華中科技大學

本科生毕业设计

这是标题这是标题这是标题

院　　系　计算机科学与技术

专业班级　计科 2108

姓　　名　金钊

学　　号　U202115568

指导教师　刘海坤

2025 年 13 月 32 日

学位论文原创性声明

本人郑重声明：所呈交的论文是本人在导师的指导下独立进行研究所取得的研究成果。除了文中特别加以标注引用的内容外，本论文不包括任何其他个人或集体已经发表或撰写的成果作品。本人完全意识到本声明的法律后果由本人承担。

作者签名：　　2025 年 13 月 32 日

学位论文版权使用授权书

本学位论文作者完全了解学校有关保障、使用学位论文的规定，同意学校保留并向有关学位论文管理部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权省级优秀学士论文评选机构将本学位论文的全部或部分内容编入有关数据进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。

本学位论文属于 1、保　密 □，在　　年解密后适用本授权书

2、不保密 ☑。

（请在以上相应方框内打 “✓”）

作者签名：　　2025 年 13 月 32 日

导师签名：　　2025 年 13 月 32 日

摘 要

随着图神经网络（GNN）在图结构数据深度学习中的兴起，GNN 因其生成高质量节点特征向量（嵌入）的能力而备受关注。然而，现有的“一刀切”式 GNN 实现难以应对不断演化的 GNN 架构、日益增长的图规模以及多样化的节点嵌入维度，导致性能无法令人满意。为此，我们提出了 GNNA，一个面向 GPU 平台、具备自适应性和高效性的运行时系统，用于加速多种 GNN 工作负载。首先，GNNA 从 GNN 模型和输入图中提取并识别出多个与性能相关的特征，并将其作为 GNN 加速的新驱动因素。其次， GNNA 实现了一种新颖且高效的二维工作负载管理机制，专为 GNN 计算优化，以提高在不同应用场景下的 GPU 利用率和性能。第三， GNNA 利用 GPU 的内存层级结构，通过协调 GNN 执行与 GPU 内存结构及 GNN 工作负载的特性，实现加速。此外，为实现自动运行时优化， GNNA 集成了一个轻量级分析模型，用于高效地搜索设计参数。大量实验证明，GNNAdvisor 在主流 GNN 架构和多种数据集上，相比当前最先进的 GNN 计算框架，如 Deep Graph Library（平均快 2.24×）和 Python Geometric（平均快 2.83×），具有更优的性能。

关键词：图神经网；GPU 加速；工作负载管理；内存层级结构。

Abstract

As the emerging trend of graph-based deep learning, Graph Neural Networks (GNNs) excel for their capability to generate high-quality node feature vectors (embeddings). However, the existing one-size-fits-all GNN implementations are insufficient to catch up with the evolving GNN architectures, the ever-increasing graph sizes, and the diverse node embedding dimensionalities. To this end, we propose **GNNA**, an adaptive and efficient runtime system to accelerate various GNN workloads on GPU platforms. First, GNNAdvisor explores and identifies several performance-relevant features from both the GNN model and the input graph, and uses them as a new driving force for GNN acceleration. Second, GNNAdvisor implements a novel and highly-efficient 2D workload management, tailored for GNN computation to improve GPU utilization and performance under different application settings. Third, GNNAdvisor capitalizes on the GPU memory hierarchy for acceleration by gracefully coordinating the execution of GNNs according to the characteristics of the GPU memory structure and GNN workloads. Furthermore, to enable automatic runtime optimization, GNNAdvisor incorporates a lightweight analytical model for an effective design parameter search. Extensive experiments show that GNNAdvisor outperforms the state-of-the-art GNN computing frameworks, such as Deep Graph Library (2.24× faster on average) and Python Geometric (2.83× faster on average), on mainstream GNN architectures across various datasets.

Keywords: Graph Neural Networks, GPU acceleration, Workload management, Memory hierarchy

目 录

1 绪 论

本章我们首先介绍了当前图神经网络（Graph Neural Network, GNN）的技术发展趋势，然后分析了 GNN 的现有实现所存在的问题，介绍了国内外在 GNN 加速领域的相关研究工作，并对本文的主要研究内容及工作意义作了具体说明。

1.1 课题背景、目的与意义

1.1.1 研究背景与趋势

图结构数据（Graph-structured Data）在现实世界中无处不在，从社交网络、蛋白质相互作用网络、知识图谱到交通流网络和金融交易网络。如何有效地从这些图结构数据中学习和提取有价值的知识，已成为机器学习和数据挖掘领域的核心挑战之一。在此背景下，图神经网络（Graph Neural Network, GNN）应运而生，并迅速成为处理图数据的强大范式。GNN 通过迭代地聚合邻居节点信息来更新节点表示，能够有效地捕捉图的拓扑结构和节点特征，在节点分类、链接预测、图分类等多种任务中取得了突破性进展，并在推荐系统、药物发现、自然语言处理、计算机视觉等众多领域展现出巨大的应用潜力。

与传统图分析方法（如随机游走[1-2]和图拉普拉斯方法[3-4]）相比，图神经网络通过独特的“聚合-更新”双阶段交替执行机制脱颖而出：在“聚合”阶段执行图操作（如分散-收集，scatter-gather），在“更新”阶段执行神经网络操作（矩阵乘法），从而显著提升准确率并增强泛化能力。然而，GNN 的独特机制也为 GNN 框架的设计与实现带来了许多问题和挑战。现有支持 GNN 训练与推理的研究可分为两类：第一类基于图处理系统扩展神经网络功能，第二类则从深度学习框架扩展图操作支持。但这些方案仍存在诸多局限性，如对图数据的支持不足、对 GNN 模型的支持不全面，不能有效利用 GNN 的运行时信息进行优化，导致在处理大规模图数据时性能瓶颈明显，在面对不断增大的图规模和多样化的节点嵌入维度时表现欠佳。

图形处理器（Graphics Processing Units, GPU）凭借其大规模并行计算能力和高内存带宽，已从最初的图形渲染专用硬件发展成为通用并行计算（General-

Purpose computing on GPUs, GPGPU）的主流平台，尤其在深度学习领域取得了巨大成功。GPU 的架构特点使其非常适合处理计算密集型和数据并行的任务。本课题选取 GPU 作为研究对象，旨在通过对 GPU 架构的深入分析，探索其在图神经网络加速中的应用潜力。本课题将重点关注 GPU 的并行计算能力、内存访问模式和负载划分等方面，为 GNN 的高效实现提供新的思路和方法。

1.1.2 面临的问题和挑战

尽管 GPU 为 GNN 计算提供了强大的硬件基础，但要充分利用硬件的计算能力，将 GNN 高效地映射到 GPU 架构上，仍然面临一系列严峻的挑战：

GNN 的计算负载混合且高度不规则：GNN 包含聚合和更新两阶段。其中，聚合阶段是性能的关键瓶颈，其计算模式具有高度的不规则性：

- 内存访问不规则：访问邻居节点的特征数据通常是非连续、随机的，这与 GPU 内存系统为优化连续、规整访问（Coalesced Access）而设计的机制相冲突，导致内存带宽利用率低下，缓存效率差。
- 计算量不规则：图中节点度数的幂律分布[5] 导致不同节点的邻居聚合计算量差异巨大。将节点或边直接映射到 GPU 线程或线程块进行处理，会引发严重的负载不均衡问题。部分处理高密度节点的线程或块成为性能瓶颈，而大量处理低密度节点的线程或块则提前完成并空闲，极大地降低了 GPU 的整体利用率[6]。

GPU 并行执行模型的挑战：GPU 的 SIMT 执行模型要求一个 Warp（通常 32 个线程）中的所有线程执行相同的指令。然而，在 GNN 计算中，由于负载不均衡或需要根据节点或边属性执行不同的处理逻辑（例如，处理不同类型的边或根据节点度选择不同聚合策略），容易导致同一 Warp 内的线程执行不同的代码分支，产生 Warp 发散。这会使部分线程被暂时屏蔽，降低了计算资源的有效利用率。

在 GNN 的聚合阶段，多个线程可能需要更新同一个目标节点的累积特征值，这通常需要使用原子操作（Atomic Operations）来保证数据一致性。在具有数千个并发线程的 GPU 上，高并发的原子操作会产生显著的竞争和串行化，成为性能瓶颈，尤其是在高节点度的情况下。

现有 GNN 系统与框架的局限性：当前许多 GNN 框架[7-8] 依赖通用的稀疏计算库（如 torch-scatter[9]）或标准的稀疏矩阵运算库（如 cuSPARSE 中的 SpMM[10]）来实现图聚合操作。这些通用计算库往往没有针对 GNN 特有的数据访问模式和计算流程进行深度优化，导致性能不理想。

最优的 GPU 执行策略（如线程映射方式、共享内存使用策略、数据布局等）强烈依赖于具体的 GNN 模型、嵌入维度大小以及输入图的拓扑特性（如度分布、社区结构等）。GNN 系统与框架如何感知输入特性并自动调整优化策略，在不同应用场景下应用不同的优化，充分利用 GPU 的计算资源，以达到最佳效果，是学界和工业界亟待解决的难题。

1.1.3 课题目的与意义

本课题的主要研究目的在于设计并实现一个面向 GPU 平台的高效、自适应的 GNN 计算加速运行时系统： GNNA。 GNNA 利用 PyTorch 作为前端，以便于用户使用和集成。在底层， GNNA 使用 C++/CUDA 构建，通过 Python 绑定到 PyTorch，提供高效的 GNN 计算内核。数据由 Pytorch 编写的数据加载器加载，并以张量形式传递给 GNNA，以便在 GPU 上进行计算。一旦 GNNAd-visor 在 GPU 上完成计算，它就会将数据张量（tensor）传回原始 Pytorch 框架进行进一步处理。如图 1-1所示， GNNA 由几个关键组件组成，以促进 GNN 优化和在 GPU 上的执行。首先， GNNA 引入了一个输入加载器和提取器（input Loader & Extractor），以利用输入级信息来指导我们的系统级优化。其次， GNNA 引入了一个由分析模型组成的 Decider，用于自动选择运行时参数，以减少设计优化过程中的人工操作，并加入了一个轻量级节点重编号例程，以提高图结构的局部性。第三， GNNA 集成了内核与运行时创建（Kernel&Runtime Crafter），用于定制参数化 GNN 内核和 CUDA 运行时，其中包括高效的二维负载管理（同时考虑邻节点数量和节点嵌入维度）和一套 GNN 专用内存优化。

本研究的顺利完成具有重要的理论价值和显著的实际应用意义。最直接地，它有望显著提升 GNN 在 GPU 上的计算效率，大幅缩短训练和推理时间。此外，虽然本课题侧重于 GNN 的加速，但是提出的自适应优化机制和负载均衡策略也可以推广到其他图计算任务和深度学习模型中，具有广泛的适用性和参考价值。

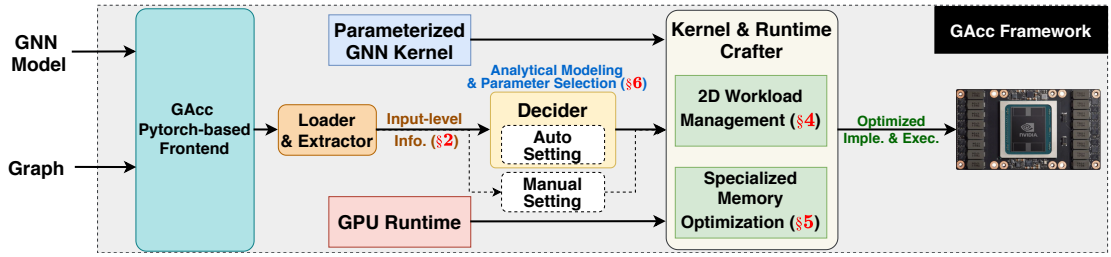


图 1-1　GNNA 的整体架构

通过深入研究 GNN 在 GPU 上的实现，本课题将为未来更高效的图计算框架和算法设计提供有益的启示。

1.2　国内外研究现状

1.2.1　通过算法加速 GNN

GNN 训练加速的核心思想在于减少计算图的规模。同时，理想的加速训练方法应当能够在模型性能上达到与未经加速的常规训练方法相近的结果。在本节中，我们讨论两类主流的 GNN 训练加速技术：图修改（graph modification）和采样（sampling）。这两种方法都旨在通过缩减计算图来加速训练过程。它们的主要区别在于是否显式地生成一个修改后的图作为中间输出。

图修改通过两个步骤加速 GNN 训练。第一阶段处理原始图，输出一个规模更小、能够被快速处理和训练的新图。第二阶段则以这个修改后的图作为输入，进行常规的 GNN 训练。接下来本文将介绍三类图结构修改方法：图粗化（graph coarsening）、图稀疏化（graph sparsification）和图压缩（graph condensation）。这些方法通过不同策略生成各异的图，但其核心目标均为构建规模更小的训练图以提升 GNN 训练效率。图 1-2直观展示了这些图修改方法的实现原理。

- 图粗化，是一种减小图大小同时保留其整体结构的技术。通过将相同局部结构中的节点合并为“超级节点”，并将连接超级节点的边合并为“超级边”，可以得到粗图。图粗化的核心步骤是图聚类，通常与图谱相关。一些算法如受限谱近似[11] 和逆拉普拉斯[12]，被用于保留一些图属性。
- 图稀疏化，通过删除冗余边来减少计算图，加速 GNN 训练。现有方法通常保留原始图的关键属性，如切割总权重、光谱特性和分层结构。稀疏化算法

