：$h_v^{(1)}, \ldots, h_v^{(kmax)}$。

实验表明cache mini-batch的中间特征向量对于提升两个算子的计算速度非常重要。

Table 5: Effects of optimized *Operators* with an order of magnitude of time speed up.

Dataset	W/O Our Implementation (ms)	Our Implementation (ms)	Speedup Ratio
Taobao-small	7.33	**0.57**	12.9
Taobao-large	17.21	**1.26**	13.7

NeuGraph

NeuGraph是微软亚洲研究院提出的面向单机多GPU环境的并行图神经网络训练框架，该框架基于TensorFlow实现。NeuGraph主要面相transductive的setting，即整个图参与训练，在目前的系统中没有考虑sample，但作者说可以集成sample到框架中。

编程模型

NeuGraph为图神经网络训练提出了SAGA-NN（Scatter-ApplyEdge-Gather-ApplyVertex with Neural Networks）编程模型。SAGA-NN模型将图神经网络中每一层的前向计算划分为4个阶段：Scatter、ApplyEdge、Gather和ApplyVertex，如Figure 2所示。其中ApplyEdge和ApplyVertex阶段执行用户提供的基于神经网络的边特征向量和点特征向量的计算。Scatter和Gather是由NeuGraph系统隐式触发的阶段，这两个阶段为ApplyEdge和ApplyVertex阶段准备数据。

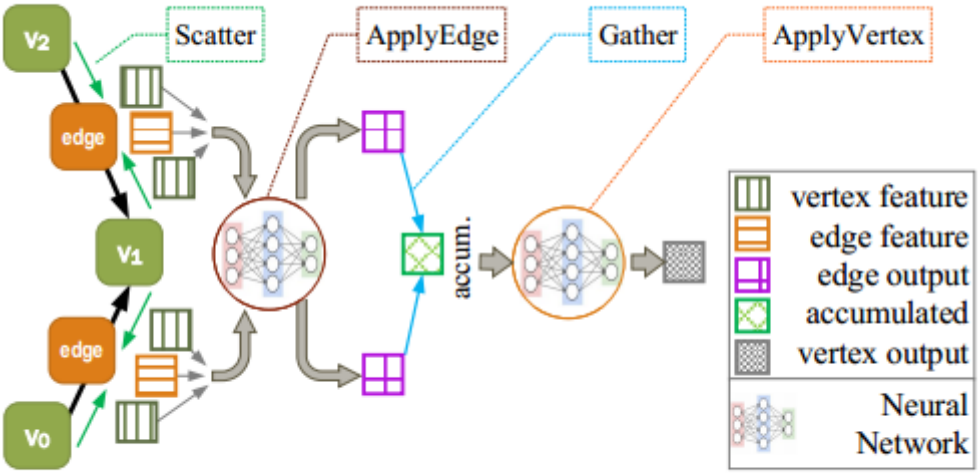

Figure 2: SAGA-NN stages for each layer of GNN.

在编程时，用户只需利用给定的算子实现ApplyEdge和ApplyVertex函数，并指定Gather方式，即可利用NeuGraph自动地完成GNN的训练。Figure 3展示了利用SAGA-NN编程模型表达Gated-GCN的编程示例。

```
G-GCN(vertex^ℓ):  // computing vertex^{ℓ+1}
  params p = [W_H^ℓ  W_C^ℓ  W^ℓ]
  // Passing data over edges
  edge^ℓ=Scatter(vertex^ℓ)
  // edge-parallel computation
  acc = ApplyEdge(edge^ℓ, p):
    η = sigmoid(p.W_H^ℓ ⊗ edge^ℓ.dest + p.W_C^ℓ ⊗ edge^ℓ.src)
    return η ⊙ edge^ℓ.src
  set Gather.accumulator = sum
  accum = Gather(acc)
  // compute new vertex data
  vertex^{ℓ+1} = ApplyVertex(vertex^ℓ, accum, p):
    return ReLU(p.W^ℓ ⊗ accum)
  return vertex^{ℓ+1}
```

Figure 3: Gated Graph ConvNet at layer ℓ in *SAGA-NN* model.

