CN 43-1258/TP
ISSN 1007-130X

计算机工程与科学
Computer Engineering & Science

第 46 卷第 1 期 2024 年 1 月
Vol. 46, No. 1, Jan. 2024

文章编号:1007-130X(2024)01-0001-11

GNNSched:面向 GPU 的图神经网络推理任务调度框架*

孙庆骁,刘　轶,杨海龙,王一晴,贾　婕,栾钟治,钱德沛
(北京航空航天大学计算机学院,北京 100191)

摘　要:由于频繁的显存访问,图神经网络 GNN 在 GPU 上运行时往往资源利用率较低。现有的推理框架由于没有考虑 GNN 输入的不规则性,直接适用到 GNN 进行推理任务共置时可能会超出显存容量导致任务失败。对于 GNN 推理任务,需要根据其输入特点预先分析并发任务的显存占用情况,以确保并发任务在 GPU 上的成功共置。此外,多租户场景提交的推理任务亟需灵活的调度策略,以满足并发推理任务的服务质量要求。为了解决上述问题,提出了 GNNSched,其在 GPU 上高效管理 GNN 推理任务的共置运行。具体来说,GNNSched 将并发推理任务组织为队列,并在算子粒度上根据成本函数估算每个任务的显存占用情况。GNNSched 实现了多种调度策略来生成任务组,这些任务组被迭代地提交到 GPU 并发执行。实验结果表明,GNNSched 能够满足并发 GNN 推理任务的服务质量并降低推理任务的响应时延。

关键词:图神经网络;图形处理器;推理框架;任务调度;估计模型
中图分类号:TP183　　**文献标志码**:A
doi:10.3969/j.issn.1007-130X.2024.01.001

GNNSched: A GNN inference task scheduling framework on GPU

SUN Qing-xiao, LIU Yi, YANG Hai-long, WANG Yi-qing, JIA Jie, LUAN Zhong-zhi, QIAN De-pei
(School of Computer Science and Engineering, Beihang University, Beijing 100191, China)

Abstract: Due to frequent memory access, graph neural network (GNN) often has low resource utilization when running on GPU. Existing inference frameworks, which do not consider the irregularity of GNN input, may exceed GPU memory capacity when directly applied to GNN inference tasks. For GNN inference tasks, it is necessary to pre-analyze the memory occupation of concurrent tasks based on their input characteristics to ensure successful co-location of concurrent tasks on GPU. In addition, inference tasks submitted in multi-tenant scenarios urgently need flexible scheduling strategies to meet the quality of service requirements for con-current inference tasks. To solve these problems, this paper proposes GNNSched, which efficiently manages the co-location of GNN inference tasks on GPU. Specifically, GNNSched organizes concurrent inference tasks into a queue and estimates the memory occupation of each task based on a cost function at the operator level. GNNSched implements multiple scheduling strategies to generate task groups, which are iteratively submitted to GPU for concurrent execution. Experimental results show that GNNSched can meet the quality of service requirements for concurrent GNN inference tasks and reduce the response time of inference tasks.

Key words: graph neural network (GNN); graphic processing unit (GPU); inference framework; task scheduling; estimation model

* **收稿日期**:2022-12-28;**修回日期**:2023-03-04
基金项目:科技创新 2030——"新一代人工智能"重大项目(2022ZD0117805);国家自然科学基金(62072018, 62322201, U22A2028);中央高校基本科研业务费专项资金(YWF-23-L-1121)
通信作者:杨海龙(hailong.yang@buaa.edu.cn)
通信地址:100191 北京市海淀区北京航空航天大学计算机学院
Address: School of Computer Science and Engineering, Beihang University, Haidian District, Beijing 100191, P. R. China

1 引言

深度学习 DL(Deep Learning)已在大量应用领域中得到广泛使用,从目标检测和图像分类到自然语言处理和机器翻译。随着更多新兴深度神经网络 DNN(Deep Neural Network)模型被提出,DNN 模型对算力的需求呈现快速增长的趋势,研究人员开始利用 TPU(Tensor Processing Unit)和 GPU 等硬件加速器来提高 DL 任务运行性能。特别是 GPU,由于其擅长处理 DNN 模型中的大量高度并行化矩阵计算,且得到了主流 DNN 框架的普遍支持,已经成为主流服务器中提供 DNN 模型算力的主体[1]。

与此同时,由于强大的节点表示能力,图神经网络 GNN(Graph Neural Network)在基于图的预测任务上取得了不错的效果[2]。GNN 结合图操作和神经计算来表征数据关系。由于图数据的不规则性,在 GPU 上实现高性能 GNN 极具挑战性。学术界虽然提出了基于节点分区和缓存合并的优化策略来解决 GNN 执行中的负载不均衡和线程分歧[3]等问题,但是图相关算子的实现仍使得 GPU 利用率低。例如,PyG(PyTorch Geometric)[4]通过消息传递单独更新节点特征,但其频繁的数据移动会导致计算停顿。DGL(Deep Graph Library)[5]使用类 SpMM(Sparse Matrix-matrix Multiplication)的内核来实现同时更新,但稀疏数据读取会降低访存效率。

为了简化集群管理,最常见的方法是将 GPU 资源分配的最小粒度设置为整个 GPU[6]。而由于 GPU 算力的不断提高,单个 DL 任务很难充分利用 GPU 资源[7],特别是对于 GNN 这类访存密集型任务,其性能会随着分配的计算资源的增加而达到饱和状态[8]。通过对 DNN 的研究发现,多个 DNN 任务可以在 GPU 上共置以提升资源利用率。工业界实现了多进程服务器 MPS(Multi-Process Server)和多实例 GPU MIG(Multi-Instance GPU),以使多个 CUDA(Compute Unified Device Architecture)进程通过资源分区共享 GPU。在学术界,时间共享[9,10]通过重叠预处理和计算来降低流水线延迟,而空间共享[11,12]允许并发执行 DNN,以提供更高的吞吐量。上述机制只适用于具有固定大小输入的 DNN,无法直接适配到显存消耗和计算强度与模型输入动态相关的 GNN[13]。

相比训练框架,基于 GPU 的推理框架需要应对的问题更加复杂。除改进整体吞吐量以外,还必须在限定时间内提供推理结果,以满足服务质量目标 QT(Quality-of-service Target)[14]。然而,单个 GPU 上运行多个推理任务可能会因为显存过载导致执行失败或延迟显著增加。因此,推理系统需要提前估计推理任务的显存占用情况,避免其需求超出显存容量进而触发开销较大的统一虚拟内存 UVM(Unified Virtual Memory)数据交换[15]。另一方面,云服务商通常以多租户方式共享 GPU 集群资源[16]。在这种情况下,需要根据推理任务的计算模式和显存占用特点设计灵活的调度机制,以满足服务质量要求并降低推理响应时延。