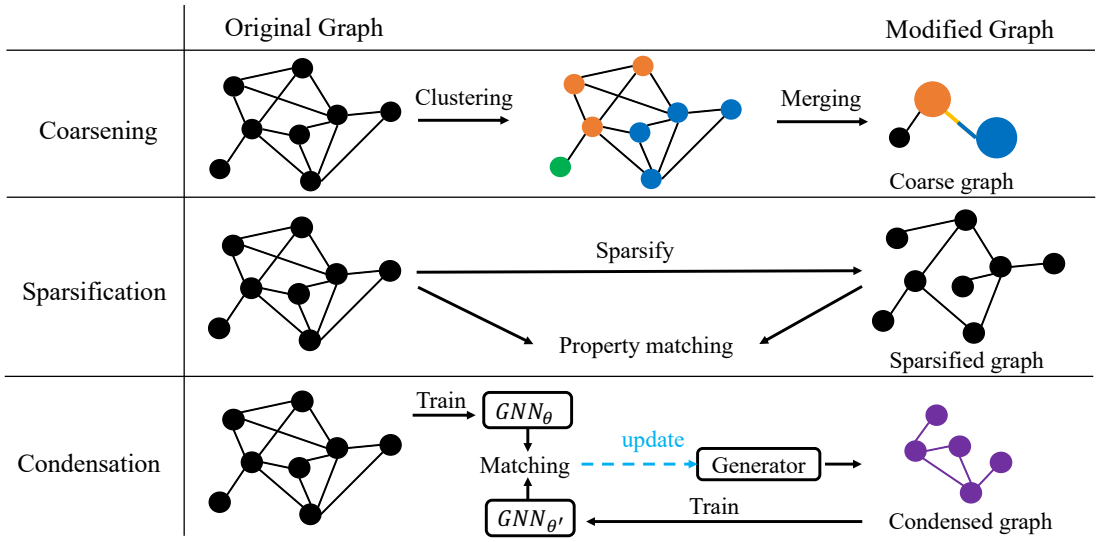


图 1-2　图修改方法图示：*图粗化* (Graph Coarsening) 方法执行图聚类并将节点簇合并成超节点。*图稀疏化* (Graph Sparsification) 方法移除不太重要的边。*图压缩* (Graph Condensation) 方法使用一个随机初始化的生成模型来生成一个新的压缩图。对于修改后的图（最右列），黑色节点/边来自原始图，彩色节点/边是新创建的。

可作为预处理步骤，降低全批次时间复杂度。现有图疏解方法保留的属性示例包括切分的总权重[13]、谱属性[14-15]和层次结构[16]。一些相关工作执行基于 GNN 的图稀疏化以提高 GNN 精度或鲁棒性，但这些方法通常需要解决额外的优化问题，可能减慢训练速度。

- 图压缩，通过匹配两个 GNN 的训练梯度来生成一个压缩图，从而减少 GNN 的计算量和内存占用。GCond[17] 是一种具有代表性的用于生成压缩图的方法，可以在保留原始图训练动态的同时显著减小图的大小。通过在原始图和压缩图上训练两个具有相同架构的 GNN，GCond 可以生成压缩图，同时保持原始 GNN 的性能。实验表明，GCond 可以在多个图基准上实现小于 1% 的压缩比，同时保持原始 GNN 超过 95% 的准确度。因此，GCond 非常适合神经架构搜索等任务。然而，为了加速单个 GNN 的训练，GCond 应该与其他加速方法或更好的压缩策略相结合。

图采样每次训练迭代中动态地选择一部分节点或边的子集，用以构建规模更小的计算图。需要注意的是，采样与图形修改不同，采样是动态且隐式的，没有中间修改的图形输出。采样方法分为节点方式、逐层和子图方式三类，图1-3这

三种采样算法的工作流程。

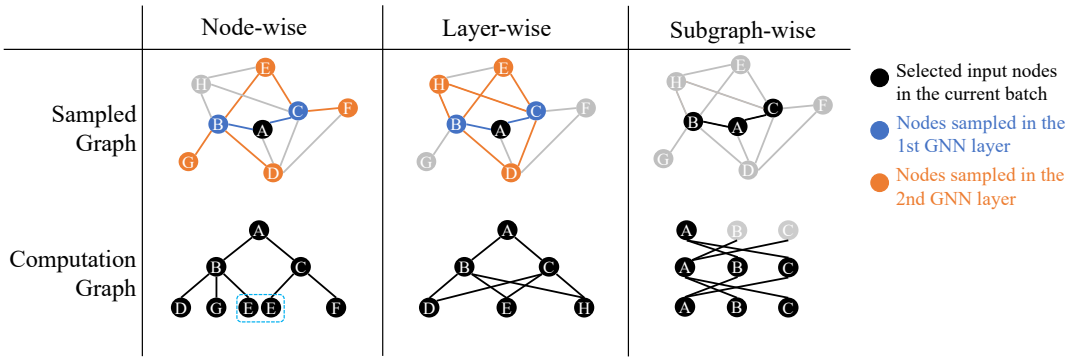


图 1-3　图采样方法的示意图。*节点级采样*（Node-wise Sampling）方法为每个计算层的每个节点单独采样，可能导致节点冗余（如节点 E 被重复采样两次）和边缺失（如节点 C 与 D 之间的边缺失）；*层级采样*（Layer-wise Sampling）方法基于前一层的节点进行逐层采样；*子图级采样*（Subgraph-wise Sampling）方法通过采样节点列表及其诱导子图，在采样得到的子图内部沿所有边进行消息传递。

- *节点方式*，通过选择一部分节点来构建一个子图。该方法的关键在于如何选择节点。现有方法通常基于节点的特征或连接性进行选择。GNN 算法加速中的节点采样方法，旨在通过减少聚合操作中涉及的邻居节点数量来降低计算复杂度。GraphSAGE[18] 是该领域的开创性工作。它在 GNN 的每一层为每个目标节点从其全部邻居中均匀地采样固定数量的节点，构成采样后的邻居集。GraphSAGE 使用这个较小的邻居集来进行聚合计算，而不是使用全部邻居，从而显著减少了计算开销，加速了 GNN 的处理过程。重要的是，通过调整采样节点贡献的权重，这种采样聚合是对使用全邻居聚合结果的无偏估计，保证了采样方法的理论正确性。
- *逐层方式*，通过逐层地选择节点和边来构建子图。该方法通常使用随机游走或邻居采样等技术来选择节点和边。FastGCN[19] 首次提出了分层重要性采样方案，以解决分节点采样的可扩展性问题。AS-GCN[20] 提出了一种自适应分层采样方法，以加速大规模图上 GCN 的训练。该方法以顶层为条件对下层进行采样，采样邻域由不同的父节点共享，避免了过度扩展。
- *子图方式*，通过选择一部分子图来构建一个新的计算图。由于子图通常较小且只表示局部信息，训练时需要添加随机性以提高准确性。这些方法与图修改方法类似，但并非真正修改图，而是动态采样。GraphSAINT[21] 是一种

广泛使用的子图采样方法，它首先对节点进行采样，然后构建由采样节点诱导的子图。GraphSAINT 提出了四种不同的节点采样算法。在所提出的四种算法中，对随机节点集进行随机漫步，并在漫步中选择节点的随机漫步采样器的经验表现更好，因此得到了更广泛的应用。同直接在节点级别采样的 GraphSAGE 相比，构建子图的过程引入了额外的开销。ClusterGCN[22] 采用图聚类算法（如 METIS[23]）将整个输入图划分为多个簇。每个簇内的节点构成连接紧密的子图，而簇间的边则被最小化。随后，ClusterGCN 在每个子图簇上进行全批量 (full-batch) GNN 训练。这种方法的一个局限性是聚类结果是固定的，导致训练过程中会丢失簇间的边。

1.2.2 通过现有硬件系统加速 GNN

除了高效的 GNN 训练/推理算法，优化底层系统对于提高 GNN 的端到端吞吐量至关重要。现有工作从三个方面加速 GNN 系统：GPU 内核加速、用户定义函数（UDF）优化和可扩展训练系统。由于 GNN 计算遵循消息传递范式，需要高效的稀疏操作，人们提出了各种高效的 GNN 内核和 UDF 优化技术。如何设计一个高效的可扩展 GNN 训练系统，仍然是机器学习系统界的一个开放研究问题。

- *GPU 内核加速*。GPU 作为高性能硬件加速器，可显著提升深度学习训练与推理中的计算效率。经过针对性优化的 GPU 内核（GPU kernels）GPU 已广泛应用于深度学习加速领域，但由于图数据固有的稀疏性（sparsity）和不规则性（irregularity），利用 GPU 加速图神经网络（GNN）仍面临显著挑战，难以充分利用硬件的 GPU 的性能。现有工作通过各种优化技术来提升 GNN 在 GPU 上的计算效率，如 TLPGNN[24] 开发了两级并行：节点级并行和特征级并行，为例均衡工作量均衡的工作量，开发了一种动态工作量分配，一旦释放了一个硬件资源，就会分配下一个计算任务。通过负载重排序和动态负载分配集中内存访问并减少了分支发散。PCGCN 通过利用图中节点通常呈现聚集分布的独特稀疏模式来提升了 GNN 计算的数据局部性。该方法采用子图级处理策略，通过将 GNN 计算负载按子图划分来集中内存访问，优化数据访问效率。此外，PCGCN[25] 创新性地引入双模式计算机制：

对稀疏子图采用稀疏矩阵乘法（SpMM），而对稠密子图则使用通用矩阵乘法（GeMM）。但该方法其依赖图聚类算法进行预处理分区，可能引入额外的计算开销。

- 用户定义函数优化。用户定义函数（User-Defined Function，简称 UDF）指的是允许用户使用高级语言（通常是 Python 结合 PyTorch、TensorFlow、JAX 等框架）来自定义核心计算逻辑的编程接口，最典型的就是消息传递（Message Passing）框架中的消息函数（Message Function）、聚合函数（Aggregation/Reduce Function）和更新函数（Update Function）。UDF 为研究人员和开发者提供了灵活性和可扩展性，也为 GNN 的优化带来了困难和挑战。目前，业界和学术界主要采取多种方案优化 GNN 中的 UDF。主流的深度学习框架如 Pytorch[26]，通过动态修改 Python 字节码，并将连续的 PyTorch 操作序列提取为 FX 计算图[27]，随后通过可扩展的后端进行 JIT 编译。TorchDynamo[28] 作为默认编译后端，能够将 PyTorch 程序转换为面向 GPU 的 OpenAI Triton[29] 代码以及面向 CPU 的 C++ 代码。但是这种通用的优化手段面对 GNN 优化效果不佳，GNN 的动态输入规模为 JIT 编译带来了巨大挑战。Seastar[30] 提出了一种面向用户自定义函数（UDF）的顶点中心式编程接口，并通过生成优化内存消耗与数据局部性的执行计划，显著提升计算性能。具体来说，Seastar 首先通过一个追踪机制将顶点中心（vertex-centric）的逻辑转换为张量操作，然后基于其定义的 Seastar 计算模式来识别算子融合的机会。这种方法借鉴了顶点中心编程的优点，通过融合每个目标节点上的相关操作，有效降低了内存消耗并增强了数据局部性。

1.2.3 通过定制硬件加速 GNN

近年来，对图神经网络 (GNN) 日益浓厚的兴趣推动了定制化加速器（如 FPGA 或 ASIC）的研发。尽管 GNN 在网络架构上与卷积神经网络 (CNN) 存在相似之处，但它们在计算复杂度和通信模式上的显著差异，使得众多现有的 CNN 加速器[31-32] 难以直接高效地应用于 GNN。此外，由于 GNN 模型种类繁多，每种模型都有其独特的通用性（灵活性）、可扩展性和专用性之间的权衡特性，以实现特定用的最佳性能。这种权衡会影响加速器可达到的峰值性能和通用性。随

着工作应用范围的缩小，会出现更多定制加速器的机会，从而提高加速器的性能，尽管这要以牺牲适应灵活性为代价。这些加速器主要用于部署，侧重于推理而非训练。训练过程中不断变化的网络架构给调整硬件设计或要求再生和或重新配置带来了挑战。因此，专门为训练阶段设计的定制加速器[33-34] 并不多见。

- 用于通用负载的加速器: 为了满足处理多种不同 GNN 算法的需求，研究人员提出了一类面向通用负载的加速器，它们旨在提供较好的灵活性和普适性。这些加速器通常采取两种主要设计策略：一是构建一个统一的硬件架构来处理 GNN 计算中的所有主要阶段；二是根据不同计算阶段（如聚合、更新）的独特计算和通信模式，为其开发专门的处理引擎。例如，EnGN[35] 采用了一种统一的神经图处理单元（NGPU），该单元基于脉动阵列，将特征提取、聚合、更新三个阶段处理为流水线化的矩阵乘法。为了处理稀疏聚合，它提出了一种特定的“环状边规约”数据流，并通过重组边、缓存高阶节点和采用维度感知的阶段重排序等优化手段来提升效率。相比之下，HyGCN[36] 则为聚合和组合（更新）阶段设计了独立的、专门的处理引擎，并以数据流方式协同工作。其聚合引擎利用并行的 SIMD 核来处理长特征向量和图稀疏性（通过稀疏性消除器和采样技术），而组合引擎则使用脉动阵列组来高效执行稠密计算。
- 用于专用负载的加速器另一类硬件加速器则选择专注于特定的、广泛使用的 GNN 算法，其中以图卷积网络（GCN）最为常见，从而能够进行更深层次的微架构定制和优化。这类工作同样展现出不同的设计思路：一部分研究构建了层间可定制的深度流水线架构，旨在最大化利用层间并行性并减少全局内存访问；另一部分则倾向于设计一个适用于所有层的统一硬件架构，通过算法与硬件的协同优化来提升效率。例如，StreamGCN[37] 是为流式处理小规模 GCN 图而设计的，它采用了深度流水线结构，为每一层分配专用模块以实现层间流水线作业，从而极大地减少了对 DRAM 的访问需求。该设计还包含了边预处理和重排序以避免数据依赖，并实现了运行时的稀疏性支持，能够即时修剪掉零值嵌入。而 GraphACT[33] 则专注于在 CPU-FPGA 异构平台上加速 GCN 的训练过程（特别是针对小图）。它并没有直接处理归一化的邻接矩阵，而是定义了三种 GCN 特有的乘法操作，并设计了可重

用的权重变换模块（基于 2D 脉动阵列）和特征聚合模块（基于 1D 累加器阵列）来执行这些操作。同时，它利用 CPU 进行预处理，通过识别和合并重复的邻居对来减少冗余计算。这些针对特定算法的加速器通过更精细的定制，往往能在目标任务上实现更高的性能和能效。

1.3 论文的主要内容与结构

1.3.1 论文的主要内容

本文的主要研究内容包括：

1) 介绍了 GNNA 的总体架构，一个面向 GPU 的 GNN 计算加速运行时系统。
2) 设计了输入加载器和提取器，利用输入信息来指导系统级优化。
3) 使用 Rabbit 重排序算法来提高图结构的局部性。
4) 设计邻居组和嵌入维度为基础的二维工作负载管理机制。
5) 设计了基于线程束的负载映射和共享内存分配机制。
6) 对比分析了 GNNA 与现有 GNN 加速器的性能。