系统实现

Graph-Aware Dataflow Translation

NeuGraph采用2D图划分方法，其将顶点集划分为**P**个分区（trunk），边集（邻接矩阵）划分为$P \times P$个分区，其中边分区E_{ij}保存了连接点分区V_i和V_j的边。NeuGraph基于chunk构建数据流图，如Figure 5所示。

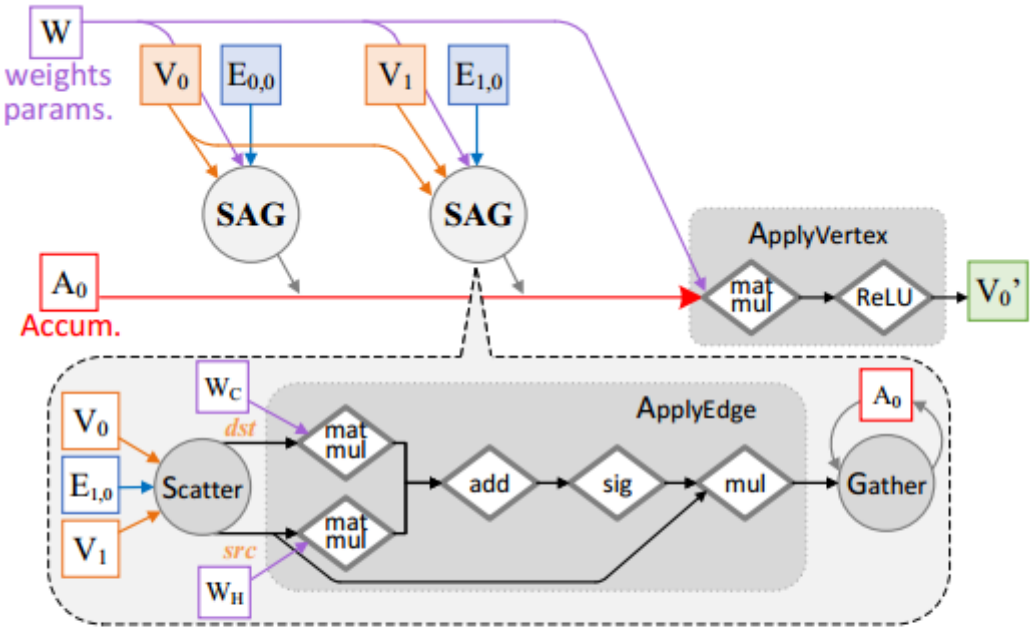

Figure 5: Chunk-based dataflow graph for a destination interval V_0 at a G-GCN layer. The backward dataflow graph and the swapping of intermediate results to host memory for backward are omitted for a clear visualization.

其中Scatter算子接收1个边分区和2个对应的点分区，将数据整理成[src, dst,data]的元组形式，该元组形式将传递给ApplyEdge函数进行处理。

在Forward阶段会产生大量中间计算结果（例如ApplyEdge中的矩阵乘法的结果），NeuGraph为了避免中间结果占用大量的GPU显存，其会将中间结果从GPU端显存上传到Host端的内存中，在Backward阶段再传回GPU端。

为了能在Gather阶段中复用数据，NeuGraph在Forward（Backward）阶段中采用列（行）优先的顺序处理边分区。例如为了在Forward阶段持续累加V0点分区中的点特征向量，其依次处理E(0,0), E(1,0), ..., E(n,0)边分区。