为了应对上述挑战,本文提出并发 GNN 推理任务调度框架 GNNSched(GNN Scheduler),其在 GPU 上高效地调度和管理并发 GNN 推理任务。GNNSched 首先将推理任务打包到队列中,并提取有关任务输入和网络结构信息。之后,GNNSched 分析每个推理任务的计算图,并量化算子对显存占用的影响。最后,GNNSched 利用多种调度策略对任务进行分组并迭代地分配显存以供执行。本文最后开展了大量的实验来评估 GNNSched,以验证其在满足服务质量和降低延迟等方面的有效性。

GNNSched 在任务组内和任务组间分别使用了空间共享和时间共享技术。具体来说,组内的任务通过空间共享提高整体 GPU 吞吐量,而组间通过重叠数据预处理与计算降低流水线延迟。本文是首次针对并发 GNN 推理任务的调度优化和显存管理进行研究。GNNSched 已开源于:https://github.com/sunqingxiao/GNNSched。

本文的具体工作如下:

(1)提出了并发 GNN 推理任务管理机制,通过细粒度的显存管理和工作器(Worker)分配以自动执行 GNN 推理任务。此外,提出了多种调度策略对任务进行分组。

(2)提出了 GNN 推理任务显存占用估计策略,针对 GNN 算子设计了显存成本函数,并通过遍历前向传播的计算图来估计 GPU 显存占用情况。

(3)实现了并发 GNN 推理任务调度框架 GNNSched,其可以有效地调度和管理在 GPU 上的并发 GNN 推理任务。实验结果表明,GNNSched 能够满足服务质量要求并降低推理任务响应时延。

2 背景

2.1 图神经网络简介

近年来,一些研究致力于将深度学习应用于图

等非结构化数据[2]。不同于传统深度学习模型处理的图像和文本等密集数据,图表示稀疏且连接不规则。图中每个节点与一个特征向量相关联,节点之间的边表示图拓扑结构,并以边的权重进行量化。GNN 以图结构数据作为输入,综合图结构和节点特征来学习数据关系。表 1 列出了重要的 GNN 符号。

Table 1 Explanations of important symbols in GNN

表 1 GNN 中重要符号说明

符号	定义
G	图 $G=(V,E)$
v,V	图中节点和节点集合,节点 $v \in V$
e_{ij},E	图中边和边集合,边 $e_{ij} \in E$
D_v	节点 v 的度
N_v	节点 v 的邻居节点集合
$\boldsymbol{h}_v$	节点 v 的特征向量
$\boldsymbol{A}$	图的邻接矩阵
$\boldsymbol{X}$	由特征向量组成的特征矩阵
σ	Sigmoid 激活函数
k	网络层索引
$\boldsymbol{W}$	权重矩阵
ε	可学习的模型参数

图卷积神经网络 GCN(Graph Convolutional Network)[17]是面向图学习的最成功的网络之一。GCN 缓解了图的局部邻域结构过拟合的问题,其图操作如式(1)所示:

$$\boldsymbol{h}_v^k=\sigma\Big(\sum_{u\in(N(v)\bigcup\{v\})}(\boldsymbol{W}^k\boldsymbol{h}_u^{k-1}/\sqrt{D_u\cdot D_v})\Big) \quad (1)$$

SAGE(SAmple and aggreGatE)[18]进一步应用采样的方式来为每个节点获取固定数目的邻居。SAGE 的图操作如式(2)所示:

$$\boldsymbol{h}_v^k=\sigma(\boldsymbol{W}^k\cdot f_k(\boldsymbol{h}_v^{k-1},\{\boldsymbol{h}_u^{k-1},\forall u\in S_{N(v)}\})) \quad (2)$$

其中 $S_{N(v)}$ 是节点 v 的随机采样邻居,$f_k(\cdot,\cdot)$ 是聚合函数。图同构网络 GIN(Graph Isomorphism Network)[19]通过可学习参数 ε^k 调整中心节点的权重,其图操作如式(3)所示:

$$\boldsymbol{h}_v^k=MLP\big((1+\varepsilon^k)\boldsymbol{h}_v^{k-1}+\sum_{u\in N_v}\boldsymbol{h}_u^{k-1}\big) \quad (3)$$

基于上述分析,GNN 的核心计算可以抽象为式(4):

$$\boldsymbol{X}^{k+1}=\sigma(\hat{\boldsymbol{A}}\boldsymbol{X}^k\boldsymbol{W}^k) \quad (4)$$

其中 $\hat{\boldsymbol{A}}$ 由 $\boldsymbol{A}$ 计算得到且随网络变化。由于 $\boldsymbol{A}$ 极大且稀疏,式(4)可以视为链式稀疏矩阵乘 SpMM[20]。然而,由于显存带宽的限制,类 SpMM 运算难以充分利用 GPU 的计算资源。

2.2 GNN 框架的计算模式

图 1 展示了 GNN 前向传播的计算流程。典型的 GNN 层包含聚合阶段和更新阶段,其结合了图操作和神经计算。聚合阶段从节点的每个邻居检索一个特征向量,并将这些向量聚合成一个新的特征向量。更新阶段执行多层感知机 MLP(MultiLayer Perceptron)等神经操作以转换每个节点的特征向量。GNN 框架根据图结构进行图操作,其中边表示数据传输。DGL 通过中心邻居模式引入了节点级并行,它从特征矩阵中获取数据,再执行类 SpMM 的归约操作以同时更新节点特征。PyG 通过 MessagePassing 抽象引入了边级并行,它通过消息传递在所有边上直接生成消息,再分别执行归约操作。

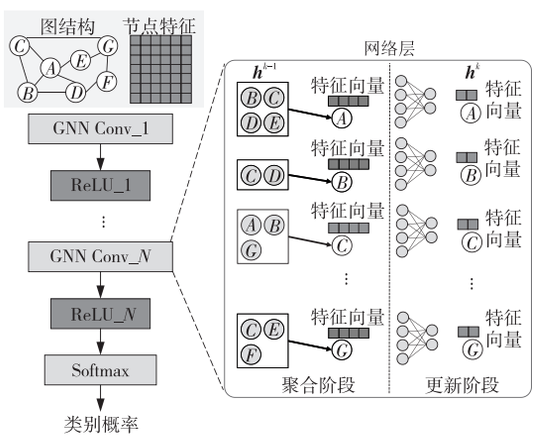

Figure 1 Computation workflow of GNN

图 1 GNN 计算流程

DGL 和 PyG 都受到了 GPU 计算资源不足的限制。DGL 应用类 SpMM 内核实现节点特征的更新,但是由于图结构的不规则性,显存访问成为了其性能瓶颈。PyG 通过聚合内核实现单独的节点特征更新,以提高访存效率,但是耗时的数据移动会导致计算停顿。为解决上述问题,可以利用 GNN 推理任务的共置机制最大化 GPU 吞吐量。然而,这需要预知推理任务的显存消耗情况以免显存过载。不同于拥有固定大小输入的神经网络,GNN 层的输出维度与图维度和特征长度紧密相关。此外,为了得到更精确的估计值,图传播的显存消耗也需要被纳入考虑。