1.3.2 论文结构

本文的主要内容如下：

第一章我们首先介绍了目前图神经网络（GNN）在图计算中的重要性和应用背景，阐述了 GNN 在 GPU 上的加速需求和面临的挑战。接着，我们回顾了现有的 GNN 加速方法，包括算法级别的优化、系统级别的优化和硬件级别的优化。最后，我们介绍了本课题的研究目的和意义。

第二章介绍了 GNNA 的相关技术基础，包括图神经网络的基本概念、GPU 计算模型、CUDA 编程模型、图处理系统和深度学习框架。我们还介绍了 GNNA 的设计理念和架构。

第三章我们介绍了 GNNA 的设计与实现，包括 GNNA 的系统架构、对 GNN 模型的输入分析、二维工作负载管理机制和调度策略，专用内容优化技术和优化指标的量化实现。

第四章我们介绍了 GNNA 的性能评估，包括实验环境的搭建、数据集的选

择、性能评估指标的设计和实验结果的分析。我们还与现有 GNN 加速器进行了对比分析。

第五章我们总结了本文的研究工作，分析了 GNNA 的优缺点，并提出了未来的研究方向。

2 相关技术基础

在本节中，我们将介绍图神经网络（GNN）和图形处理单元（GPU）的基础知识以及两种主要的 GNN 计算框架：基于 GPU 的图系统和深度学习框架。

2.1 图神经网络

图 2-1展示了图神经网络在单次迭代中的计算流程。GNN 再 $k+1$ 层基于第 k 层（$k \geq 0$）的节点嵌入信息，如公式 2-1所示：

$$\begin{aligned} a_v^{(k+1)} &= \textbf{Aggregate}^{(k+1)}(h_u^{(k)} | u \in \mathbf{N}(v) \cup h_v^{(k)}) \\ h_v^{(k+1)} &= \textbf{Update}^{(k+1)}(a_v^{(k+1)}) \end{aligned} \tag{2-1}$$

其中，$h_v^{(k)}$ 表示节点 v 在第 k 层的嵌入向量；$h_v^{(0)}$ 通过仅用于符号值嵌入空间映射的初始嵌入函数，由节点特定任务特征（如顶点关联的文本或顶点所表征实体的标量属性）计算得出；$a_v^{(k+1)}$ 是通过聚合邻居信息（如节点嵌入）得到的聚合结果；$\mathbf{N}(v)$ 表示节点 v 的邻居集合。不同图神经网络在聚合方法与更新顺序上存在差异：部分方法[38-39] 只依赖相邻节点信息，而另一些方法[40] 还会结合边属性——通过计算每条边两端节点嵌入的点积，并融合边特征（边类型及其他属性）来实现。更新函数通常由标准神经网络操作构成，例如采用 $w \cdot a_v^{(k+1)} + b$ 形式的全连接层或多层感知机（MLP），其中 w 和 b 分别为可学习的权重和偏置参数。节点嵌入向量的维度常见取值为 16、64 和 128，不同层间的嵌入维度可能发生变化。

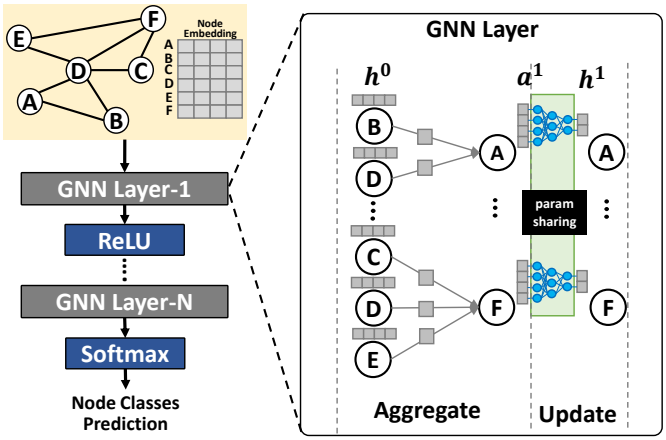


图 2-1 GNN General Computation Flow.

在经历多次聚合与更新的迭代过程（即多轮 GNN 层运算）后，我们将获得各节点的输出嵌入向量。该向量通常可用于基于图的深度学习下游任务，例如节点分类任务[41-43] 和链接预测任务[44-46]。需特别说明的是，GNN 输入层的初始节点嵌入既可以原始图数据集自带特征，还可以通过图嵌入算法生成（例如[1,47-48]）。这类初始嵌入的生成过程独立于 GNN 模型的计算流程（即不参与隐藏层及输出层节点嵌入的计算）。

2.2 图形处理器

图形处理器（GPU）最初为加速图形渲染而设计，但其架构内在的大规模并行计算能力和高内存带宽，使其迅速发展成为通用并行计算（General-Purpose computing on GPUs, GPGPU）的主流平台，尤其在深度学习、科学计算等计算密集型领域取得了革命性的成功。对于同样有着巨大计算需求的图神经网络（GNN）而言，GPU 自然成为了极具吸引力的加速硬件选择。

图形处理器（GPU）通过其大规模并行架构为 GNN 计算提供了强大的硬件基础。GPU 包含数千个计算核心，组织在多个流式多处理器（SMs）中，并配备高带宽内存系统，以支持高吞吐量计算。要充分发挥其性能，关键在于有效利用 GPU 复杂的内存层次结构：这包括高速的寄存器和 SM 内部可编程的共享内存（Shared Memory），后者允许同一线程块内的线程高效协作。优化目标是最大化利用片上高速内存，并实现对全局内存的合并访问（Coalesced Access），即 Warp 内线程访问连续内存地址，以提升带宽利用率。

编程 GPU 通常采用如 NVIDIA CUDA 之类的模型。CUDA 执行并行内核（Kernel），其执行单元是线程（Thread）。线程被组织成线程块（Block），块内线程可共享数据并同步。多个块组成线程网格（Grid）。硬件层面，线程以 Warp（通常 32 个）为单位进行调度和执行，遵循单指令多线程（SIMT）模式。然而，GNN 的不规则性给这种架构带来了挑战：图遍历导致的不规则内存访问难以合并；节点度的巨大差异造成负载不均衡；条件分支引发 Warp 发散，降低 SIMT 效率；聚合更新所需的原子操作则可能产生竞争瓶颈。因此，高效的 GNN 加速需要深入结合 GNN 的模型信息和 GPU 的硬件架构，根据不同的模型和硬件平台自适应做出深度优化。

2.3 图处理系统

当前已有诸多图处理系统[49-53]致力于加速传统图算法。这些系统普遍采用顶点与节点编程抽象和边中心处理范式，并通过系统级优化来缓解计算不规则性（如负载不均衡）和内存访问不规则性（如非合并的全局内存访问）等问题。然而，将这些图处理系统扩展应用于 GNN 计算时却面临显著挑战。

首先，传统图处理中常见的算法优化策略对 GNN 往往收效甚微。以广度优先搜索为代表的图遍历算法依赖于对前驱节点（即活跃邻居）的迭代计算，因此催生了推拉式遍历[52-53]和前驱过滤[51-53]等优化技术。但 GNN 在迭代过程中始终需要处理每个节点的全部邻居，这种固定规模的前驱访问特性使得传统前驱优化技术失去用武之地。

其次，图处理系统的优化技术必须经过针对性改造才能适配 GNN 特性。现有系统采用的节点与边处理机制[51,53]和基于分片的图表示方法[49]原本是针对单标量属性节点的优化方案。而 GNN 引入了高维嵌入向量这一新的并行维度，这就要求系统设计需要在更细粒度上重新权衡维度级并行性与节点嵌入局部性，而非简单沿用传统的节点级并行优化策略。

更为关键的是，现有图处理系统普遍缺乏支持 GNN 计算的核心功能模块。无论是基于神经网络的前向传播节点更新，还是复杂的反向梯度传播机制，在当前主流图系统[49-55]中均未实现。反观 PyTorch[56]和 TensorFlow[57]等深度学习框架，其内置的自动微分功能可灵活支持各类模型架构的梯度计算。这种功能代差使得直接扩展图处理系统支持 GNN 面临巨大技术障碍，也促使我们选择基于深度学习框架开发全新的系统。

2.4 深度学习框架

目前学界和工业界已有提出了多种神经网络框架，包括 TensorFlow[57]和 PyTorch[56]等。这些框架为传统深度学习模型提供了端到端的训练与推理支持，涵盖线性算子、卷积算子等多种神经网络算子。然而，这些算子主要针对图像等欧式数据进行了高度优化，缺乏对 GNN 中非欧式图数据的原生支持。要将这些神经网络框架扩展以支持处理高度不规则图数据的 GNN 计算，主要面临许多挑

战。

首先，现有基于神经网络框架扩展的 GNN 计算平台[7-8] 虽然注重不同 GNN 模型的编程通用性，但其底层运行时缺乏高效支持。以 PyTorch Geometric（PyG）[7] 为例，其采用 CUDA 实现的 torch-scatter[9] 库作为图聚合操作的核心组件。该实现在处理中小规模稠密图时表现尚可，但当面对具有高维节点嵌入的大规模稀疏图时，由于内核设计沿用了图处理系统理念，过度依赖高开销的原子操作来支持节点嵌入传播，导致性能急剧下降。类似的可扩展性问题也出现在 Deep Graph Library（DGL）[8] 中：该系统对简单的求和聚合[38-39] 直接调用 cuSparse[10] 的稀疏矩阵乘法（如 *csrmm2*），而对涉及边属性的复杂聚合[40,58] 则采用自研 CUDA 内核，均存在性能瓶颈。

其次，主流的计算内核[7-8] 采用硬编码实现，缺乏设计灵活性。从高层接口来看，用户仅能定义这些内核的外部组合方式，而无法根据已知的 GNN 模型架构特征、GPU 硬件特性和图数据属性，对这些内核进行内部定制化优化。这种刚性设计难以适应不同规模图数据与节点嵌入维度的多样化应用场景，仍然存在巨大的优化空间。在 Pytorch 2.0 中引入的 torch.compile[26] 通过即时编译（Just-In-Time Compilation，JIT）技术，允许用户在运行时对计算图进行优化。但是，主流的计算内核（如 DGL[8]）的许多核心功能依赖自定义 C++/CUDA 内核的操作，这些内核在编译时引发的图中断裂（Graph Breakage）会导致 JIT 优化失效，此外，这种通用的编译优化方法在处理 GNN 计算时仍然难以充分发挥 GPU 的硬件性能。

2.5 本章小结

本章旨在为后续深入探讨 GNN 加速技术奠定基础。我们首先阐述了图神经网络（GNN）的基本计算范式，包括其核心的聚合与更新流程，以及节点嵌入在多种下游任务中的关键作用。随后，深入剖析了图形处理器（GPU）的并行计算架构和内存体系，分析了其作为 GNN 加速硬件平台的巨大潜力，同时也指出了 GNN 固有的计算和访存不规则性给 GPU 高效利用带来的挑战。进一步地，本章回顾了传统的图处理系统，分析了其设计哲学和优化手段，并论证了这些系统难以直接、高效地扩展以支持 GNN 计算的根本原因，尤其是在处理高维特征向

量和缺乏必要的神经网络及自动微分功能方面。最后，我们审视了当前主流深度学习框架在集成 GNN 支持时所面临的性能瓶颈，特别是底层稀疏计算内核的效率问题和硬编码实现带来的灵活性缺失，并初步探讨了如 torch.compile 等编译优化技术在这一背景下的潜力和局限。通过对这些基础技术及其相互作用的理解，为后续章节详细介绍针对 GNN 的算法、系统及硬件层面的加速策略提供了必要的背景知识和技术铺垫。

3 基于 GPU 的 GNN 加速算法设计

本章节中我们将会介绍 GNNA 的设计思路和实现细节。我们将从 GNN 模型的输入分析开始，接着介绍二维工作负载管理和专用内存优化，最后介绍设计优化。我们将会详细介绍每个部分的设计思路和实现细节。

3.1 GNN 模型的输入分析

在本小节中，我们基于一个关键观察提出论点：GNN 输入信息可以指导系统优化，因为不同的 GNN 应用设置会倾向于不同的优化选择。我们介绍了两种类型的 GNN 输入信息，并讨论了它们的潜在性能优势和提取方法

3.1.1 GNN 模型信息

尽管图神经网络的更新阶段遵循相对固定的计算模式，其聚合阶段却呈现出高度多样性。主流的图神经网络聚合方法可分为两大类型：

第一类是基于邻居节点嵌入的聚合操作（如求和、取极小值等），典型代表如图卷积网络（GCN）[38]。采用此类聚合方法的图神经网络，通常会在更新阶段（即节点嵌入矩阵与权重矩阵相乘）[7-8,38] 先行降低节点嵌入维度，随后在各网络层执行聚合操作（收集邻居节点嵌入信息）。这种设计能显著减少聚合阶段的数据传输量，此时提升内存局部性将更为有利——更多节点嵌入可被缓存至高速存储器（如 GPU 的 L1 缓存），从而获得性能增益。

第二类则是需要结合特殊边特征（如权重、通过源节点与目标节点组合计算得到的边向量等）的聚合操作，如图同构网络（GIN）[58] 为代表。此类图神经网络必须基于完整高维节点嵌入来计算边特征。在此场景下，高速存储器（如 GPU 流式多处理器的共享内存）容量不足以实现有效的数据局部性优化。但考虑到计算负载可沿嵌入维度拆分并由更多并发线程共享，提升计算并行度（如基于嵌入维度的工作负载划分）反而能更有效地提高整体吞吐量。

我们通过 GCN 与 GIN 的数学表达式进一步阐释这种聚合类型差异。对于

GCN，其输出嵌入 $\mathbf{X}$ 的计算公式为：

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{W}, \tag{3-1}$$

其中 $\hat{\mathbf{D}}$ 为对角节点度矩阵，$\mathbf{W}$ 为权重矩阵，$\hat{\mathbf{A}}$ 为图邻接矩阵。而 GIN 每层的输出嵌入 $\mathbf{X}$ 计算公式为：