NeuGraph在不超过GPU显存容量限制的情况下选择尽可能小的分区数P。

Streaming Processing out of GPU core

为了能让GPU处理超过其显存容量的数据，必须将数据动态地在GPU和Host端进行交换。

Selective Scheduling技巧：一个边分区可能只与对应点分区中少量的点发生关联，因此在从CPU端向GPU端发送点分区数据时，可以只传递点分区中的少部分数据。NeuGraph根据CPU端内存拷贝的带宽、CPU-GPU端数据交换的带宽，动态地确定阈值，来确定是向GPU发送整个点分区，还是只发送点分区中的部分数据。

Pipeline Scheduling技巧：将一个边分区进一步划分为sub-trunk，流水线地向GPU发送sub-trunk并在GPU端并发地进行sub-trunk的计算。为了使计算和HtoD数据传输充分地重叠，NeuGraph采用一个基于profile的sub-trunk调度方案，其在头几轮迭代中profile各个sub-trunk的计算开销和数据传输开销，并根据开销计算出更优的调度方案，如Figure 6所示。

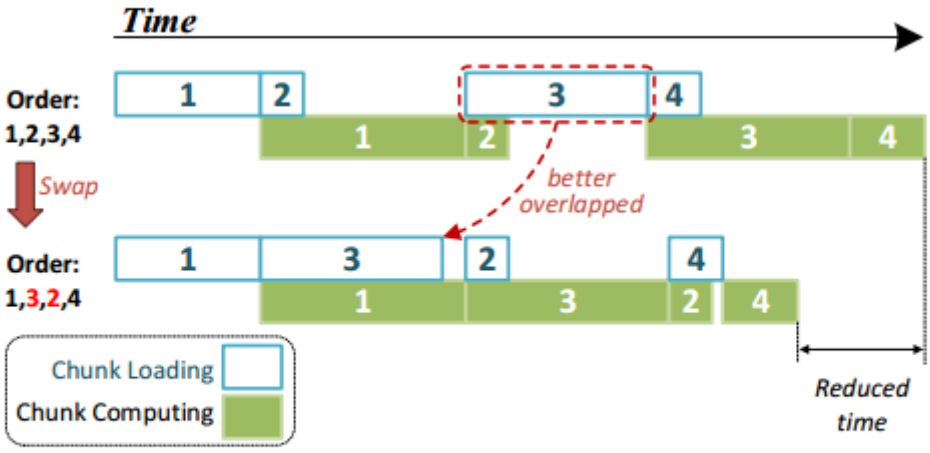

Figure 6: The swapping heuristic for a case of streaming two edge sub-chunks ($k = 2$).

Parallel Multi-GPU Processing

在拥有多GPU卡的环境中，可以充分利用各GPU卡之间的高速P2P PCIe通信来降低Host端PCIe总线带宽压力。NeuGraph将共享PCIe Switch的GPU卡视作一个虚拟GPU卡组，点分区、边分区的数据从Host memory中广播到各个虚拟GPU卡的第一个物理GPU中（例如Figure 8中的GPU0和GPU2）。第一个物理GPU在对该点分区进行处理的同时，并发地将数据发送给同一个虚拟GPU中的下一个物理GPU（例如Figure 8中的GPU1和GPU3），并发地从Host Device载入下一批Vertex Chunk和Edge Chunk数据。流水线地处理，直到所有点分区和边分区均处理完。