2.3 推理任务的 GPU 共享

在工业界,主流做法是将 GPU 分配的最小粒度设置为整个 GPU[6]。虽然这样的设置简化了集群资源管理,但导致 GPU 资源的利用率较低。因此,GPU 共享逐渐成为在 GPU 上共置推理任务的

一项基本技术。例如,即使是单个服务也可能包含多个异构推理任务[21],如何将其映射到GPU是重要挑战。对具有自身计算要求的不同推理任务必须适时地加载到GPU,以满足其服务质量目标,同时提高整体吞吐量。

现有工作提出了基于时间或空间共享的机制[9,12],以实现深度学习任务在GPU上的共置。时间共享高度灵活,将GPU显存和核心专用于特定持续时间的单次执行。PipeSwitch[9]利用流水线模型传输和主备Worker来最小化切换开销,从而满足推理任务的严格服务质量目标。REEF[10]改造了GPU驱动以支持软件队列和计算单元的重置,并主动抢占批量内核从而在微秒级启动实时推理任务。尽管时间共享通过重叠数据预处理和计算来隐藏延迟,但仍难以充分发掘GPU的计算潜力。例如,对于递归神经网络构成的语言模型,计算单元往往会长时间闲置[16]。

相比之下,空间共享允许在不违反服务质量的情况下提供更高的GPU吞吐量。应用空间共享的一个限制是并发任务的工作集大小。如果工作集大小超过GPU显存,系统必须将数据交换到主机,这会掩盖空间共享的优势。GSLICE[14]采用自调整算法并根据性能反馈调整每个推理任务的线程占用率。Abacus[11]通过确定性算子重叠实现了并发推理任务的延迟可预测性,并设计基于配额的控制器以确定算子执行顺序。Choi等人[12]创建GPU资源抽象层为推理任务分配有效资源,再通过性能预测模型评估空间共享的潜在干扰开销。然而,这些方法均未考虑并发推理任务可能带来的显存过载,这可能会导致任务运行失败或耗时的设备-主机数据交换[15]。

上述机制针对的是具有固定大小输入的传统神经网络,没有量化不规则图对显存消耗的影响。此外,对于同一批次到达的推理任务,可以结合时间共享和空间共享在GPU上实现更灵活高效的调度机制。

3 方法设计与实现

3.1 设计概要

本节提出GNN推理调度框架GNNSched来维护GNN推理队列,每次从队列头部取同一批次的推理任务进行高效调度和管理。如图2所示,灰色模块是由GNNSched设计或扩展的。GNNSched由4个重要组件组成,包括显存管理器、峰值显存消耗*PMC*(Peak Memory Consumption)分析器、任务调度器和Worker分配器。显存管理器维护统一的显存池,按需分配显存;*PMC*分析器提取运行时信息以估计推理任务的显存消耗;任务调度器根据调度策略确定分组和任务执行顺序;Worker分配器将训练任务绑定到具体的Worker上,执行后返回结果。注意Worker是指负责任务执行的进程,其常驻在推理系统中,跨不同组串行执行推理任务。

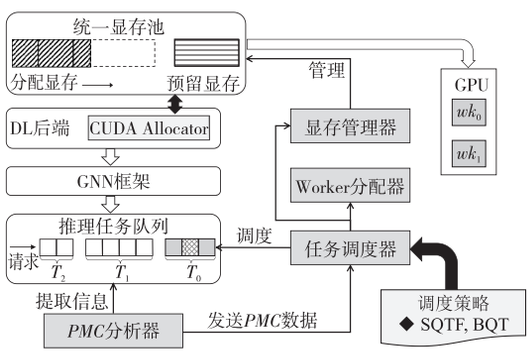

Figure 2 Design overview of GNNSched

图2 GNNSched设计概要

图2为GNNSched的设计概要。GNNSched将CUDA Allocator集成到DL后端以实现显式的GPU显存管理。GNNSched将使用GNN框架实现的推理任务打包到任务队列中(T_0,T_1和T_2为推理请求到达时间),其中*PMC*分析器提取模型输入和网络结构的详细信息。*PMC*分析器将GNN模型的计算图表示为有向无环图DAG(Directed Acyclic Graph),并使用公式量化每个算子对显存消耗的影响。*PMC*分析器将*PMC*信息发送给任务调度器,任务调度器使用特定策略对推理任务进行分组和重排,再迭代地从队列中弹出任务组。在每次迭代中,显存管理器和Worker分配器接收信号以分配共享GPU显存和执行推理任务。

3.2 任务管理机制

并发任务空间共享的关键在于GPU显存的细粒度管理。本文扩展了DL后端(PyTorch)的统一显存池,针对DL任务的特点,将显存池划分为预留显存和分配显存。预留显存存储框架内部数据,例如CUDA上下文和模型工作区,通常在任务执行前预先分配,分配显存存储任务运行时产生的张量,例如层输出。GNNSched从分配显存的一端连续插入任务Buffer,以确保分配显存被完全占用(见图2)。GNNSched通过显存消耗估计来指定每个任务Buffer的分配大小,再将其插入到特定的显存位置。对于分配显存,GNNSched通过满足

对齐要求的额外显存填充来处理内部张量碎片。

图 3 给出了 GNNSched 中任务管理的整体工作流程。对于同一批次的推理任务,任务调度器执行分组操作,Worker 分配器以任务组为粒度迭代地提交到 GPU。以这种方式,GNNSched 避免了显存碎片问题。显存碎片可能浪费并发推理机会,同时使内存维护复杂化。在每次迭代中,Worker 分配器将推理任务映射到具体 Worker(wk_0,wk_1 和 wk_2)。每个 Worker 串行处理 Buffer 插入、任务执行和结果返回等操作;Worker 间并行以重叠相关操作,其中不同推理任务以空间共享的方式执行。并行 Worker 均返回结果后,GNNSched 清除所有 Buffer 并前进到下一组。注意,GNNSched 在 GPU 上执行当前任务组的同时,在主机端预先处理下一组的输入数据,因而有效地降低了流水线延迟。