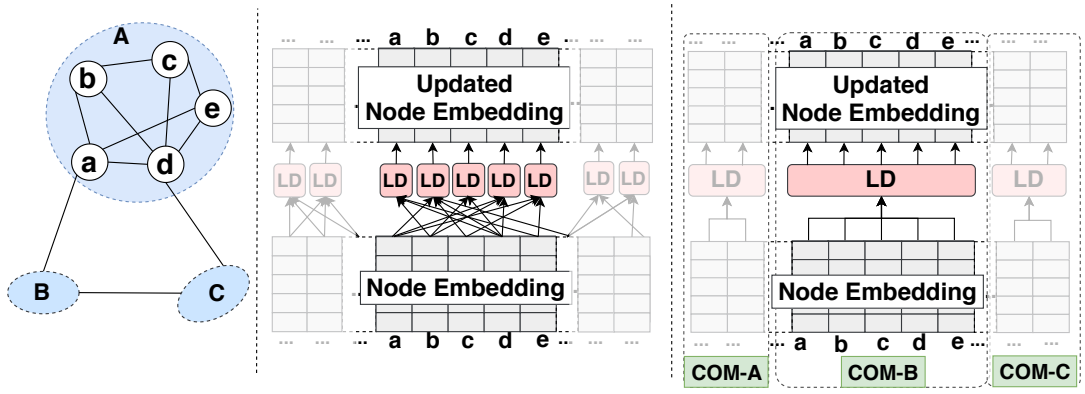
$$\mathbf{x}_i' = h\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right) \tag{3-2}$$

此处 h 表示神经网络（如多层感知机），负责将输入嵌入维度的节点特征 x 映射至输出嵌入维度；ϵ 是可根据用户需求或应用场景配置或训练的参数；$\mathcal{N}(i)$ 表示节点 i 的邻居集合。

假设 GCN 与 GIN 的隐藏维度均为 16，输入数据集的节点嵌入维度为 128。对于 GCN，我们会在节点更新阶段先对节点嵌入执行基于 GEMM（通用矩阵乘法）的线性变换，因此在聚合阶段仅需处理 16 维的节点数据。而 GIN 必须先在 128 维节点嵌入上执行邻居聚合，再通过线性变换将节点嵌入从 128 维降至 16 维。这种聚合差异导致优化策略的分化：GCN 更适合对小规模节点嵌入实施内存优化，而 GIN 更需针对大规模节点嵌入提升计算并行度。因此，GNN 的聚合类型应作为系统级优化的重要考量因素，这些信息可通过 GNNA 内置的模型属性解析器自动获取。

3.1.2 图信息

节点度和嵌入维度：现实世界中的图数据通常遵循节点度的幂律分布[5]。这种分布特性在传统图处理系统[6,52,59]中已经导致工作负载不均衡问题。而在图神经网络聚合阶段，如果采用以节点为中心的工作负载划分策略，节点嵌入的高维特性会进一步加剧这种不均衡现象。更为关键的是，节点嵌入的引入使得原本适用于图处理系统的缓存优化方案失效。由于缓存容量有限（如 GPU 线程块的 L1 缓存通常为 64KB），难以容纳足够数量的高维节点嵌入数据。例如，在图处理场景中，当每个节点仅含 4 字节浮点属性时，单个 GPU 线程块的 L1 缓存可以容纳 16×10^3 个节点，此时时空局部性优化能带来显著性能提升。然而，对于采用 64 维节点嵌入的主流图神经网络而言，每个线程块仅能缓存 256 个节点。当处理高度节点时，系统仍需从低速全局内存中频繁获取未命中的数据，这一



(a) Graph Community　　(b) Loading without Community　　(c) Loading with Community

图 3-1　图社区结构及其优化潜力（注：LD 表示加载操作，COM 表示社区）

瓶颈严重制约了聚合阶段的执行效率。

通过获取节点度（node degree）与嵌入维度信息，我们得以基于输入数据精确估算节点工作负载及其具体构成，从而为图神经网络开辟新的优化空间。当工作负载主要受邻居节点数量主导时（例如高度节点场景），可以定制化设计并行处理更多邻居的计算方案以提升邻居间计算并行度；反之，如果工作负载主要由节点嵌入维度决定（例如高维嵌入场景），则应优先考虑沿嵌入维度增强计算并行性。需要说明的是，节点度与嵌入维度信息可直接从加载的图结构和节点嵌入向量中提取， GNNA 系统正是基于此类信息实现 GNN 工作负载的智能化管理。。

图社区结构（graph community）[60-62] 是现实图数据的关键特征，表现为小规模节点群内部存在”强连接”（边密集），而与图其余部分仅保持”弱连接”（边稀疏）。图 3-1展示了利用图社区结构优化 GNN 的典型案例。现有图处理系统[49,51]采用的节点中心聚合方案如图 3-1所示，各节点需先加载其邻居再独立执行聚合。当邻居仅含轻量级标量属性时，该策略可获得优异计算并行性——此时并行加载的收益足以抵消共享邻居的重复加载开销。

然而在 GNN 计算中，节点嵌入的高维特性使得重复加载成本成为主导因素。以节点 a、b、c、d、e 的聚合为例，传统方案需加载 15 个节点嵌入（其中节点 d 被 a 和 b 重复加载），且嵌入维度越高，这种冗余加载越显著。若考虑图社区结构（图 3-1），仅需加载 5 个独立节点即可完成相同聚合，有效降低内存访问量。

这个想法虽然听起来很有潜力，但在 GPU 上的实现面临重大挑战。现有利用图社区的方法[62-63] 主要面向 CPU 平台，其线程并行度有限且单线程享有 MB 级缓存，优化重点在于提升单线程数据局部性。而 GPU 具备海量并行线程但单线程仅配 KB 级缓存，因此关键在于通过 L1 缓存实现线程间数据局部性优化。具体而言需完成两个转化：将输入级的节点邻接关系转化为 GPU 内核级的线程、线程束和线程块邻接关系。硬件层面的核心洞见在于：相邻 ID 的线程更容易共享内存和计算资源，从而提升数据时空局部性。 GNNA 通过社区感知的节点重编号和 GNN 专用内存优化（第3.3节）实现上述机制。

3.2 二维工作负载管理

图神经网络（GNN）在图计算领域占据独特地位，其核心特征在于采用高维特征向量（即嵌入表示）来表征每个节点。GNN 的计算负载主要沿两个维度增长：邻居节点数量与嵌入维度规模。为此， GNNA 系统创新性地引入了面向 GNN 的输入驱动型参数化二维工作负载管理机制，该机制包含三大关键技术：粗粒度邻居分区，细粒度维度分区，线程束级线程对齐。

3.2.1 粗粒度邻居分区

粗粒度邻居划分是一种专为基于 GPU 的 GNN 计算设计的新颖工作负载均衡技术。它旨在解决节点间工作负载不均衡和冗余原子操作的挑战。

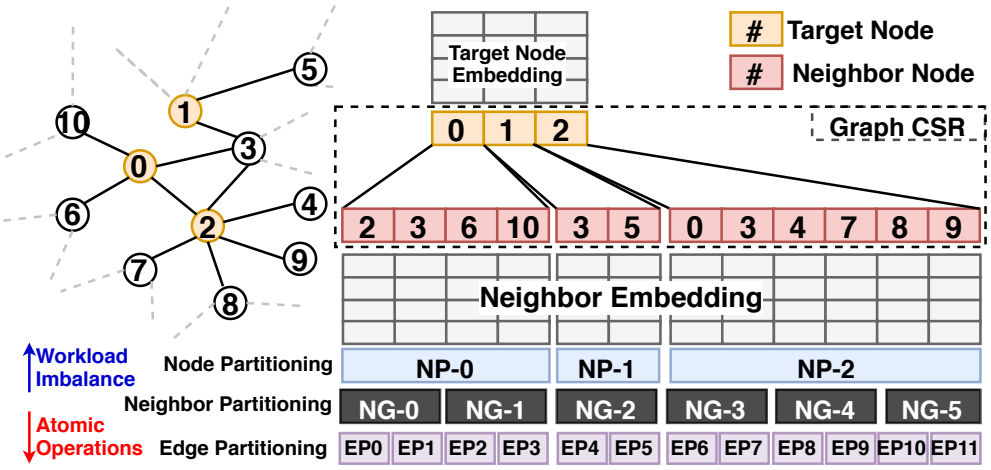


图 3-2 邻居划分。注意：“NP”：节点划分；“EP”：边划分；“NG”：邻居组。

具体来说，基于加载的图压缩稀疏行 (CSR) 表示，我们的粗粒度邻居划分会首先将一个节点的邻居分解为一系列大小相等的邻居组，并将每个邻居组 (NG) 的聚合工作负载视为调度的基本工作负载单元。图 3-2 例示了一个无向图及其对应的邻居划分结果。节点 0 的邻居被划分为两个邻居组（NG-0 和 NG-1），预定组大小为 2。节点 1 的邻居（节点 3 和节点 5）由 NG-2 覆盖，而节点 2 的邻居则分布在 NG-3,4,5 中。为了支持邻居组，我们引入了两个组件：邻居划分模块和邻居划分图存储。前者是构建在图加载器之上的一个轻量级模块，它通过将图的 CSR 划分为大小相等的组来实现。注意，为了便于调度和同步，每个邻居组仅覆盖一个目标节点的邻居。邻居划分图存储维护每个邻居组基于元组的元数据，包括其 ID、其邻居节点在 CSR 表示中的起始和结束位置，以及源节点。例如，NG-2 的元数据将被存储为 (2, 1, (4, 6))，其中 2 是邻居组 ID，1 是目标节点 ID，(4, 6) 是 CSR 中邻居节点的索引范围。

应用基于邻居划分的聚合有以下三个好处：1) 与基于节点/顶点中心划分的更粗粒度聚合[49] 相比，邻居划分可以在很大程度上缓解工作负载单元的大小不规则性，从而提高 GPU 占用率和吞吐量性能；2) 与更细粒度的基于边中心的划分（被现有 GNN 框架如 PyG[7] 用于批处理和张量化，以及被图处理系统[51,53] 用于大规模计算并行化）相比，邻居划分方案可以避免管理许多微小工作负载单元的开销，这些开销可能从多方面损害性能，例如调度开销和过多的同步操作；3) 它引入了一个与性能相关的参数，即**邻居组大小**（ngs），该参数用于设计参数化和性能调优。邻居划分以单个邻居节点的粗粒度进行工作。对于低维设置，它可以很大程度上缓解工作负载不均衡问题。对于高维节点嵌入，我们采用下一小节将讨论的一种细粒度维度划分方法，将每个邻居组的工作负载进一步分发到线程。注意，当邻居数量不能被邻居组大小整除时，会导致邻居组不均衡。通过将邻居组大小设置为一个较小的值（例如 3），可以分摊这种不规则性。

3.2.2 细粒度邻居分区

GNN 与传统图算法的区别在于其对节点嵌入的计算。为了探索沿着这一维度加速的潜在并行性，我们利用细粒度维度划分来沿着嵌入维度进一步分配邻居组的工作负载，以提高聚合性能。如图 3-3所示，原始的邻居组工作负载被均

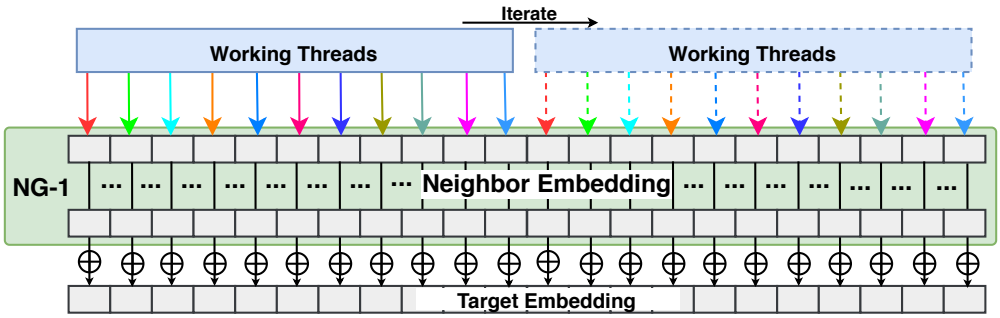


图 3-3 维度划分。⊕：累加。

匀地分配给 11 个连续的线程，其中每个线程独立地管理沿一个维度的聚合（即，所有邻居节点嵌入向目标节点嵌入的累加）。如果维度大小大于工作线程的数量，则需要更多的迭代来完成聚合。

使用维度划分有两个主要原因。首先，它可以适应更多样化的嵌入维度大小范围。我们可以通过增加并发维度工作线程的数量或启用更多迭代来灵活地处理维度变化。这对于具有日益复杂的模型结构和不同嵌入维度大小的现代 GNN 至关重要。其次，它引入了另一个与性能相关的参数——用于设计定制的工作线程数量（**维度工作线程** (dw)）。该参数的值有助于平衡线程级并行性和单线程效率（即每个线程的计算工作负载）。

3.2.3 线程束级线程对齐

虽然上述两种技术回答了我们如何逻辑地平衡 GNN 工作负载，但是如何将这些工作负载映射到底层 GPU 硬件以实现高效执行的问题仍未解决。一个直接的解决方案是分配连续的线程来并发处理来自不同邻居组的工作负载（图 3-4）。然而，这些线程之间不同的行为（例如，数据操作和内存访问操作）会导致线程分化和 GPU 利用率不足。来自同一个 Warp 的线程以单指令多线程（SIMT）的方式执行，而 Warp 调度器每个周期只能处理一种类型的指令。因此，不同的线程必须等待轮到它们执行，直到流式多处理器（SM）的 Warp 调度器发出它们相应的指令。

为了应对这一挑战，我们引入了一种与我们的邻居和维度划分相协调的 Warp 对齐线程映射方法，以系统地利用均衡工作负载带来的性能优势。如图 3-4所示，每个 Warp 将独立管理来自一个邻居组的聚合工作负载。因此，不同邻