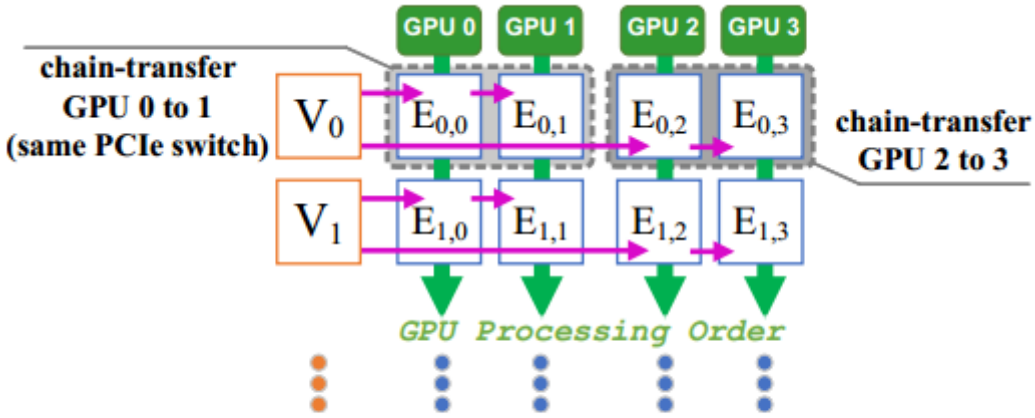

Figure 8: NeuGraph transfers vertex chunks along the chain.

Propagation Engine

NeuGraph在GPU上实现Graph Propagation时额外采用了如下优化手段：

- 在ApplyEdge中出现的只与source vertex或destination vertex相关的计算移动到上一步中的ApplyVertex中进行。这样避免对于每一条边都进行相应的计算。
- 在GPU上实现高效的Scatter和Gather kernel。
- Scatter-ApplyEdge-Gather算子融合：当ApplyEdge算子是element-wise的简单逻辑时，其整个SAG的过程被直接替换为Fused-Gather算子，该算子直接将source vertex和edge data读入到kernel的register中，并在register中完成ApplyEdge的计算，计算结果累加如destination vertex的accumulation向量。通过融合可以避免将中间计算结果保存回GPU显存，节省GPU内存访问开销。

其他技巧

- 模型参数每个GPU一份，通过all-reduce在各GPU之间保持同步。
- GPU之间通过P2P通信。

实验评估

- 实验在点分类任务上进行，我们可以借鉴论文中的叙述来说明。

(amazon) [30]. The column *feature* in Table 1 reports the sizes of the vertex feature vectors, and the *label* column contains the numbers of label classes. As different GNN tasks share the same GNN architecture and differ only on the output layer, we tested the performance of our system on the task of vertex classification (e.g., classifying academic papers into different subjects in the PubMed citation dataset, which contains sparse bag-of-words feature vectors for each document and a list of citation links between documents) and set the number of layers $\ell = 2$ in experiments.

- 数据集的平均度数影响Graph Propagation的时间开销，平均度数越高的数据集其Propagation的开销越高。
- 系统实验中的优化技巧是有效的，能够比单纯在TensorFlow上实现SAGA模型快2.4~4.9倍。
- “The results under other models are similar”这句表达可以借鉴。
- Selectively scheduling适合sparse graph而graph kernel optimization适合dense graph。
- 即使采用了IO（GPU与Host端数据交换）和GPU kernel互相重叠的优化，IO耗时依然长于GPU kernel计算耗时。

Time (s)	TF-SAGA			NeuGraph		
Dataset	IO	Comp.	Runtime	IO	Comp.	Runtime
reddit-full	7.67	13.27	20.94	3.84	2.46	4.28
enwiki	5.93	5.13	11.07	3.24	1.77	3.63
amazon	5.11	1.44	6.55	1.56	1.18	1.82

Table 2: GCN on large graphs: TF-SAGA vs. NeuGraph. NeuGraph overlaps I/O and computation time.