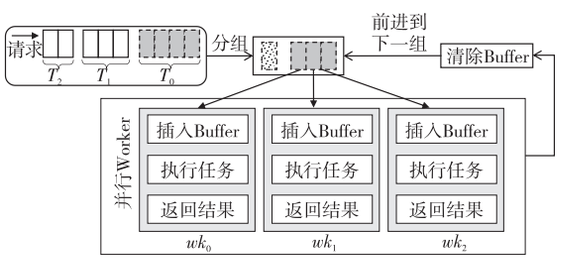

Figure 3 Overall workflow of task management

图 3 任务管理的整体工作流程

从以上分析可知,GNNSched 灵活的管理机制可以支持推理任务的任意分组和调度顺序。这对于提高输入敏感 GNN 的访存效率是必要的,其 PMC 随图维度和特征长度而显著变化。

3.3 显存消耗估计

受文献[22]的启发,本文将 GNN 推理的计算图 CG 表示为 DAG,如式(5)所示:

$$CG=\langle\{op_i\}_{i=1}^{n},\{ed_j=(op_x,op_y)\}_{j=1}^{m}\rangle \quad (5)$$

其中,节点 op_i 表示数学调用的算子,边 ed_j 指定执行依赖。$TO=\langle ed_1,ed_2,\cdots,ed_m\rangle$ 是 DAG 规定的拓扑顺序,其通过查阅 DL 后端[23]内的拓扑顺序预先生成。GNNSched 利用 TO 遍历计算图并根据张量的分配和释放更新 PMC。

图 4 说明了 2 层 SAGE 模型推理的 DAG 示例。SAGE 层(SAGEConv)由图算子(Propagate)和神经算子(Linear)组成。激活函数(如 ReLU)由于其零显存成本而未在图中显示。输入图数据($\boldsymbol{Data\_X}$)馈入到神经网络并经由上述算子操作。注意和训练不同的是,推理只具有前向传播的过程。因此,SAGE 层的输出张量($\boldsymbol{O}_{\mathrm{m}}^{3}$)用完即释放,而无需经反向传播来计算权重梯度。

根据拓扑遍历,估计 GPU 显存消耗可以形式化为计算图上每个算子所需显存的累加。本文为每个算子定义了 DL 后端无关的显存成本函数 MCF(Memory Cost Function)。显存成本函数返回一组已分配的具有类别和形状的张量,这些张量通过输入维度和形状推断来推导得到。算子 op 的 MCF 可以表示为式(6):

$$MCF(op)=W(op)\cup O(op)\cup E(op) \quad (6)$$

其中,W、O 和 E 分别是权重张量、输出张量和临时张量的集合。

表 2 给出了 GNN 中典型算子的分配张量和张量大小,其中 NE 和 NN 表示图结构的节点数和边数,FL_{in}^{i} 和 FL_{out}^{i} 分别表示第 i 个 GNN 层的输入特征长度和输出特征长度。$Matmul$ 是具有求和归约的 GCN 的核心算子,其前向过程可以抽象为链式 SpMM。一般张量的生命周期主要取决于它是否会被后续的算子使用(即计算图上是否存在边)。然而,权重张量是个例外,它们属于模型内部参数且持久存放在 GPU 显存。此外,临时张量仅在算子内部临时分配,并在算子完成后释放。根据张量的生命周期信息,GNNSched 在图遍历结束时得到 GNN 推理任务的估计 PMC。GNNSched 根据估计结果指定插入任务 Buffer 的显存位置,从而避免并发任务出现访存冲突而崩溃。

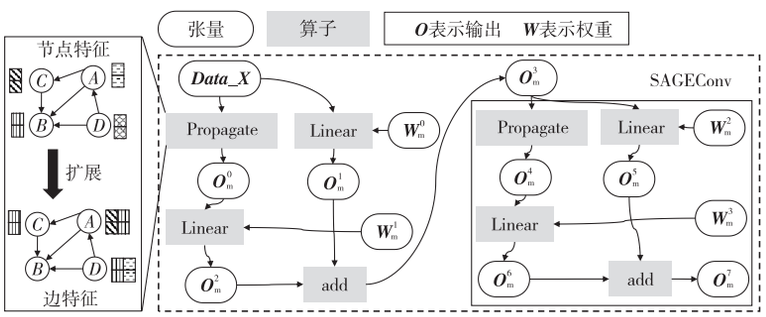

Figure 4 Computation graph of two-layer SAGE model

图 4 2 层 SAGE 模型的计算图

本文还需要考虑GNN的定制化细节。GCN补充自循环以使中心节点的聚合表示包含其自身特征。即便如此,GNN框架中仍不可避免地会创建部分临时张量来维护任务执行。因此,GNNSched将估计*PMC*乘以固定阈值*threshhold*,以保证插入的Buffer能够满足计算过程的显存需求。

Table 2 Allocated tensors and their sizes

表2 分配张量的类型和大小

算子类型	算子	张量类型	张量大小
图算子	*Propagate*	临时张量	$NE\times FL_{\text{in}}^{i}$
		输出张量	$NN\times FL_{\text{in}}^{i}$
矩阵算子	*Matmul*	权重张量	$FL_{\text{in}}^{i}\times FL_{\text{out}}^{i}$
		输出张量	$NN\times FL_{\text{out}}^{i}$
神经算子	*Linear*	权重张量	$FL_{\text{in}}^{i}\times FL_{\text{out}}^{i}$
		输出张量	$NN\times FL_{\text{out}}^{i}$

上述抽象和形式化对于各种GNN框架的*PMC*估计是通用的。GNNSched还可以通过修改计算图或显存成本函数来适配其他GNN。

3.4 调度策略

GNNSched的灵活管理机制为调度策略提供了很大的设计空间。与文献[11]一致,本文将每个推理任务的*QT*设置为其单独运行时间的2倍。因此,高*QT*任务具有更长的执行时间,而执行时间通常与计算强度正相关。本文实现了2种非抢占式的调度策略,包括最短*QT*优先SQTF(Shortest-QT-First)和平衡*QT* BQT(Balanced-QT)。所有策略均具有"安全"条件,以确保并发任务的显存使用不超过GPU显存容量。幸运的是,GNNSched可以估计推理任务的*PMC*并进行累加,从而使每个任务组都处于"安全"位置。接下来,说明本文2种调度策略的具体细节:

(1)SQTF策略:最简单的先进先出FIFO(First-In-First-Out)算法可以在满足"安全"条件下尽可能地将更多的任务打包到同一组中。然而,FIFO算法可能会导致短期任务因等待大型正在进行的任务完成而遭受长时间的排队延迟。因此,提出了最短作业优先算法和最短剩余时间优先算法[24],以降低队列中任务的排队延迟。然而,任务持续时间或剩余时间需要离线分析,这会影响可用性和运营成本[25]。为了解决以上问题,本文提出SQTF策略(见算法1),其将队列中的任务索引按照*QT*的升序进行排序,然后遍历排序生成任务组。此外,本文设置了分组阈值以改善时间维度的负载均衡。具体来说,分组阈值由当前批次的推理任务的*PMC*总和计算得到,在分组过程中保证不同任务组的显存分配大小相近。

算法1 SQTF调度策略

输入:升序队列*Deque*,分配显存大小*MA*,推理任务的*PMC*总和*SP*;

输出:任务组列表*taskGroup*。

```
1: queSize ← Deque.size; //原始队列大小
2: gTH = ceil(SP/ceil(SP/MA)); //分组阈值
   //初始化组PMC和组计数器
3: gPMC,gCounter←MA,−1;
4: for i in range[0,queSize) do
5:    task ← Deque.popleft(); //从右侧弹出任务
6:    if gPMC > gTH then
7:       taskGroup.append([]);//前进到下一组
8:       gPMC,gCounter←0,gCounter+1;
9:    end if
10:   taskGroup[gCounter].append(task);
11:   gPMC←gPMC+task.PMC;//累加PMC
12:end for
```

(2)BQT策略:该策略继承了SQTF策略的*QT*排序和分组阈值机制。尽管SQTF策略降低了短期任务的排队延迟,但将具有高*QT*的推理任务置于同一组可能会加剧资源冲突。BQT策略用于平衡任务时长和排队时间。BQT策略的原则是将计算强度高的任务和强度低的任务归为一组,从而降低并发执行的性能干扰。算法2为BQT策略的实现细节。该策略根据*QT*的升序将推理任务推入双端队列(*Deque*)。每次迭代从*Deque*的右端或左端弹出任务(第5～9行),再将任务分到当前组并累加*PMC*(第14,15行)。当组*PMC*大于分组阈值时,前进到下一组并初始化组计数器(第10～13行)。重复上述步骤直到*Deque*为空,当前批次的任务组生成结束。

算法2 BQT调度策略

输入:升序队列*Deque*,分配显存大小*MA*,推理任务的*PMC*总和*SP*;

输出:任务组列表*taskGroup*。

```
1: queSize←Deque.size;//原始队列大小
2: gTH=ceil(SP/ceil(SP/MA));//分组阈值
   //初始化组PMC和组计数器
3: gPMC,gCounter←MA,−1;
4: for i in range[0,queSize) do
5:    if i%2==1 then
6:       task←Deque.pop();//从右侧弹出任务
7:    else
8:       task←Deque.popleft();//从左侧弹出
```

```
9:   end if
10:  if gPMC>gTH then
11:    taskGroup.append([]);//前进到下一组
12:    gPMC,gCounter←0,gCounter+1;
13:  end if
14:  taskGroup[gCounter].append(task);
15:  gPMC←gPMC+task.PMC;//累加 PMC
16:end for
```

3.5 实现细节

GNNSched 的系统原型由 C++和 Python 代码实现,并构建在 PyG 和 PyTorch[26]之上。然而,GNNSched 背后的思想可通用于其他 GNN 框架或 DL 后端。本文使用 CUDA IPC 扩展 PyTorch 的 Allocator 模块从而实现 GPU 显存池的显式管理。本文通过添加函数来支持在特定 CUDA 流中插入 Buffer 以及从显存池中清除 Buffer。Buffer 插入函数可以被多次调用,以实现 GNN 推理任务的空间共享。任务组执行完成后,Buffer 清除函数被调用来清除显存分配。注意 PyTorch 为张量分配的实际显存大小需满足一定的对齐要求。为了解决这一问题,本文通过填充来使得显存分配大小满足 512 字节的倍数。

GNNSched 由分别负责队列管理和任务执行的调度进程和 Worker 进程组成。调度进程监听客户端通过 TCP 端口发送的任务请求,将任务打包到队列中并加载模型结构。接下来,调度进程通过计算图遍历来分析 *PMC* 并生成任务组。在每个组迭代中,调度进程将任务的哈希索引发送到 Worker 进程。Worker 进程根据哈希索引识别任务并将其分配给 Worker 线程。每个 Worker 线程加载相应的 GNN 模型并将其附加到 CUDA 流。结果返回后,Worker 线程通过 PyTorch Pipe API 向调度进程发送“完成”信号。当接受到的信号数量等于组大小时,调度进程迭代到下一个任务组。

4 实验与结果分析

4.1 实验设置

(1)硬件和软件配置。硬件规格如表 3 所示。操作系统为 Ubuntu 20.04,编译器版本为 GCC v9.3 和 NVCC v11.1。GNNSched 基于 PyG v1.7 和 PyTorch v1.8 构建。本文修改了 PyTorch 以支持 GNNSched 的显式任务共置和显存管理。

Table 3 Hardware configuration

表 3 硬件配置

规格	CPU	GPU
模型	Intel® Xeon® E5-2680 v4	NVIDIA® Tesla® V100
频率/GHz	2.4	1.5
核心	28	13 440 (80 SMs)
缓存	256 KB L2,35 MB L3	6 MB L2
显存	378 GB DDR4	32 GB HBM2
带宽/(GB/s)	76.8	900

(2)图数据集和任务队列。用于实验的图数据集如表 4 所示。本文为每个图数据集随机生成 25 个子图,用作网络输入。本文选取 3 个典型 GNN(包括 GCN、SAGE 和 GIN),其中层数和层宽分别设置为 8 和 256。本文将 SAGE 的采样率设置为 0.5(与 PyG 默认设置一致)。本文为每个 GNN 生成由 100 个推理任务组成的队列,不同任务具有不同的子图输入。此外,本文从以上队列中均匀采样 100 个任务以获得具有不同 GNN 的任务队列(命名为 MIX)。队列中任务到达的时间遵循泊松分布。同时到达的任务数量均值被设置为 1 和 2,分别代表低负载(Low Load)和高负载(High Load)。多样的任务组织用于对 GNNSched 的有效性进行全面评估。

Table 4 Graph datasets

表 4 图数据集

数据集	节点数	边数	特征数	类别数
pubmed	19 717	88 676	500	3
artist	50 515	1 638 396	100	12
amazon	410 236	4 878 875	96	22
reddit	232 965	114 615 891	602	50