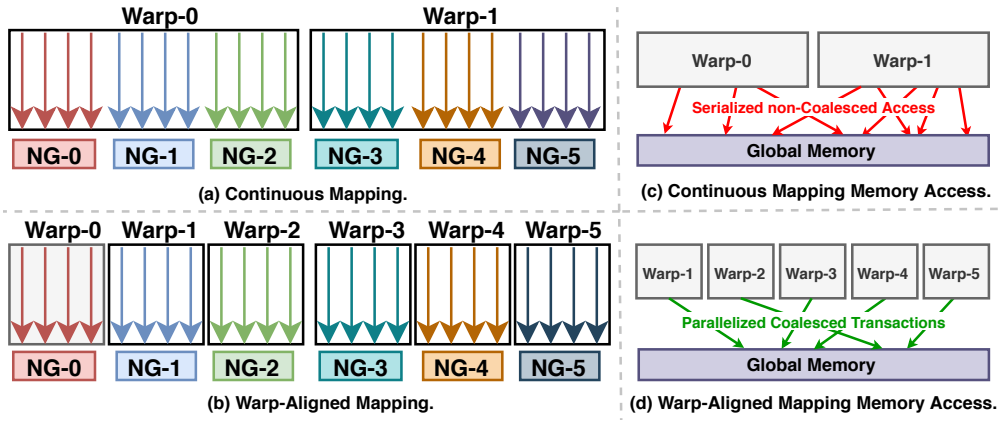


图 3-4　基于 Warp 的线程对齐。

居组（例如，NG-0 到 NG-5）的执行可以很好地并行化，而不会引起 Warp 分化。采用基于 Warp 的线程对齐有以下几个好处。首先，线程间的同步（例如，原子操作）可以被最小化。同一个 Warp 的线程工作在同一个邻居组的不同维度上，因此，对于来自同一个 Warp 的线程，无论是访问全局内存还是共享内存都不会发生冲突。其次，单个 Warp 的工作负载减少了，不同的 Warp 将处理更均衡的工作负载。因此，SM Warp 调度器可以更灵活地管理更多的 Warp，以提高整体并行性。考虑到聚合过程中每个 Warp 不可避免的全局内存访问，增加 Warp 的数量可以提高 SM 占用率以隐藏延迟。第三，内存访问可以被合并。来自同一个 Warp 且具有连续 ID 的线程将访问全局内存中用于节点嵌入的连续内存地址。因此，与连续线程映射（图 3-4）相比，Warp 对齐的线程映射可以将来自同一个 Warp 的内存请求合并为一个全局内存事务（图 3-4）。

3.3　专用内存优化

为了进一步利用二维工作负载的优势，我们引入了针对 GNN 的专用内存优化方法：社区感知的节点重编号和 *Warp* 感知的内存定制化。

3.3.1　面向社区感知的节点重编号

为了探索图社区（第 3.1.2节）带来的性能优势，我们采用了轻量级的节点重编号方法，通过重新排序节点 ID 来改善 GNN 聚合过程中的时间/空间局部性，且不影响输出的正确性。其关键思想是，节点 ID 的邻近性将映射到 GPU 上处

理它们的计算单元的邻近性。在 GNNA 中，我们的二维工作负载管理根据节点 ID 将一个节点的邻居组分配给连续的 Warp。如果两个节点被分配了连续的 ID，那么它们对应的邻居组（Warp）在其 Warp ID 上也会彼此接近。因此，它们更有可能被紧密地调度到具有共享 L1 缓存的同一个 GPU SM 上，从而提高加载的共同邻居的数据局部性。为了有效地应用节点重编号，必须解决两个关键问题。

何时应用：虽然图重排序为性能提供了潜在的好处，但我们仍需弄清楚哪种类型的图能从此类重排序优化中受益。我们的关键见解是，对于邻接矩阵已经呈现近似块对角模式形状的图（图 3-5），重排序无法带来更多的局部性好处，因为每个社区内的节点在其节点 ID 方面已经彼此接近。对于形状更不规则的图（图 3-5），其边的连接以不规则模式分布在节点之间，重排序可以带来显著的性能提升（高达 2 倍加速，将在第 4.4节中讨论）。为此，我们提出了一个新的度量指标——*平均边跨度*（AES），用于判断进行图重排序是否有利。

$$\mathbf{AES} = \frac{1}{\#E} \sum_{(src_{id}, trg_{id}) \in E} |src_{id} - trg_{id}| \tag{3-3}$$

其中 E 是图的边集；$\#E$ 是总边数；src_{id} 和 trg_{id} 分别是每条边的源节点和目标节点 ID。计算 AES 是轻量级的，并且可以在初始图加载期间动态完成。我们对大量图进行的性能剖析也表明，当 $\sqrt{AES} > \lfloor \frac{\sqrt{\#N}}{100} \rfloor$ 时，节点重编号更有可能提高运行时性能。

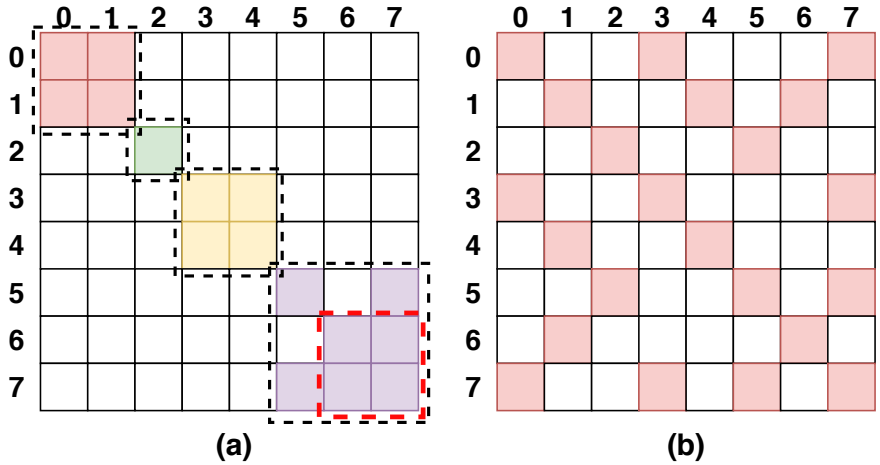


图 3-5　图边连接模式。注意，每个彩色方块代表两个节点之间的边。(a) 中的不同颜色代表来自不同社区的边。红色虚线框表示子社区。

如何应用：我们利用 Rabbit Reordering[64] 来进行重排序，这是一种完全并行

化且低成本的图重排序技术。具体来说，它首先通过分层合并边和聚类节点来最大化图的模块度。然后，它通过 DFS 遍历在每个簇内生成节点顺序。Rabbit Reordering 已经被证明在捕获的图社区质量（数据局部性）、并行化的便捷性以及性能方面优于其他图聚类方法[23,65-68]，包括基于社区的方法（如 METIS[23]）和基于 BFS 的方法（如 Reverse Cuthill-McKee (RCM)[68]）。更重要的是，Rabbit Reordering 可以分层地捕获图社区（即，一组较小的子社区包含在一个较大的社区中，如图 3-5所示例）。这种不同粒度级别的社区能够很好地匹配 GPU 的缓存层次结构，其中较小的子社区（占用一个 SM）可以从 L1 缓存中获得数据局部性的好处，而较大的社区（占用多个 SM）可以从更大的 L2 缓存中获得数据局部性的好处。我们在第 **??**节中定量讨论了这种局部性带来的好处。

3.3.2 Warp 感知的内存定制化

现有的工作[7,51] 利用大量的全局内存访问来读取和写入嵌入，并使用大量的原子操作进行聚合（一种规约操作）。然而，这种方法会产生巨大的开销，并且未能利用共享内存的潜在优势。例如，当将一个具有 k 个邻居组（每个组有 ngs 个邻居，嵌入维度为 Dim）的目标节点的邻居聚合到一个 Dim 维嵌入时，这涉及到 $O(k \cdot ngs \cdot Dim)$ 次原子操作和 $O(k \cdot ngs \cdot Dim)$ 次全局内存访问。

相比之下，我们提出了一种以 Warp 为中心的共享内存优化技术。我们的关键见解是，通过根据块级 Warp 组织模式（图 3-5）定制共享内存布局，我们可以显著减少原子操作和全局内存访问的数量。首先，我们为每个邻居组（Warp）的目标节点保留一个共享内存空间（对于浮点嵌入为 $4 \times Dim$ 字节），这样来自一个 Warp 的线程可以将规约的中间结果缓存在共享内存中。随后，在一个线程块内，考虑到每个节点的邻居可能分布在不同的 Warp 中，我们仅指定一个 Warp（称为*领导者*）负责将每个目标节点的中间结果从共享内存复制回全局内存。详细的定制化过程在算法 3-1中描述。具体来说，每个 Warp（保存在 $warpPtr$ 中）具有三个属性：$nodeSharedAddr$（用于邻居组聚合结果的共享内存地址），$nodeID$（目标节点的 ID），以及 $leader$（一个布尔标志，指示当前 Warp 是否为将结果从共享内存写回到全局内存的领导者 Warp）。主要的定制化例程（第 4 行到第 22 行）根据 Warp 相对于线程块的索引位置来处理不同的 Warp。注意，这种共享内

存定制化成本低廉，并且仅在 GPU 核函数执行前，与常规的图初始化过程一起动态完成一次。

在我们的设计中，当处理一个具有 k 个邻居组（每个组有 ngs 个邻居，嵌入维度为 Dim）的目标节点时，仅涉及 $O(Dim)$ 次原子操作和 $O(Dim)$ 次全局内存访问。由此，我们可以将原子操作和全局内存访问减少 $(k \cdot ngs)$ 倍，从而显著加速聚合操作。在这里，我们将 ngs 视为一个超参数，用以平衡内存访问效率和计算并行性，我们将在第 3.4节进一步讨论其值的选择。

3.4 设计优化

我们 GPU 核函数配置中的参数可以进行调整，以适应各种 GNN 模型和图数据集。但是，如何自动选择能够带来最优性能的参数仍然是一个未知的问题。在本节中，我们介绍 GNNA 中 **Decider**（决策器）所使用的分析模型和自动参数选择方法。

分析建模： GNNA 的性能/资源分析模型有两个变量：每个线程的工作负载 (*WPT*) 和每个块的共享内存使用量 (*SMEM*)。

$$\textbf{WPT} = ngs \times \frac{Dim}{dw}, \quad \textbf{SMEM} = \frac{tpb}{tpw} \times Dim \times FloatS \tag{3-4}$$

其中 *ngs* 和 *dw* 分别是邻居组大小和维度工作线程数量（第 3.2.2节）；*Dim* 是节点嵌入维度；*IntS* 和 *FloatS* 在 GPU 上均为 4 字节；*tpb* 是每块线程数，*tpw* 是每 Warp 线程数；对于 GPU，*tpw* 为 32，而 *tpb* 由用户选择。

参数自动选择：为了确定 *ngs* 和 *dw* 的值，我们遵循两个步骤。首先，我们根据 *tpw*（硬件约束）和 *Dim*（输入属性）来确定 *dw* 的值，如公式 3-5所示。注意，我们是通过对不同数据集和 GNN 模型进行性能剖析来得出这个公式的。

$$dw = \begin{cases} tpw & Dim \geq tpw \\ \frac{tpw}{2} & Dim < tpw \end{cases} \tag{3-5}$$

其次，我们根据选定的 *dw* 和用户指定的 *tpb* 来确定 *ngs* 的值。约束条件包括使 $WPT \approx 1024$ 且 $SMEM \leq SMEMperBlock$。注意，$SMEMperBlock$ 在现代 GPU 上为 48KB 到 100KB[69]。在不同的 GPU 上，即使 CUDA 核心数量和全局内存带宽不同，单线程的工作负载能力（由 *WPT* 衡量）仍然相似。*tpb* 通常选择为 2 的

算法 3-1 Warp 感知的内存定制化

Input: 邻居组大小 *ngs*，线程块线程数 *threadPerBlock*，每 Warp 线程数 *threadPerWarp*

Output: Warp 指针数组 *warpPtr*

1: $warpNum \leftarrow$ **computeGroups**(ngs) ▷ 计算邻居组数量（即 *Warp* 数量）
2: $warpPerBlock \leftarrow$ **floor**$(threadPerBlock/threadPerWarp)$ ▷ 计算每个线程块的 *Warp* 数量
3: 初始化追踪变量：$cnt \leftarrow 0$, $local\_cnt \leftarrow 0$, $last \leftarrow 0$
4: **while** $cnt < warpNum$ **do**
5: **if** $cnt \bmod warpPerBlock == 0$ **then** ▷ 位于线程块起始位置的 *Warp*
6: $warpPtr[cnt].nodeSharedAddr \leftarrow local\_cnt \times Dim$
7: $last \leftarrow warpPtr[cnt].nodeID$
8: $warpPtr[cnt].leader \leftarrow$ **true**
9: **else**
10: **if** $warpPtr[cnt].nodeID == last$ **then** ▷ *Warp* 的目标节点与前一个 *Warp* 相同
11: $warpPtr[cnt].nodeSharedAddr \leftarrow local\_cnt$
12: **else** ▷ *Warp* 的目标节点与前一个 *Warp* 不同
13: $local\_cnt \leftarrow local\_cnt + 1$
14: $warpPtr[cnt].nodeSharedAddr \leftarrow local\_cnt$
15: $last \leftarrow warpPtr[cnt].nodeID$
16: $warpPtr[cnt].leader \leftarrow$ **true**
17: **end if**
18: **end if**
19: **if** $(++cnt) \bmod warpPerBlock == 0$ **then** ▷ 下一个 *Warp* 属于新的线程块
20: $local\_cnt \leftarrow 0$
21: **end if**
22: **end while**

幂且小于或等于 1024。我们基于微基准测试和先前文献[70] 的分析表明，较小的块（1 到 4 个 Warp，即 $32 \leq tpb \leq 128$）可以提高 SM Warp 调度的灵活性并避免尾部效应，从而带来更高的 GPU 占用率和吞吐量。我们将在第 4.4节进一步证明我们分析模型的有效性。