- Chain-based mechanism对于多GPU卡的扩展性至关重要。

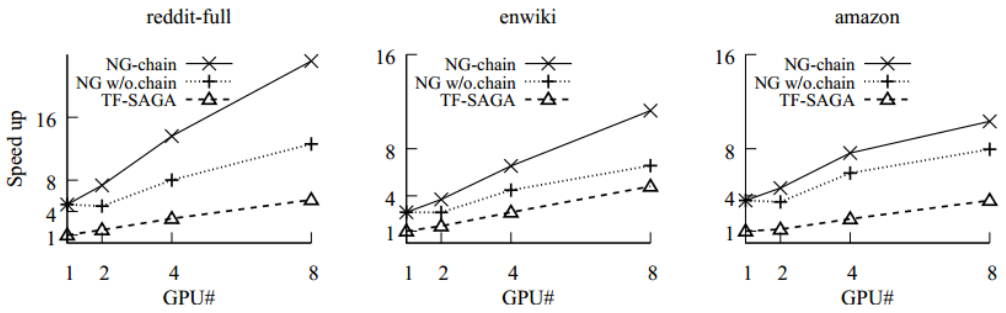

Figure 17: Scaling out GCN with NeuGraph on large graphs (w/o refers to without). The speedup is measured over the single GPU TF-SAGA (speedup = 1). Chain-base scheduling works on multi-GPU, resulting in the same 1 GPU point with it enabled/disabled.

- 当GPU卡数量上来时，被多GPU共享的CPU和Memory带宽将成为制约多GPU扩展性的瓶颈。
- NeuGraph的speedup曲线虽然是线性的，但距离理想的线性可扩展性还有差距。当其GPU卡数量从1增加到8时，其Speedup的增长远没有到8倍。

DGL

DGL是支持多后端（MXNet、TensorFlow、PyTorch）的一个面向深度图神经网络的计算框架。

DGL也采用基于message-passing的编程模型，用户提供自定义的message function（边计算）、update function（点计算）和reduce operation （消息规约操作）。但与PyG不同的是，DGL支持用户提供自定义的reduce operation，而不局限于sum、mean、max等少数几种。

DGL引用了文献(Xu-2018)表明aggregator（即reduce）所支持的复杂度与图神经网络的表达能力密切相关。

DGL支持sampling机制，该机制通过指定active vertex/set set实现，以data loader的形式提供给用户。

用户提供的message function和update function必须是向量化的，即可以对多个顶点同时操作。

kernel fusion是一个提高计算性能的有效技巧。

- kernel fusion将message function与update function合并进行，从而避免生成message的实体中间结果，从而大幅降低计算开销和GPU显存使用量。
- kernel fusion要求message function非常简单，不能使用任何参数。
- 实验表明DGL相比PyG的主要性能提升即来自kernel fusion。

Architectural Implications of Graph Neural Networks

GNN吸引人的一个优点是end-to-end的训练能力。

作者认为computation-intensive GEMM kernel不是GNN的性能热点。**这个结论要和我们的实验结果对照一下**。

本文关注**inference**阶段的性能热点。

如Fig. 3所示，实际GNN中用到的基本算子的种类是有限的，各算子经过组合得到丰富的GNN架构。

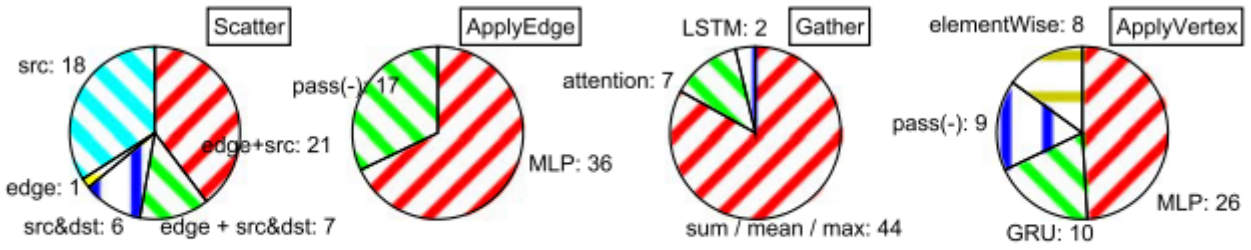

Fig. 3. The proportion of operations used in each stage of SAGA-NN.