(3)对比方法和指标。本文将具有 2 种调度策略(SQTF 和 BQT)的 GNNSched 与 Default 和 MPS 进行对比。Default 采取 FIFO 的方式串行执行推理任务。为了公平起见,本文基于 MPS 实现了 2 个变体 MPS-Base 和 MPS-Aggr。MPS-Base 的最大并发数量为 4,MPS-Aggr 并发执行同一批次的所有任务。本文为 MPS 启动 UVM 以处理可能的显存过载。为了评估推理任务的服务质量和执行效率,本文选取服务质量违反率(QoS violation)、响应延迟(medium/90%-ile/99%-ile latency)、作业完成时间 *JCT*(job completion time)和排队时间 *QUET*(QUEuing Time)作为评估指标。本文将每种方法运行 10 次并给出平均结果以隔离随机性的影响。

4.2 服务质量评估

图5给出了不同方法的服务质量违反率对比。相比Default和MPS,GNNSched在所有任务队列下均取得更低或等同的服务质量违反率。在低负载下,GNNSched使得所有任务均满足其服务质量要求。即便在高负载下,GNNSched的服务质量违反率最高仅为8%,而Default和MPS的最高违反率分别达到了93%和91%。另外,还注意到,MPS-Base和MPS-Aggr呈现明显差异性的实验结果。MPS-Base的服务质量违反率总是低于20%,而MPS-Aggr在半数队列下甚至比Default的违反率更高。这是因为MPS的贪心机制使得显存过载,UVM的分页开销显著降低了并发任务的性能。尽管MPS-Base的静态分组有效规避了UVM的使用,然而其欠缺任务调度的灵活性。GNNSched通过*PMC*预估计和高效调度来保证在显存安全下尽可能地充分利用计算资源。

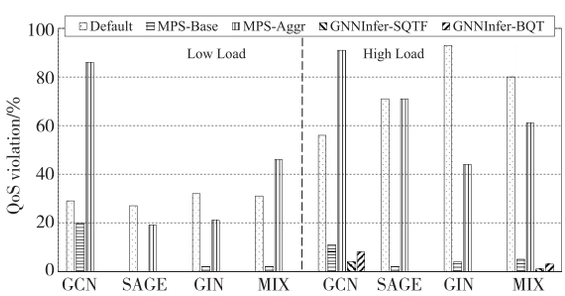

Figure 5 Comparison of QoS violation rate

图5 服务质量违反率对比

4.3 延迟评估

图6给出了归一化为*QT*的响应延迟对比。Default的串行执行忽略了任务并发的机会,在GIN队列下的medium延迟、90%-ile延迟和99%-ile延迟分别达到了*QT*的6.6倍、12.2倍和17.3倍。MPS-Aggr由于UVM开销表现不稳定,在GCN队列下取得7.5倍*QT*的99%-ile延迟。相比之下,GNNSched和MPS-Base在所有任务队列下均取得小于2倍*QT*的99%-ile延迟。以上结果表明,GNNSched和MPS-Base具有更低的尾延迟,这对于提升用户粘性尤为重要。

从图6可以观察到,GNNSched在大多数案例中取得了最低延迟。GNNSched相比MPS-Base分别平均降低了26.2%,27.4%和31.9%的medium延迟、90-ile延迟和99-ile延迟。这说明GNNSched能够提供稳定的服务质量,有效避免了处理批量任务时的长尾延迟。GNNSched的优越性主要来自于其充分发掘了推理任务的并发机会。

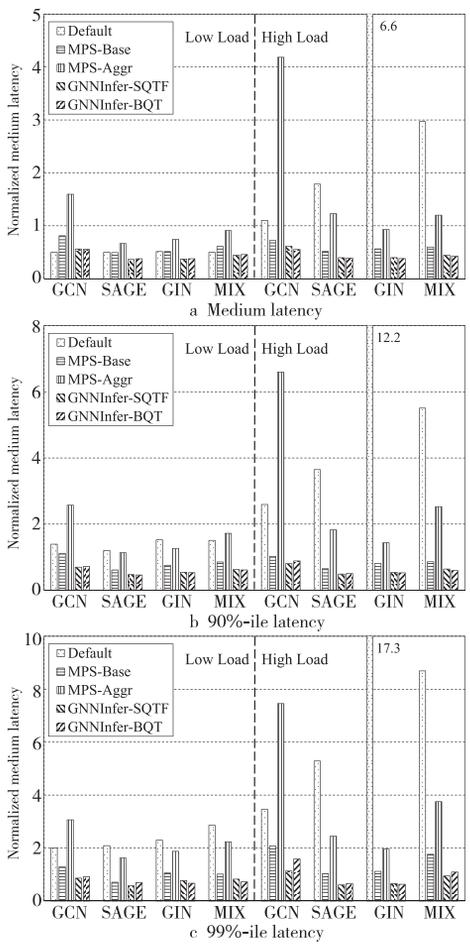

Figure 6 Comparison of request latency

图6 响应延迟对比

另外,GNNSched设置的分组阈值不但改善了时间维度的负载均衡,还在很大程度上缓解了任务空间共享时的性能干扰。

4.4 性能评估

图7给出了归一化为*QT*的平均*JCT*对比。GNNSched在所有任务队列下均取得了最短的*JCT*。与Default、MPS-Base和MPS-Aggr对比,GNNSched的*JCT*分别平均降低了60.6%,26.8%和62.7%。除了灵活的任务并发机制,GNNSched的显存池管理同样有助于性能提升,其避免了张量的频繁分配和释放。值得注意的是,GNNSched-BQT在大多数任务队列下略微好于GNNSched-SQTF,*JCT*平均降低了0.6%。这是因为BQT策略将计算强度高的和计算强度低的任务归为一组,降低了对计算和缓存资源的竞争。

图8给出了归一化为*QT*的平均排队时间对比。对于低负载,MPS-Base和GNNSched在所有任务队列下的排队时间均为0。这意味着上一批次的推理任务总能在当前批次到达前完成,侧面证明了空间共享的有效性。对于高负载,MPS-Base

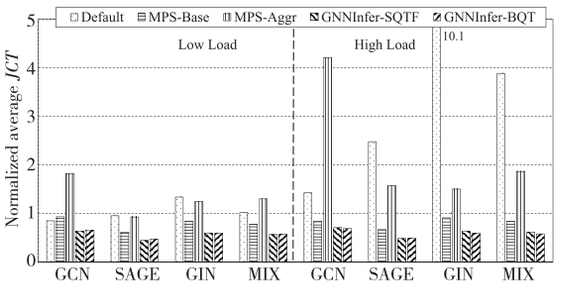

Figure 7 Comparison of average *JCT*

图 7 平均作业完成时间对比

在所有队列下均存在等待延迟,而 GNNSched 在 SAGE 队列和 GIN 队列下的排队时间仍为 0。GNNSched 的灵活共享机制重叠了数据预处理和计算,改进了任务组间的流水线执行效率。另外,GNNSched-SQTF 明显比 GNNSched-BQT 的排队时间更短,平均降低了 0.8%。原因是 SQTF 策略优先调度短时执行任务,缓解了等待长时任务完成时的队头堵塞。