3.5 本章小结

本章详细阐述了面向 GPU 的高性能图神经网络（GNN）加速算法框架 GNNA 的设计理念与关键技术实现。针对 GNN 计算中存在的计算模式多样性、工作负载不均衡以及高维嵌入带来的内存访问瓶颈等核心挑战，本章从 GNN 输入特性分析出发，提出了一系列优化策略。首先，本章分析了 GNN 模型信息（聚合类型）和图信息（节点度、嵌入维度、社区结构）对优化策略选择的指导意义，强调了基于输入信息进行自适应优化的重要性。其次，为应对 GNN 计算负载在邻居数量和嵌入维度上的双重增长特性，GNNA 创新性地引入了二维工作负载管理机制。该机制包含：1) 粗粒度邻居分区，通过将邻居划分为固定大小的邻居组（NGs），有效缓解了节点间工作负载不均衡，并引入了可调参数 ngs；2) 细粒度维度分区，将邻居组的计算任务沿嵌入维度分发给多个线程（dw），提升了计算并行度以适应不同嵌入维度；3) 线程束级线程对齐，将每个邻居组的聚合任务映射到一个独立的 Warp，最大限度地减少了 Warp 内分化，降低了同步开销，并改善了内存访问合并效率。再次，为了进一步挖掘性能潜力并利用图数据的内在结构，本章提出了专用内存优化技术：1) 社区感知的节点重编号，利用轻量级图重排序算法（如 Rabbit Reordering）和平均边跨度（AES）度量，在预处理阶段优化节点 ID 布局，增强了计算过程中的数据时空局部性，尤其是在具有明显社区结构或边分布不规则的图上效果显著；2) Warp 感知的内存定制化（详见算法 3-1），通过为每个 Warp 分配目标节点的共享内存缓冲，并指定 Leader Warp 负责最终结果写回，显著减少了聚合过程中的原子操作和全局内存访问次数，大幅降低了内存开销。最后，为了实现算法参数的自动化配置，本章介绍了 GNNA 内置决策器（Decider）所采用的分析模型与参数自动选择方法。通过建立工作负载（WPT）和共享内存（SMEM）的量化模型，并结合硬件约束与输入特性，

该方法能够自动推导出较优的邻居组大小（ngs）和维度工作线程数（dw），从而简化了用户配置，并确保了算法在不同 GNN 模型和数据集上的高效执行。综上所述，本章提出的 GNNA 设计，通过对 GNN 输入信息的深刻洞察、创新的二维工作负载管理、以及针对性的专用内存优化和自动化参数调优，构建了一套系统性的 GNN 加速方案，旨在显著提升 GNN 在 GPU 平台上的训练与推理性能。

4 实验与结果分析

在本章节中，我们评估了 GNNA 在各种 GNN 模型和图数据集上的性能，对 GNNA 的加速效果进行了全面的分析，并与现有的 GNN 框架（如 DGL 和 PyG）进行了比较。我们还进行了一些附加实验研究，对 GNN 的隐藏维度，GNNA 的参数选择等问题进行了探讨。

4.1 实验设置

基准测试 (Benchmarks): 我们选择了先前工作[7-8,71] 在节点分类任务中广泛使用的两个最具代表性的 GNN 模型，以覆盖不同类型的聚合操作。1) 图卷积网络 (GCN)[38] 是最流行的 GNN 模型架构之一。它也是许多其他 GNN（如图采样网络 GraphSAGE[39] 和可微分池化 Diffpool[72]）的关键骨干网络。因此，提升 GCN 的性能也将惠及广泛的 GNN 模型。对于 GCN 的评估，我们采用以下设置：*2 个层，隐藏维度为 16*，这也是原始论文[38] 中的设置。2) 图同构网络 (GIN)[58]。GIN 与 GCN 的不同之处在于其聚合函数，该函数对节点自身的嵌入值进行加权。此外，GIN 也是许多其他具有更多边属性的更高级 GNN（如图注意力网络 GAT[40]）的参考架构。对于 GIN 的评估，我们采用以下设置：*5 个层，隐藏维度为 64*，这是原始论文[58] 中使用的设置。

基线 (Baselines): 我们选择了两种同样基于 Pytorch 的图神经网络框架作为基线进行比较。1) Deep Graph Library (DGL)[8] DGL 是一个专注于 GNN 计算效率的框架，特别是在 GPU 加速方面。它提供了针对多种主流 GNN 架构的高度优化的内核实现，并在许多标准图数据集和任务上展现出强大的性能。DGL 通过其底层的张量操作优化和高效的图表示，致力于为用户提供易于使用且性能卓越的 GNN 开发体验。2) Pytorch-Geometric (PyG)[7] PyG 是另一个极其流行且功能强大的 GNN 框架，同样构建于 PyTorch 之上。PyG 以其灵活性和易于扩展性而闻名，尤其是在定义自定义 GNN 层和消息传递机制方面。它提供了一套简洁的 API，方便研究人员快速实现和迭代新颖的 GNN 模型。PyG 同样在性能方面表现出色，并在各种研究和应用中被广泛采用。

数据集 (Datasets): 我们涵盖了先前 GNN 相关工作[7-8,71] 中使用的所有三种类型的数据集。I 型图 是先前 GNN 算法论文[38-39,58] 常用的典型数据集。它们的节点和边数量通常较小，但节点嵌入信息丰富且维度较高。II 型图[73] 是图核 (graph kernels) 的流行基准数据集，并被选为 PyG[7] 的内置数据集。每个数据集包含一组小图，这些小图仅有图内边连接，没有图间边连接。III 型图 [38,74] 在节点和边的数量都很大。这些图表现出高度的结构不规则性，这对大多数现有的 GNN 框架来说都具有挑战性。这些数据集的详细信息列于表 4-1。其中，**# 类型**于节点分类任务。

表 4-1 用于评估的数据集

类型	数据集	# 顶点	# 边	维度	# 类别
I	Citeseer	3,327	9,464	3,703	6
	Cora	2,708	10,858	1,433	7
	Pubmed	19,717	88,676	500	3
	PPI	56,944	818,716	50	121
II	PROTEINS_full	43,471	162,088	29	2
	OVCAR-8H	1,890,931	3,946,402	66	2
	Yeast	1,714,644	3,636,546	74	2
	DD	334,925	1,686,092	89	2
	TWITTER-Partial	580,768	1,435,116	1,323	2
	SW-620H	1,889,971	3,944,206	66	2
III	amazon0505	410,236	4,878,875	96	22
	artist	50,515	1,638,396	100	12
	com-amazon	334,863	1,851,744	96	22
	soc-BlogCatalog	88,784	2,093,195	128	39
	amazon0601	403,394	3,387,388	96	22

平台与指标 (Platforms & Metrics): 我们使用 CUDA 12.4 和 PyTorch 2.4.0 作为编程平台, 基于 C++ 和 CUDA 实现了 GNNA 的后端，并通过 Python 绑定 PyTorch 前端。评估平台采用了一台配备 16 核心、32 线程的 AMD Ryzen 9 7945HX 处理器和 NVIDIA RTX 4060 GPU 的笔记本电脑，运行环境为 WSL2 下的 Ubuntu 24.04。在性能评估过程中，为了量化加速效果，我们通过计算 200 次端到端推理（前向传播）和训练（前向传播和反向传播）的平均延迟（不包括 i 数据预处理和加载时间）来衡量性能提升。

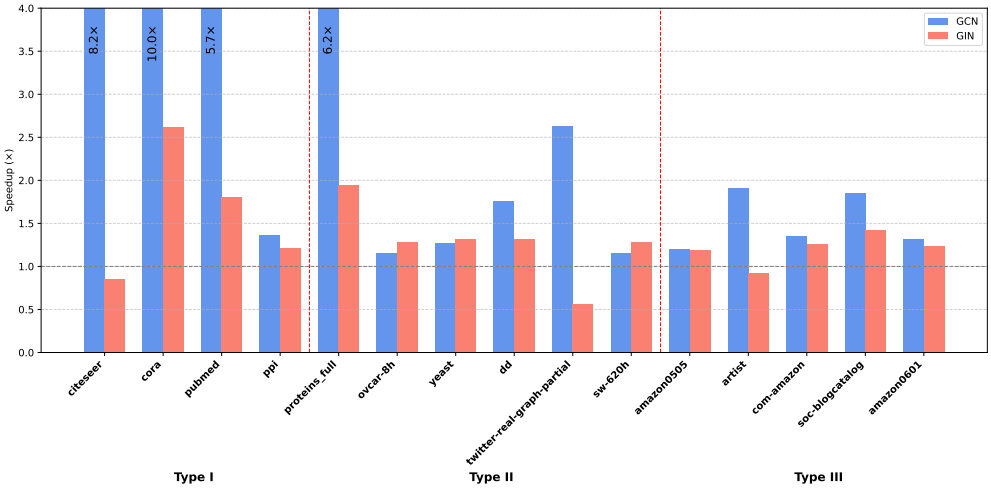


图 4-1　GNNA 在 GCN 和 GIN 上相较于 DGL 的推理加速比 (×)。

4.2　性能评估

在本节中，我们首先对比分析了 GNNA 与 DGL 在图神经网络（GNN）推理任务中的性能差异，随后将对比扩展至训练任务。如图 4-1 所示，GNNA 在 GCN 和 GIN 两类模型的推理任务中，分别在三类数据集上相较于 DGL[8] 平均加速达到了 3.13× 和、1.35×。尤其是在挑战性较高的第三类图上，GNNA 对于 GCN 和 GIN 的平均加速分别为 1.53× 和 1.12×。这得益于 GNNA 充分挖掘输入图的结构信息（如节点度数）用于指导系统优化。具体而言，我们基于节点分组策略进行工作负载划分，从而提升了 GPU 利用率。我们接下来将针对三类图数据展开具体分析，并探讨其中的性能优化机理。

I 型图: 对于该类图，GCN 相比 DGL 的加速效果明显优于 GIN，平均分别为 6.30× 与 1.35×。两者差异主要源自其各自的计算流程：GCN 在聚合操作之前进行节点特征降维（即 DGEMM 操作），显著减少了聚合阶段的数据移动和线程同步开销。这种结构非常契合 GNNA 所提出的二维任务划分机制和面向局部性的内存优化策略，能充分发挥 GPU 的数据局部性优势。而 GIN 的计算顺序则相反，必须先执行聚合操作再进行节点特征变换，因此无法避免高频次的全局内存访问和数据传输，限制了数据局部性和共享内存的优势发挥。尽管如此，我们提出的细粒度维度划分策略依然能够较好地处理高维特征场景，保障整体性能。

II 型图: 在这类图上，GNNA 依然取得了较为显著的性能提升，在 GCN 和

GIN 上平均加速比分别为 2.37× 与 1.27×，但在 *TWITTER-Partial* 数据集上差距略大，尤其是在 GIN 上，效果逊色于 DGL。该数据集的节点特征维度高达 1323，是此类图中最高的。而 GIN 必须先在 1323 维节点嵌入上执行邻居聚合，再进行线性变换降维。这导致了大量的全局内存访问和线程间同步开销，限制了性能提升。相比之下，GCN 先进行线性变换降维，再执行邻居聚合，因此在该数据集上仍然能够获得较好的加速效果。

III 型图: 这些图在节点和边的数量上都很大，表现出高度的结构不规则性，极具挑战性，但是 GNNA 依然取得了一定效果。例如 *soc-BlogCatalog*， GNNA 对 GCN 和 GIN 的平均加速比分别达到 1.85× 与 1.43×。这类图通常存在较高的线程间同步和全局内存访问开销，而我们提出的二维任务划分与内存优化机制可有效缓解这一瓶颈。此外，我们基于社区结构的信息进行的节点重新编号，进一步提升了线程之间在局部节点处理中的数据共享效率，从而提高了空间与时间局部性。但在 *artist* 数据集上，GIN 的加速效果相对较低。分析发现，该数据集社区划分的标准差最高，社区结构分布极不均衡，导致我们难以充分利用社区信息在 GNN 聚合阶段捕捉局部性，也限制了线程映射对齐、共享内存调度等系统优化策略的实施效果。

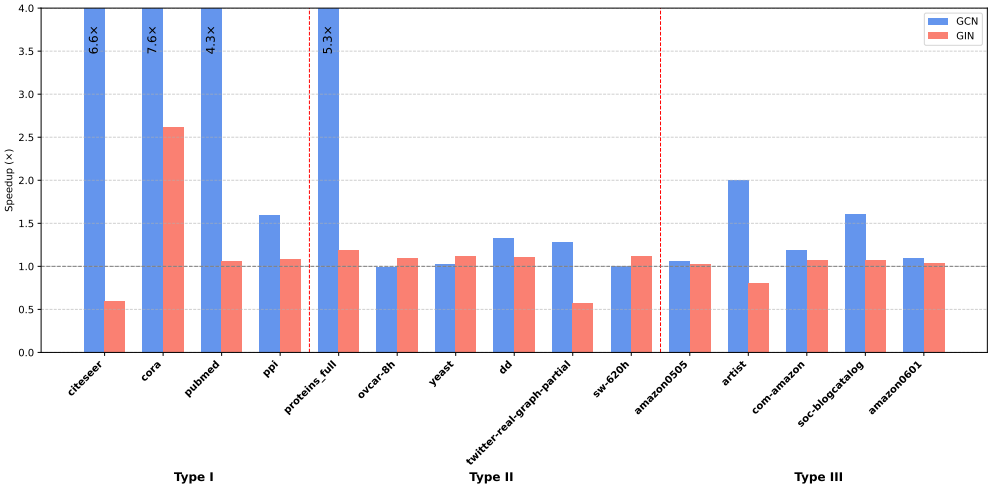


图 4-2　GNNA 在 GCN 和 GIN 上相较于 DGL 的训练加速比 (×)。

训练性能评估：我们进一步评估了 GNNA 在 GCN 和 GIN 模型上的训练性能，并与 DGL 进行了对比。相比推理，训练过程更加复杂，涉及前向传播与反向传播两个阶段，二者均严重依赖底层图聚合计算的性能。如图 4-2所示， GNNA

在 GCN 和 GIN 上的平均训练加速比分别为 1.61× 和 2.00×，验证了我们基于输入特性优化策略的普适性与有效性。训练与推理的主要差异体现在两个方面：首先，训练过程中需要执行反向传播，这一阶段与推理中的前向计算类似，因此仍能从我们提出的优化中获益；其次，训练中需额外存储前向过程中的中间结果，直到反向传播使用，这会引入额外的数据访问与内存占用开销。因此，训练阶段的整体加速比相对推理阶段略低，这是由其本质的资源需求所决定的。