DGL中允许Gather阶段采用任何的累加函数，包含LSTM，因此作者论文中覆盖了GraphSAGE-LSTM版本。

Tab 2中列出了实验中采用数据集情况。在列图数据集的情况后，可以把Graph Type也列上，如[Tab 2](#fig-aignn-tab2)所示。

TABLE 2
The Evaluated Datasets for the Vertex Classification Task

Name	Cora	Citeseer	Pubmed	AIFB	MUTAG	BGS
Vertex#	2,708	3,327	19,717	8,285	23,644	333,845
Edge#	5,429	4,732	44,338	29,043	74,227	916,199
Graph Type	Citation Network	Citation Network	Citation Network	Semantic Network	Molecular Structure	Geological Graph

> 作者选用的数据集平均度数有些低。
> 采用sampling技巧后，处理的图的平均度数也可能很低，需要结合实验。
> 我感觉图的平均度数可能会是影响性能的重要指标。

GPU硬件资源的利用率与图规模和隐向量的规模密切相关。

作者认为GNN没有固定的性能瓶颈，性能瓶颈会随着数据集和算法的不同而变化，因此各阶段都需要优化，都有优化的价值。

本文在处理GAT时，其ApplyEdge只有简单的矩阵向量乘法，而将耗时的softmax阶段算到Gather里，因此作者的实验结果中GAT的Gather阶段非常耗时，如Fig. 5所示。

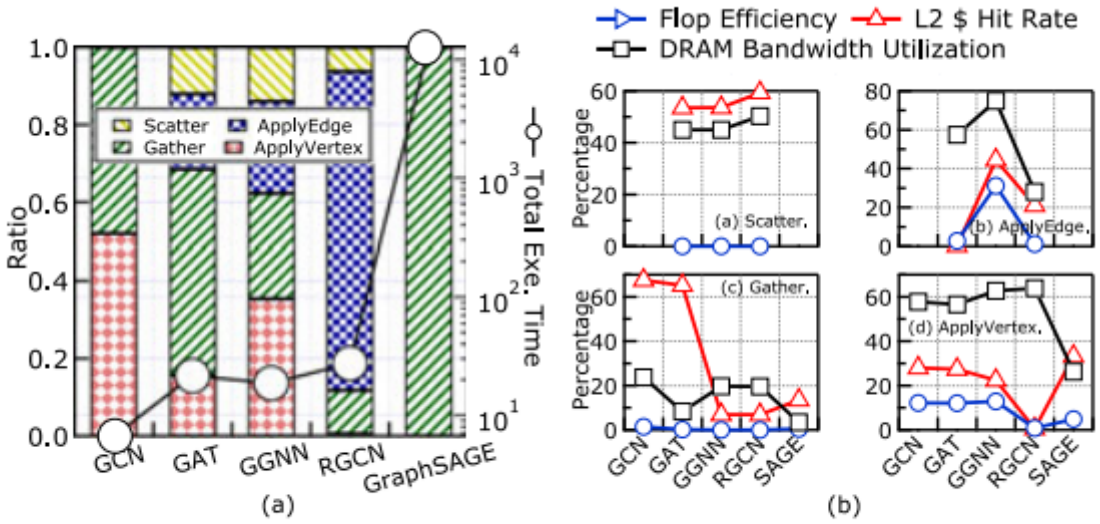

Fig. 5. (a) Stage time breakdown. (b) Stage execution statistics.