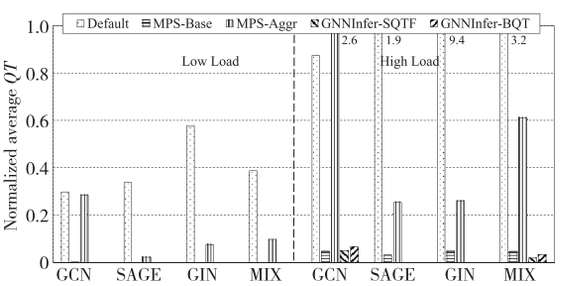

Figure 8 Comparison of average *QUET*

图 8 平均排队时间对比

4.5 显存估计精度

PMC 的精确估计对于保证并发执行的显存安全是必要的。本文使用相对误差 *RE*(relative error)来度量估计精度。图 9 给出了 GNN 推理任务的 *PMC* 估计的相对误差,可以观察到所有任务的误差均低于 8%。原因是显存成本函数准确地获取了 GNN 算子的显存消耗。另一方面,随着子图节点数和边数的变化,GNNSched 取得了稳定的估计精度。这表明基于网络结构生成的 DAG 很好地表示了前向传播的计算流程。根据以上结果,本文将乘法阈值(*threshhold*)设置为 1.1 从而确保空间共享下的显存安全。

4.6 开销分析

本节将 GNNSched 的处理开销归一化为任务推理时间。处理开销可以分为 *PMC* 估计、任务调度和 Buffer 插入 3 个部分。*PMC* 估计通过显存成本函数遍历计算图以获取 *PMC* 信息。任务调度将并发推理任务组织为队列,再根据调度策略对队列进行重排并生成任务组。Buffer 插入基于 *PMC* 信息将 Buffer 插入到分配显存的特定位置。图 10 显示了 GNNSched 处理开销的时间分解。可以看到,处理开销相对于任务执行可以忽略不计,在低负载和高负载下的平均时间仅为推理时间的 2.4‰和 3.0‰。这表明使用 GNNSched 来管理相异的推理任务可以有效提高 GPU 吞吐量。

Figure 9 Relative errors of *PMC* estimation

图 9 *PMC* 估计的相对误差

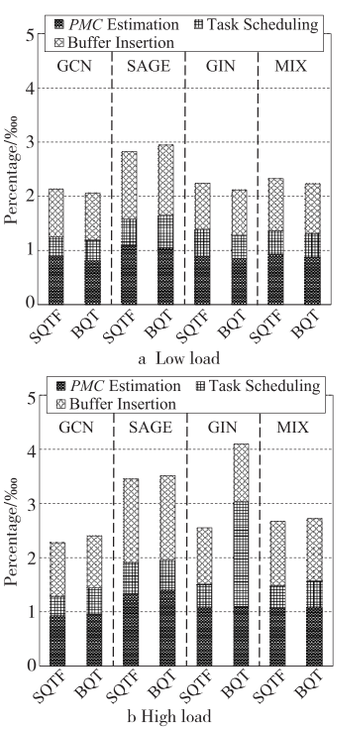

Figure 10 Time breakdown of processing overhead

图 10 处理开销的时间分解

5 相关工作

5.1 图神经网络的算子加速

最近研究工作深入挖掘了 GNN 的计算特征,并针对 GPU 架构进行细粒度优化。FeatGraph[27] 结合图划分与特征维度优化聚合阶段的缓存利用。

Huang 等人[3]通过局部敏感哈希对节点进行聚类,再对邻居进行分组以缓解负载不均衡。GNNAdvisor[13]通过引入 warp 对齐的线程映射和维度划分来减少线程分歧。QGTC(Quantized graph neural networks via GPU Tensor Core)[28]基于低位数据表示和位分解的量化技术实现了张量核心定制的计算内核。本文工作与上述工作在性能优化上互补,GNNSched 的重点在于 GNN 推理任务的并发调度。

5.2 深度学习推理调度系统

许多研究工作提出了各种调度系统来改进部署在集群上的深度学习推理任务的服务质量。Clipper[29]设计了模型抽象层以在框架之上实现缓存和自适应批处理策略。MArk[30]动态处理批量推理请求并适时地使用 GPU 以改进性能。Nexus[21]通过打包机制将推理任务组调度到集群并指定每个任务的 GPU 使用。Clockwork[31]提供性能可预测系统,其通过中央控制器调度请求并预先放置模型。这些系统均未考虑推理任务在单 GPU 上的空间共享,从而导致计算资源的利用率较低。

6 结束语

本文提出了并发 GNN 推理任务调度框架 GNNSched,其可以高效管理 GPU 上共置的推理任务。GNNSched 将推理任务组织为队列,并提取图输入数据和模型网络结构信息;之后,GNNSched 对每个任务的计算图进行分析,并利用算子成本函数估计任务显存占用;最后,GNNSched 实现了多种调度策略用于生成任务组,并将其迭代地提交到 GPU 上执行。实验结果表明,GNNSched 可以满足推理任务服务质量要求并降低响应时延。

参考文献:

[1] Xiao W C,Bhardwaj R,Ramjee R,et al. Gandiva: Introspective cluster scheduling for deep learning[C]//Proc of the 13th USENIX Conference on Operating Systems Design and Implementation,2018: 595-610.

[2] Wu Z H,Pan S R,Chen F W,et al. A comprehensive survey on graph neural networks[J]. IEEE Transactions on Neural Networks and Learning Systems,2020,32(1): 4-24.

[3] Huang K Z, Zhai J D, Zheng Z, et al. Understanding and bridging the gaps in current GNN performance optimizations [C]//Proc of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,2021: 119-132.

[4] Fey M,Lenssen J E. Fast graph representation learning with PyTorch geometric[J]. arXiv:1903.02428,2019.

[5] Wang M J, Zheng D, Ye Z H, et al. Deep graph library: A graph-centric, highly-performant package for graph neural networks[J]. arXiv:1909.01315,2019.

[6] Xiao W C,Ren S R,Li Y,et al. AntMan: Dynamic scaling on GPU clusters for deep learning[C]//Proc of the 14th USENIX Conference on Operating Systems Design and Implementation,2020: 533-548.

[7] Wu X F,Rao J,Chen W,et al. SwitchFlow: Preemptive multitasking for deep learning[C]//Proc of the 22nd International Middleware Conference,2021: 146-158.