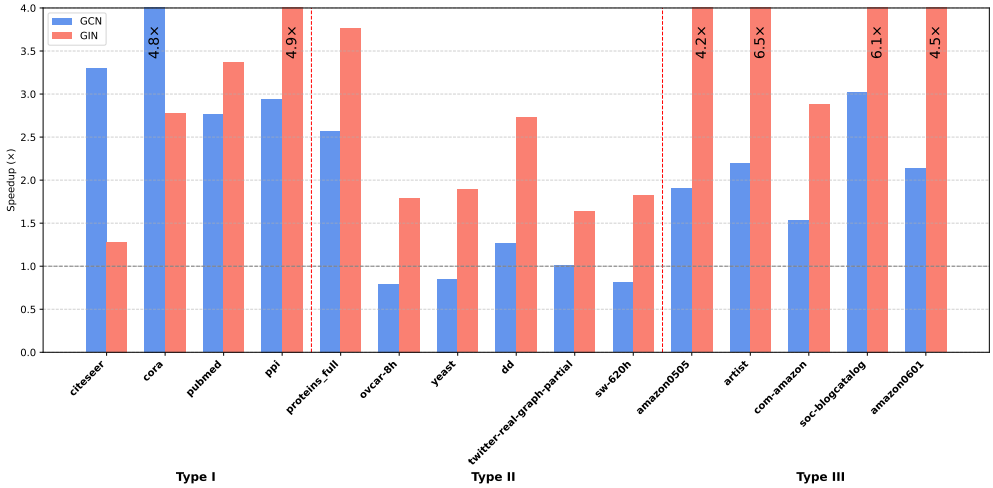


图 4-3　GNNA 在 GCN 和 GIN 上相较于 PyG 的推理加速比 (×)。

4.2.1　和 PyG 的比较

相比于 PyG，GNNA 展现出显著的性能优势。在 GCN 和 GIN 模型上，GNNA 的训练速度分别提升了 2.13× 和 3.34×，推理速度则提升了 3.13× 和 1.35×。其中，在 I 型图与 III 型图数据集上表现尤为突出，具体原因已在与 DGL 的对比分析中详细探讨，此处不再赘述。以下重点分析 GNNA 在 II 型图上的表现差异。

在 II 型图数据集上，GNNA 的性能优势不明显，甚至在某些情况下表现不佳。以 GCN 为例，除 *Proteins_full* 数据集外，其余数据集的平均训练加速比 0.92×，平均推理加速比 0.94×，未能超过 PyG。这是因为 PyG 针对此类具有分块对角化邻接矩阵特性的批处理图数据优化最为全面（例如小批量处理机制），能够高效处理此类图结构数据。相比之下，GNNA 仍然沿用了以节点度为基础的任务划分策略，在图规模较小、结构较为稠密的 II 型图中未能充分利用硬件资源，导致性能受限。

值得注意的是，在 GIN 模型上，GNNA 即便在 II 型图上仍取得了良好效果，实现了 1.95× 的训练加速和 2.67× 的推理加速。这主要归因于 GIN 所引入的线性变换操作在特征聚合阶段引入了更重的计算负载， GNNA 的维度工作线程划分策略和线程束级线程对齐策略能够更好地适应这一计算模式，充分利用 GPU 的计算资源和内存带宽，从而实现了更高的性能提升。

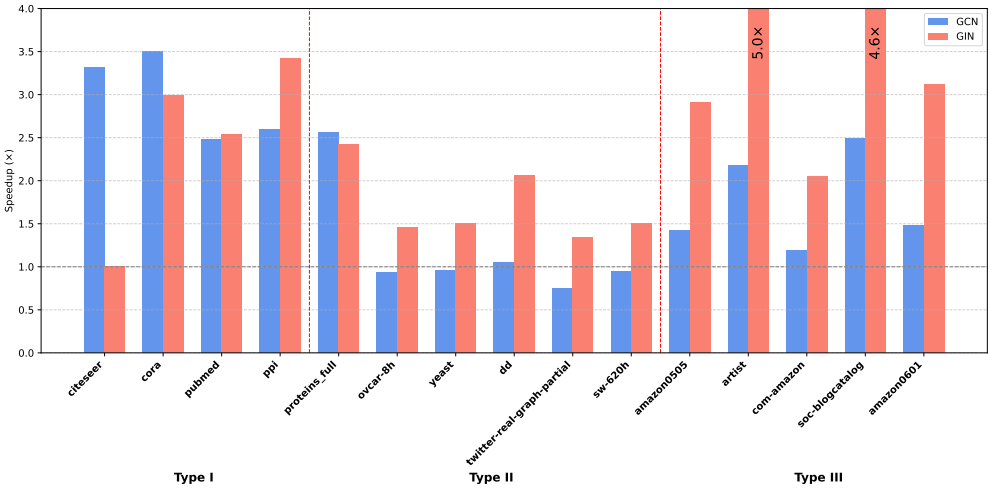
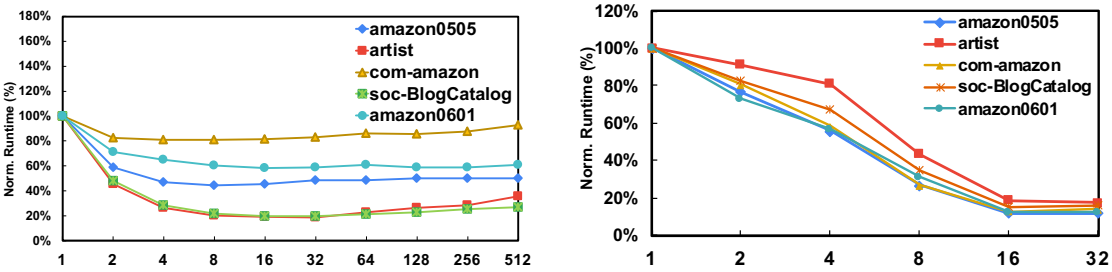


图 4-4　GNNA 在 GCN 和 GIN 上相较于 PyG 的训练加速比 (×)。

4.3　优化分析

在这一节中，我们详细分析了第三章中设计的优化策略。



(a) 邻居分区大小对运行时间的影响　　(b) 维度工作线程数对运行时间的影响

图 4-5　隐藏维度与延迟的关系

4.3.1　邻居分区

从图 4-5a可以看出，邻居分区策略在优化线程并行效率方面起到了关键作用。随着邻居组大小的增加， GNNA 的运行时间最初呈下降趋势。这是因为更

大的邻居组可以让每个线程处理更多的邻接节点，从而提高线程的计算饱和度，减少线程间的竞争与同步（如原子操作），同时提升缓存命中率，改善数据访问局部性。然而，当邻居组大小超过某个临界值（例如在 artist 数据集中为 32）后，线程的并行计算能力已达到上限，继续扩大邻居组会导致工作负载不均衡、部分线程空闲或等待，从而引发新的瓶颈，反而会增加总体运行延迟。

4.3.2 维度分区

如图 4-5b 所示，维度分区策略通过将节点嵌入特征向量的维度划分给不同线程来实现并行计算。当维度工作线程数量从 1 增加到 16 时，可以显著提升并行度和吞吐量，因为原本顺序执行的维度计算被拆分并发执行，显著缩短了关键路径。但进一步增加线程数量至 32 时，性能提升变得不明显。这是因为此时线程之间的并行性已基本饱和，系统资源（如寄存器、共享内存）竞争加剧，导致额外线程无法有效提升整体性能，还会引入额外的引入调度和同步开销。

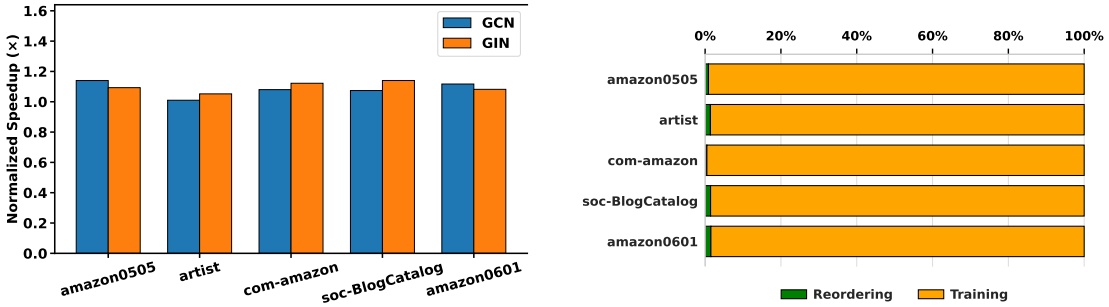
4.3.3 社区感知节点重编号

我们通过对 GCN 和 GIN 在 III 型数据集上的分析，展示了节点重新编号的好处。如图 4-6a所示，重新编号节点可以使 GCN 和 GIN 分别加速 1.14 倍和 1.09 倍。主要原因是我们的社区感知节点重新编号可以在 GNN 聚合过程中提高数据的空间和时间局部性。

为了量化这种局部性优势，我们提取了详细的 GPU 内核指标——DRAM 内存读写字节数。CUDA 内核指标分析结果表明，节点重编号可以有效提升（GCN 平均减少 12.4%，GIN 平均减少 9.3%），因为重排序后提升了图的局部性，具有连续 ID 的节点更有可能共享已加载的节点嵌入。

4.4 额外研究

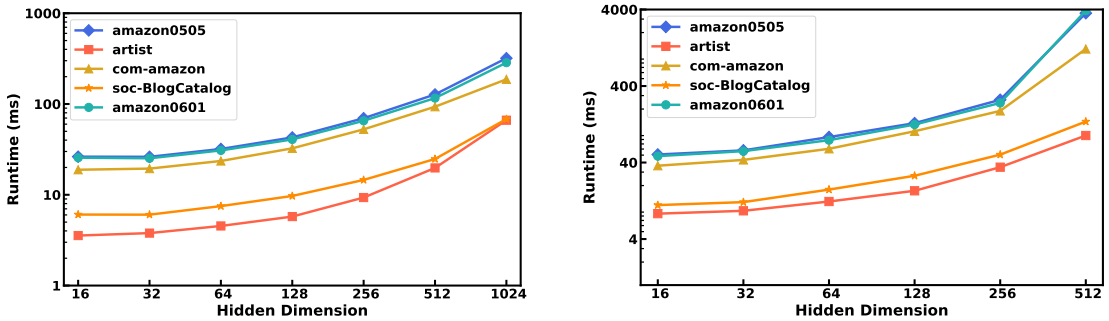
在这一节中，我们将对 GNN 的隐藏维度、GNNA 预处理开销和 **Decider** 的参数选择等问题进行探讨。



(a) III 型图节点重编号前后的加速比　　(b) III 型图节点重编号的开销

图 4-6　节点重编号的加速比和开销

4.4.1　GNN 的隐藏维度



(a) GCN 隐藏维度　　(b) GIN 隐藏维度

图 4-7　隐藏维度与延迟的关系

隐藏维度（hidden dimension）是 GNN 中用于表示节点特征的向量空间大小，它直接影响每一层神经网络的参数数量和计算复杂度。在本项实验中，我们重点分析了 GNN 架构中隐藏维度大小对两种主流模型——GCN 和 GIN——运行性能的影响。如图 4-7所示，随着隐藏维度的增加，两种模型的运行时间均有所上升。这一趋势主要源于更大的隐藏维度带来了更多的计算和内存操作。在 GCN 中，聚合阶段会涉及更多的加法运算和数据移动操作，节点更新阶段的嵌入矩阵尺寸也随之扩大，从而增加了整体计算负担。

值得注意的是，GIN 的运行时间增长幅度远高于 GCN。这是因为 GIN 模型本身具有更深的网络结构（5 层），相比之下 GCN 仅为 2 层，导致隐藏维度带来的开销在 GIN 中呈线性甚至更高阶的叠加效应。此外，由于，GCN 节点维度缩减（DGEMM）总是在聚合之前进行，这大大减少了聚合阶段的数据移动和线程同步开销。GIN 的聚合操作设计更复杂，在每一层的计算密度和内存访问强度

都高于 GCN，因此在隐藏维度扩大时其性能退化更为明显。

4.4.2 开销分析

在 GNNA 中，使用社区感知的节点重新编号来优化图的局部性，进而提升 GNN 的性能。尽管我们选用了轻量级的重新编号算法，但仍然需要在读取图后进行一次性预处理。为了评估这一开销对整体性能的影响，我们使用 GCN 模型对 III 型图在训练阶段的性能开销进行了深入分析。III 型图的节点和边的数量，节点重编号的开销较大，同时结构高度不规则，容易从节点重编号中收益。图 4-6b展示了训练过程中的重编号开销比例。可以看到，节点重新编号阶段的开销仅最高占训练时间的 1.52%，平均仅占 1.14%。这一处理过程所带来的时间成本非常小，完全可以在训练或推理的运行时周期中进行摊销，从而避免对整体性能产生显著影响。

结合上一节的优化分析我们可以发现，这种一次性预处理策略不仅代价低，而且带来了显著的下游性能提升。通过对图结构的重组优化了图中节点的访问局部性，从而增强了后续计算过程中缓存命中率和数据传输效率。因此，社区感知的节点重新编号不仅在理论上具备可行性，在实践中也被验证为高效且轻量的优化方案。

4.4.3 参数选择

为了展示 GNNA 中的轻量级分析模型 **Decider** 在内核参数选择中的有效性，我们在不同的数据集上进行了实验。我们选择了两个具有代表性的图数据集：*amazon0505* 和 *soc-BlogCatalog*。实验结果如图 4-8所示，**Decider** 的参数选择策略能够为上述四种设置精确地找到最优的低延迟设计。这证明了我们的分析模型在辅助参数选择以优化 GNN 计算性能方面的有效性。

4.5 本章小结

本章对 GNNA 在多种 GNN 模型和三类图数据集上的推理与训练性能进行了系统评估。实验结果表明，GNNA 在推理任务中相较于 DGL 平均加速可达

图 4-8 Decider 的参数选择有效性验证，左侧为 *amazon0505* 数据集，右侧为 *soc-BlogCatalog* 数据集

3.13×（GCN）和 1.35×（GIN），在训练任务中也实现了最高 2.00× 的性能提升。这样的性能优势得益于 GNNA 基于图结构特征的任务划分策略和高效的内存优化机制，尤其在大规模、结构不规则的图上依然展现出显著效果。同时，我们还对 GNNA 的预处理开销、参数选择等进行了深入分析，验证了其在实际应用中的有效性和通用性。

5 总结与展望

本工作提出了一种面向 GPU 的自适应高效 GNN 运行时系统 GNNA。我们探索了 GNN 输入特征在系统优化中的潜力，并基于此设计了一系列面向 GNN 的系统级优化策略：

（1）二维任务调度机制和

（2）专用内存优化技术

（3）输入属性提取和轻量级参数决策模型

以提升性能与适应性。我们在多个主流 GNN 模型和广泛的数据集上进行了大量实验，验证了 GNNA 在训练与推理中的显著性能优势。总体而言，GNNA 为用户提供了一个系统化、全面化的 GPU GNN 加速工具。

尽管 GNNA 在大多数图类型和模型上表现出色，但在某些特定图结构（如 II 型图）上仍存在进一步优化的空间。未来的工作可以从两个方向展开：一方面，进一步引入结构感知的任务划分机制，以提升在小规模、稠密图上的性能；另一方面，为 GNNA 添加用户定义函数（UDF）支持，以便用户可以根据特定需求自定义任务划分和内存优化策略。通过这些改进， GNNA 有望在更广泛的应用场景中实现更高的性能和灵活性。

参考文献

[1] GROVER A, LESKOVEC J. node2vec: Scalable feature learning for networks [C]//Proceedings of the 22nd ACM international conference on Knowledge discovery and data mining (SIGKDD). 2016.