本文确认了Scatter阶段（对应于PyG的collect阶段）中只有数据拷贝，没有计算，并给出了该阶段的实现示意图。

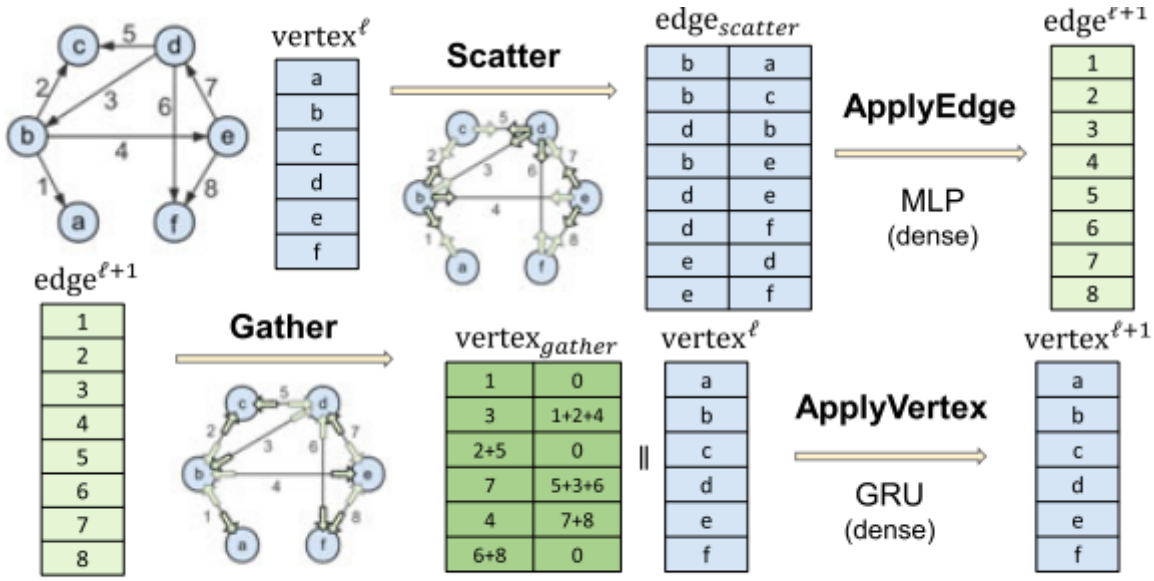

Fig. 2. The computation stages in a GGNN [6] layer.

相比传统的图分析计算PageRank、SCC等，因为GNN中每个顶点和边上都是向量，因此对于硬件cache来说locality比较好。

本文进一步验证了kernel fusion对于性能提升的重要性，如Fig.6所示。fused gattern kernel是由稀疏矩阵乘法实现，因此是可微的。

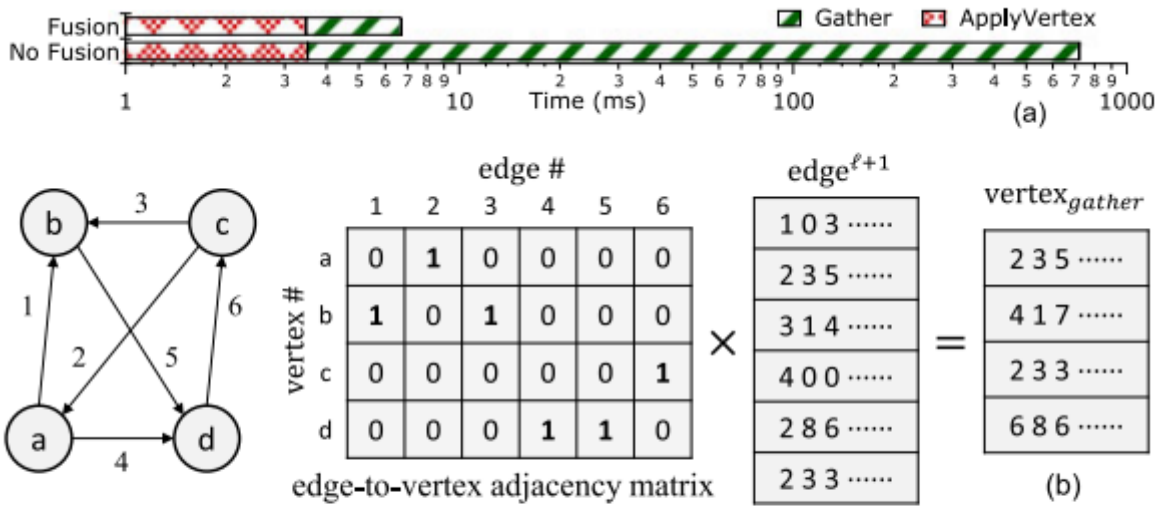

Fig. 6. (a) GCN time with/without fusion. (b) The fused `Gather` can be implemented by multiplying the graph adjacency matrix and edge embedding matrix.