[8] Sun Q X, Liu Y, Yang H L, et al. QoS-aware dynamic resource allocation with improved utilization and energy efficiency on GPU [J]. Parallel Computing, 2022, 113 (C): 102958.

[9] Bai Z H,Zhang Z,Zhu Y B,et al. PipeSwitch: Fast pipelined context switching for deep learning applications[C]//Proc of the 14th USENIX Conference on Operating Systems Design and Implementation,2020: 499-514.

[10] Han M C,Zhang H Z,Chen R,et al. Microsecond-scale preemption for concurrent GPU-accelerated DNN inferences [C]//Proc of the 16th USENIX Conference on Operating Systems Design and Implementation,2022: 539-558.

[11] Cui W H,Zhao H,Chen Q,et al. Enable simultaneous DNN services based on deterministic operator overlap and precise latency prediction[C]//Proc of the International Conference for High Performance Computing,Networking,Storage and Analysis,2021:Article No.:15.

[12] Choi S,Lee S,Kim Y,et al. Serving heterogeneous machine learning models on multi-GPU servers with spatio-temporal sharing[C]//Proc of USENIX Annual Technical Conference,2022: 199-216.

[13] Wang Y K,Feng B Y,Li G S,et al. GNNAdvisor: An adaptive and efficient runtime system for GNN acceleration on GPUs[C]//Proc of the 15th USENIX Conference on Operating Systems Design and Implementation,2021: 515-531.

[14] Dhakal A, Kulkarni S G, Ramakrishnan K K. GSLICE: Controlled spatial sharing of GPUs for a scalable inference platform[C]//Proc of the 11th ACM Symposium on Cloud Computing,2020: 492-506.

[15] Sun Q X,Liu Y,Yang H L,et al. CoGNN:Efficient scheduling for concurrent GNN training on GPUs[C]//Proc of the International Conference on High Performance Computing, Networking,Storage and Analysis,2022: 1-15.

[16] Peng Y H,Bao Y X,Chen Y R,et al. Optimus: An efficient dynamic resource scheduler for deep learning clusters[C]// Proc of the 13th European Conference on Computer Systems,2018: 1-14.

[17] Kipf T N, Welling M. Semi-supervised classification with graph convolutional networks[J]. arXiv:1609.02907,2016.

[18] Hamilton W, Rex Y, Leskovec J. Inductive representation learning on large graphs[C]//Proc of the 31st International

Conference on Neural Information Processing Systems, 2017: 1025-1035.

[19] Xu K, Hu W H, Leskovec J, et al. How powerful are graph neural networks[J]. arXiv:1810.00826, 2018.

[20] Li J J, Louri A, Karanth A, et al. GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks[C]//Proc of 2021 IEEE International Symposium on High-Performance Computer Architecture, 2021: 775-788.

[21] Shen H C, Chen L Q, Jin Y C, et al. Nexus: A GPU cluster engine for accelerating DNN-based video analysis[C]//Proc of the 27th ACM Symposium on Operating Systems Principles, 2019: 322-337.

[22] Gao Y J, Liu Y, Zhang H Y, et al. Estimating GPU memory consumption of deep learning models[C]//Proc of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020: 1342-1352.

[23] PyTorch: The topological sorting algorithm for computation graphs in PyTorch[EB/OL]. [2021-11-12]. https://github.com/pytorch/pytorch/blob/v1.8.0/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h.

[24] Gu J C, Chowdhury M, Shin K G, et al. Tiresias: A GPU cluster manager for distributed deep learning[C]//Proc of the 16th USENIX Conference on Networked Systems Design and Implementation, 2019: 485-500.

[25] Hu Q H, Sun P, Yan S G, et al. Characterization and prediction of deep learning workloads in large-scale GPU datacenters[C]//Proc of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021: Article No.: 104.

[26] Paszke A, Gross S, Massa F, et al. PyTorch: An imperative style, high-performance deep learning library[C]//Proc of the 33rd International Conference on Neural Information Processing Systems, 2019: 8026-8037.

[27] Hu Y W, Ye Z H, Wang M J, et al. FeatGraph: A flexible and efficient backend for graph neural network systems[C]//Proc of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2020: Article No.: 71.

[28] Wang Y K, Feng B Y, Ding Y F. QGTC: Accelerating quantized graph neural networks via GPU Tensor core[C]//Proc of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2022: 107-119.

[29] Crankshaw D, Wang X, Zhou G, et al. Clipper: A low-latency online prediction serving system[C]//Proc of the 14th USENIX Conference on Networked Systems Design and Implementation, 2017: 613-627.

[30] Zhang C L, Yu M C, Wang W, et al. MArk: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving[C]//Proc of USENIX Annual Technical Conference, 2019: 1049-1062.

[31] Gujarati A, Karimi R, Alzayat S, et al. Serving DNNs like clockwork: Performance predictability from the bottom up[C]//Proc of USENIX Symposium on Operating Systems Design and Implementation, 2020: 443-462.

作者简介:

孙庆骁(1996-),男,黑龙江大庆人,博士,副教授,CCF 会员(97024G),研究方向为高性能计算。**E-mail**: qingxiaosun@buaa.edu.cn

SUN Qing-xiao, born in 1996, PhD, associate professor, CCF member (97024G), his research interest includes high performance computing.

刘轶(1968-),男,河北安新人,博士,教授,CCF 会员(14335D),研究方向为高性能计算。**E-mail**: yi.liu@buaa.edu.cn

LIU Yi, born in 1968, PhD, professor, CCF member (14335D), his research interest includes high performance computing.

杨海龙(1985-),男,山东曹县人,博士,副教授,CCF 会员(43790M),研究方向为高性能计算。**E-mail**: hailong.yang@buaa.edu.cn

YANG Hai-long, born in 1985, PhD, associate professor, CCF member (43790M), his research interest includes high performance computing.

王一晴(2001-),女,山东济南人,博士生,CCF 会员(J0979G),研究方向为高性能计算。**E-mail**: 19373408@buaa.edu.cn

WANG Yi-qing, born in 2001, PhD candidate, CCF member (J0979G), her research interest includes high performance computing.

贾婕(1995-),女,河北沧州人,博士生,CCF 会员(O4147G),研究方向为高性能计算。**E-mail**: jj@buaa.edu.cn

JIA Jie, born in 1995, PhD candidate, CCF member (O4147G), her research interest includes high performance computing.

栾钟治(1971-),男,山东青岛人,博士,副教授,CCF 会员(07769S),研究方向为高性能计算。**E-mail**: zhongzhi.luan@buaa.edu.cn

LUAN Zhong-zhi, born in 1971, PhD, associate professor, CCF member(07769S), his research interest includes high performance computing.