[2] PEROZZI B, AL-RFOU R, SKIENA S. DeepWalk: Online Learning of Social Representations[C]//The 20th ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD). 2014.

[3] LUO D, NIE F, HUANG H, et al. Cauchy graph embedding[C]//The 28th International Conference on Machine Learning (ICML). 2011.

[4] LUO D, DING C, HUANG H, et al. Non-negative laplacian embedding[C]//Ninth IEEE International Conference on Data Mining (ICDM). 2009.

[5] SALA A, ZHENG H, ZHAO B Y, et al. Brief Announcement: Revisiting the Power-Law Degree Distribution for Social Graph Analysis[C]//Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC). 2010.

[6] HAN M, DAUDJEE K, AMMAR K, et al. An experimental comparison of pregel-like graph processing systems[J]. The VLDB Endowment, 2014.

[7] FEY M, LENSSEN J E. Fast Graph Representation Learning with PyTorch Geometric[C]//ICLR Workshop on Representation Learning on Graphs and Manifolds (ICLR). 2019.

[8] WANG M, YU L, ZHENG D, et al. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs[J/OL]. ICLR Workshop on Representation Learning on Graphs and Manifolds, 2019. https://arxiv.org/abs/1909.01315.

[9] FEY M, LENSSEN J E. PyTorch Extension Library of Optimized Scatter Operations[EB/OL]. 2019. https://github.com/rusty1s/pytorch_scatter.

[10] Nvidia. CUDA Sparse Matrix library (cuSPARSE)[EB/OL]. developer.nvidia.com/cusparse.

[11] LOUKAS A, VANDERGHEYNST P. Spectrally Approximating Large Graphs

with Smaller Graphs[C/OL]//DY J, KRAUSE A. Proceedings of Machine Learning Research: Proceedings of the 35th International Conference on Machine Learning: vol. 80. PMLR, 2018: 3237-3246. https://proceedings.mlr.press/v80/loukas18a.html.

[12] BRAVO HERMSDORFF G, GUNDERSON L. A unifying framework for spectrum-preserving graph sparsification and coarsening[J]. Advances in Neural Information Processing Systems, 2019, 32.

[13] BENCZÚR A A, KARGER D R. Approximating st minimum cuts in Õ (n 2) time[C]//Proceedings of the twenty-eighth annual ACM symposium on Theory of computing. 1996: 47-55.

[14] SPIELMAN D A, SRIVASTAVA N. Graph sparsification by effective resistances [J]. SIAM Journal on Computing, 2011, 40(6): 1913-1926.

[15] SPIELMAN D A, TENG S H. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems[J]. SIAM Journal on Matrix Analysis and Applications, 2014, 35(3): 835-885.

[16] SERRANO M Á, BOGUNÁ M, VESPIGNANI A. Extracting the multiscale backbone of complex weighted networks[J]. Proceedings of the national academy of sciences, 2009, 106(16): 6483-6488.

[17] JIN W, ZHAO L, ZHANG S, et al. Graph Condensation for Graph Neural Networks[J]. arXiv preprint arXiv:2110.07580, 2021.

[18] HAMILTON W, YING Z, LESKOVEC J. Inductive representation learning on large graphs[C]//Advances in neural information processing systems. 2017: 1024-1034.

[19] CHEN J, MA T, XIAO C. Fastgcn: fast learning with graph convolutional networks via importance sampling[J]. arXiv preprint arXiv:1801.10247, 2018.

[20] HUANG W, ZHANG T, RONG Y, et al. Adaptive sampling towards fast graph representation learning[J]. Advances in neural information processing systems, 2018, 31.

[21] ZENG H, ZHOU H, SRIVASTAVA A, et al. Graphsaint: Graph sampling based

inductive learning method[J]. arXiv preprint arXiv:1907.04931, 2019.

[22] CHIANG W L, LIU X, SI S, et al. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks[C]//Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. 2019: 257-266.

[23] KARYPIS G, KUMAR V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs[J/OL]. SIAM J. Sci. Comput., 1998, 20(1): 359-392. https://doi.org/10.1137/S1064827595287997. DOI: 10.1137/S1064827595287997.

[24] FU Q, JI Y, HUANG H H. TLPGNN: A Lightweight Two-Level Parallelism Paradigm for Graph Neural Network Computation on GPU[C]//Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing. 2022: 122-134.

[25] TIAN C, MA L, YANG Z, et al. Pcgcn: Partition-centric processing for accelerating graph convolutional network[C]//2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 2020: 936-945.

[26] ANSEL J, YANG E, HE H, et al. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation[C/OL]// ASPLOS '24: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2. La Jolla, CA, USA: Association for Computing Machinery, 2024: 929-947. https://doi.org/10.1145/3620665.3640366. DOI: 10.1145/3620665.3640366.

[27] REED J K, DEVITO Z, HE H, et al. Torch.fx: Practical Program Capture and Transformation for Deep Learning in Python[EB/OL]. 2022. https://arxiv.org/abs/2112.08429. arXiv: 2112.08429 [cs.LG].

[28] PyTorch. TorchDynamo[EB/OL]. 2023. https://github.com/pytorch/torchdynamo.

[29] WANG D, ZHU W, LING L, et al. ML-Triton, A Multi-Level Compilation and Language Extension to Triton GPU Programming[EB/OL]. 2025. https://arxiv.o

rg/abs/2503.14985. arXiv: 2503.14985 [cs.CL].

[30] WU Y, MA K, CAI Z, et al. Seastar: vertex-centric programming for graph neural networks[C]//Proceedings of the Sixteenth European Conference on Computer Systems. 2021: 359-375.

[31] SOHRABIZADEH A, WANG J, CONG J. End-to-End Optimization of Deep Learning Applications[C]//FPGA. 2020: 133-139.

[32] ZHANG X, WANG J, ZHU C, et al. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs[C]//ICCAD. 2018: 1-8.

[33] ZENG H, PRASANNA V. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms[C]//Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. 2020: 255-265.

[34] CHEN X, WANG Y, XIE X, et al. Rubik: A hierarchical architecture for efficient graph neural network training[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2021.

[35] LIANG S, WANG Y, LIU C, et al. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks[J]. IEEE Transactions on Computers, 2020, 70(9): 1511-1525.

[36] YAN M, DENG L, HU X, et al. Hygcn: A gcn accelerator with hybrid architecture[C]//2020 IEEE International Symposium on High Performance Computer Architecture (HPCA). 2020: 15-29.

[37] SOHRABIZADEH A, CHI Y, CONG J. StreamGCN: Accelerating Graph Convolutional Networks with Streaming Processing[C]//2022 IEEE Custom Integrated Circuits Conference (CICC). 2022: 1-8.

[38] KIPF T N, WELLING M. Semi-supervised classification with graph convolutional networks[J]. International Conference on Learning Representations (ICLR), 2017.

[39] HAMILTON W, YING Z, LESKOVEC J. Inductive representation learning on large graphs[C]//Advances in neural information processing systems (NeurIPS).

2017.

[40] VELIČKOVIĆ P, CUCURULL G, CASANOVA A, et al. Graph Attention Networks[C]//International Conference on Learning Representations (ICLR). 2018.

[41] KASPAR R, HORST B. Graph classification and clustering based on vector space embedding[M]. World Scientific, 2010.

[42] GIBERT J, VALVENY E, BUNKE H. Graph embedding in vector spaces by node attribute statistics[J]. Pattern Recognition, 2012.

[43] DURAN A G, NIEPERT M. Learning graph representations with embedding propagation[C]//Advances in neural information processing systems (NeurIPS). 2017.

[44] CHEN H, LI X, HUANG Z. Link prediction approach to collaborative filtering[C]//Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL). 2005.

[45] KUNEGIS J, LOMMATZSCH A. Learning spectral graph transformations for link prediction[C]//Proceedings of the 26th Annual International Conference on Machine Learning (ICML). 2009.

[46] TYLENDA T, ANGELOVA R, BEDATHUR S. Towards time-aware link prediction in evolving social networks[C]//Proceedings of the 3rd workshop on social network mining and analysis. 2009.

[47] BORDES A, USUNIER N, GARCIA-DURAN A, et al. Translating Embeddings for Modeling Multi-relational Data[C]//Advances in Neural Information Processing Systems (NeurIPS). 2013.

[48] DUVENAUD D, MACLAURIN D, AGUILERA-IPARRAGUIRRE J, et al. Convolutional networks on graphs for learning molecular fingerprints[J]. arXiv preprint, 2015.

[49] KHORASANI F, VORA K, GUPTA R, et al. CuSha: vertex-centric graph processing on GPUs[C]//Proceedings of the 23rd international symposium on High-performance parallel and distributed computing (HPDC). 2014.

[50] NODEHI SABET A H, QIU J, ZHAO Z. Tigr: Transforming Irregular Graphs

for GPU-Friendly Graph Processing[C]//Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). 2018.

[51] WANG Y, DAVIDSON A, PAN Y, et al. Gunrock: A high-performance graph processing library on the GPU[C]//Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP). 2016.

[52] LIU H, HUANG H H. Enterprise: breadth-first graph traversal on GPUs[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2015.

[53] LIU H, HUANG H H. SIMD-x: Programming and processing of graph algorithms on gpus[C]//USENIX Annual Technical Conference (ATC). 2019.

[54] KYROLA A, BLELLOCH G, GUESTRIN C. Graphchi: Large-scale graph computation on just a PC[C]//The 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI). 2012.

[55] ROY A, MIHAILOVIC I, ZWAENEPOEL W. X-stream: Edge-centric graph processing using streaming partitions[C]//Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP). 2013.

[56] PASZKE A, GROSS S, MASSA F, et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library[G]//Advances in Neural Information Processing Systems (NeurIPS). 2019.

[57] ABADI M, BARHAM P, CHEN J, et al. Tensorflow: A system for large-scale machine learning[C]//12th USENIX symposium on operating systems design and implementation (OSDI). 2016.

[58] XU K, HU W, LESKOVEC J, et al. How Powerful are Graph Neural Networks? [C]//International Conference on Learning Representations (ICLR). 2019.

[59] KHAYYAT Z, AWARA K, ALONAZI A, et al. Mizan: A System for Dynamic Load Balancing in Large-Scale Graph Processing[C]//Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys). 2013.

[60] FORTUNATO S. Community detection in graphs[J]. Physics reports, 2010.

[61] LANCICHINETTI A, FORTUNATO S, RADICCHI F. Benchmark graphs for testing community detection algorithms[J]. Physical review E, 2008.

[62] NEWMAN M E. Spectral methods for community detection and graph partitioning[J]. Physical Review E, 2013.

[63] HENDRICKSON B, KOLDA T G. Graph partitioning models for parallel computing[J]. Parallel computing, 2000.

[64] ARAI J, SHIOKAWA H, YAMAMURO T, et al. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis[C]//2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). 2016.

[65] BOLDI P, ROSA M, SANTINI M, et al. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks[C]// Proceedings of the 20th international conference on World wide web (WWW). 2011.

[66] RAGHAVAN U N, ALBERT R, KUMARA S. Near linear time algorithm to detect community structures in large-scale networks[J]. Physical review E, 2007.

[67] KARANTASIS K I, LENHARTH A, NGUYEN D, et al. Parallelization of reordering algorithms for bandwidth and wavefront reduction[C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC). 2014.

[68] CUTHILL E, MCKEE J. Reducing the Bandwidth of Sparse Symmetric Matrices [C]//Proceedings of the 1969 24th National Conference. 1969.

[69] Nvidia. RTX 4060[EB/OL]. https://www.nvidia.com/en-us/geforce/graphics-cards/40-series/rtx-4060-4060ti/.

[70] YANG C, BULUÇ A, OWENS J D. Design principles for sparse matrix multiplication on the GPU[C]//European Conference on Parallel Processing. 2018.

[71] MA L, YANG Z, MIAO Y, et al. Neugraph: parallel deep neural network computation on large graphs[C]//USENIX Annual Technical Conference (ATC'19).

[72] YING R, YOU J, MORRIS C, et al. Hierarchical Graph Representation Learning with Differentiable Pooling[C]//The 32nd International Conference on Neural

Information Processing Systems (NeurIPS). 2018.

[73] KERSTING K, KRIEGE N M, MORRIS C, et al. Benchmark Data Sets for Graph Kernels[EB/OL]. 2016. http://graphkernels.cs.tu-dortmund.de.

[74] LESKOVEC J, KREVL A. SNAP Datasets: Stanford Large Network Dataset Collection[Z]. http://snap.stanford.edu/data. 2014.

致 谢

论文致谢页页码一般应该大于等于 40，描述实际工作部分篇幅应在 20 页以上。如果达不到要求可能直接二辩了!!!

致谢属于论文的辅文部分。使用第一人称，采用散文体，对指导教师以及协助完成设计的有关人员表示谢意，并可简述自己通过本次毕业设计的体会，注意致谢是查重最容易出问题的地方，请千万不要照搬别人写的，2025 年开始致谢放到论文最后面