本图很好的揭示了kernel fusion的实现原理。fusion后还是使用底层的线性代数算子实现，因此是可微的。

GNN相比传统DL的最大特点是引入了Sparse Matrix Operation。

文中Tab 3中总结了各阶段算子的Kernel和计算特性，这与我们的结论互相对照一下。

TABLE 3
The Characteristics of Different GNN Computation Stages

Stage	Description	Kernel
Scatter	Vertex/edge embedding movement	IndexSelection
ApplyEdge	DL-based edge embedding transformation	GEMM/GEMV
Gather (fused)	Edge embedding reduction	Sparse GEMM
Gather	Edge embedding movement	IndexSelection
(non-fused)	Complex reduction (e.g., LSTM)	GEMM/GEMV
ApplyVertex	DL based vertex embedding transformation	GEMM/GEMV

到时候我们可以将这张表的结论与我们观察到的现象对照一下。
另外该论文中是分阶段分析，我们也可以考虑。

Characterizing and Understanding GCNs on GPU [Yan-2020]

本文分析了GCN类的算法在inference阶段的特性，同时与经典的图分析算法（PageRank）和基于MLP的经典神经网络做了特性对比分析。

GCN顶点和边上的属性值是特征向量（维度至少为几十），而PageRank中顶点和边上的属性是标量。

- 特征向量带来了更加良好的locality（一个顶点的数据被连续的访问）。
- 特征向量带来了更高的顶点内的并行性。

本文利用sgemm实现GCN中特征向量（稀疏）与权重矩阵的乘法。

本文发现在GCN算法中的每一层$H = AXW$中，先计算$X' = XW$再计算$H = AX'$会带来更好的性能，因为X的纬度一般很高，而W的维度一般较低。

> 但是这样做会带来X也要参与W的梯度计算的问题，反而可能会得不偿失。

本文发现实际图中的顶点度数分布符合幂律分布的特性，因此缓存高度数的顶点，有可能可以提升硬件Cache的命中率。

因为aggregation阶段需要并发地、原子地更新顶点的输出特征向量，因此向量化原子访问有可能可以提升aggregation阶段的效率。

参考文献

1. [AliGraph]ZHU R, ZHAO K, YANG H, 等. AliGraph: A Comprehensive Graph Neural Network Platform[J]. Proceedings of the VLDB Endowment, 2019, 12(12): 2094–2105. DOI:10.14778/3352063.3352127.
2. [NeuGraph]MA L, YANG Z, MIAO Y, 等. NeuGraph: Parallel Deep Neural Network Computation on Large Graphs[C/OL]//2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA: USENIX Association, 2019: 443–458. https://www.usenix.org/conference/atc19/presentation/ma.
3. [DGL]WANG M, YU L, ZHENG D, 等. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs[J/OL]. arXiv:1909.01315 [cs, stat], 2019[2020–06–20]. http://arxiv.org/abs/1909.01315.
4. [Xu-2018]XU K, HU W, LESKOVEC J, 等. How Powerful are Graph Neural Networks?[C/OL]//7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. . https://openreview.net/forum?id=ryGs6iA5Km.
5. [Zhang-ICAL-2020]Z. ZHANG, J. LENG, L. MA, 等. Architectural Implications of Graph Neural Networks[J]. IEEE Computer Architecture Letters, 2020, 19(1): 59–62. DOI:10.1109/LCA.2020.2988991.
6. [Yan-2020]M. YAN, Z. CHEN, L. DENG, 等. Characterizing and Understanding GCNs on GPU[J]. IEEE Computer Architecture Letters, 2020, 19(1): 22–25. DOI:10.1109/LCA.2020.2970395.