中国图计算挑战赛提交文档

基于稀疏矩阵乘法和 SIMD 向量化的高效 GCN 计算加速方案

单　　位：重庆大学

队伍名称：makka_pakka

指导老师：谭玉娟、任骜、刘铎

团队成员：甘雁、王玉娇、郭佳艺

1 基本算法介绍

图卷积网络（Graph Convolutional Network，GCN）是一种用于处理图数据的深度学习模型。在海量数据和超强计算资源的推动下，GCN 因其强大的表征能力，广泛应用于自然语言处理、计算机视觉、推荐系统等领域。然而，在许多实际应用中，图数据结构非常复杂，并且包含大量的节点和边，大规模、高稀疏性、不规则的图数据分布给 GCN 计算带来了巨大挑战。如何充分利用系统资源提升 GCN 推理效率是目前亟需解决的难题。本节首先阐述 GCN 基本原理，然后针对 GCN 计算过程中的瓶颈问题展开详细分析，最后提出我们的解决方案。

1.1 GCN 基本原理

GCN 是一种基于图数据的卷积神经网络，它通过在图上执行卷积操作来学习节点的表示，充分利用了图结构数据中节点之间的关系。对于图$G = (V, E)$，V为节点集合，E为边集合。矩阵X_{N*F}表示图的特征矩阵，其中N代表节点数，F代表特征向量的维度。GCN 通常由多个图卷积层堆叠而成（如图 1.1 所示），单个图卷积层的传播公式如下：

$$X^{l+1} = \sigma\left(\hat{A}X^{l}W^{l}\right) \tag{1.1}$$

其中l表示层数，$\sigma(\cdot)$为非线性激活函数，例如 ReLU。$\hat{A} \in \mathbb{R}^{N*N}$为归一化后的图邻接矩阵，$X^{l} \in \mathbb{R}^{N*F_l}$为输入节点特征矩阵，$W^{l} \in \mathbb{R}^{F_l*F_{l+1}}$为权重矩阵，$X^{l+1} \in \mathbb{R}^{N*F_{l+1}}$为输出节点特征矩阵。

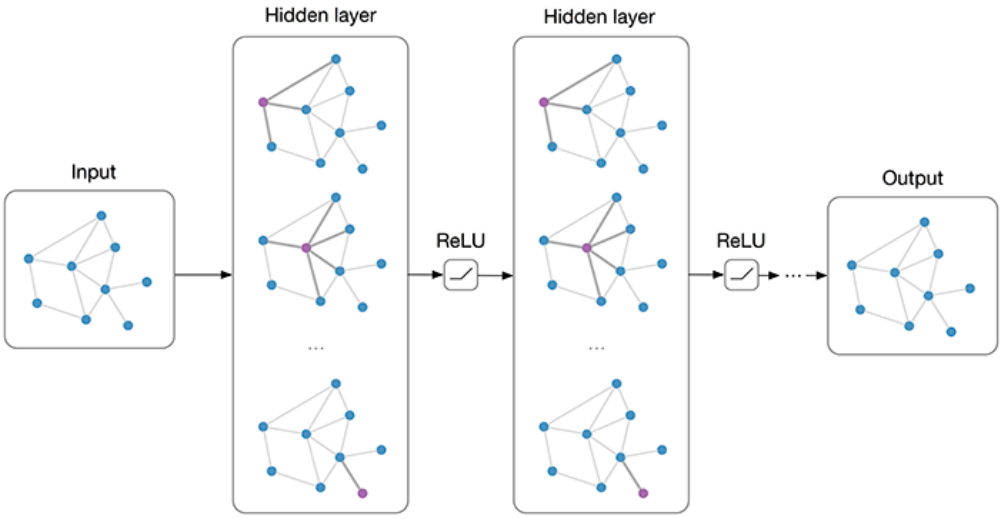


图 1.1 多层图卷积网络结构示意图

为了在信息传递的过程中保持特征矩阵X的原有分布，防止一些度数高的节

点和度数低的节点在特征分布上产生较大的差异，需要对邻接矩阵进行归一化处理，归一化邻接矩阵$\hat{A}$的计算公式如下：

$$\hat{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}} \tag{1.2}$$

其中$D \in \mathbb{R}^{N*N}$为节点度矩阵，$A \in \mathbb{R}^{N*N}$为邻接矩阵。

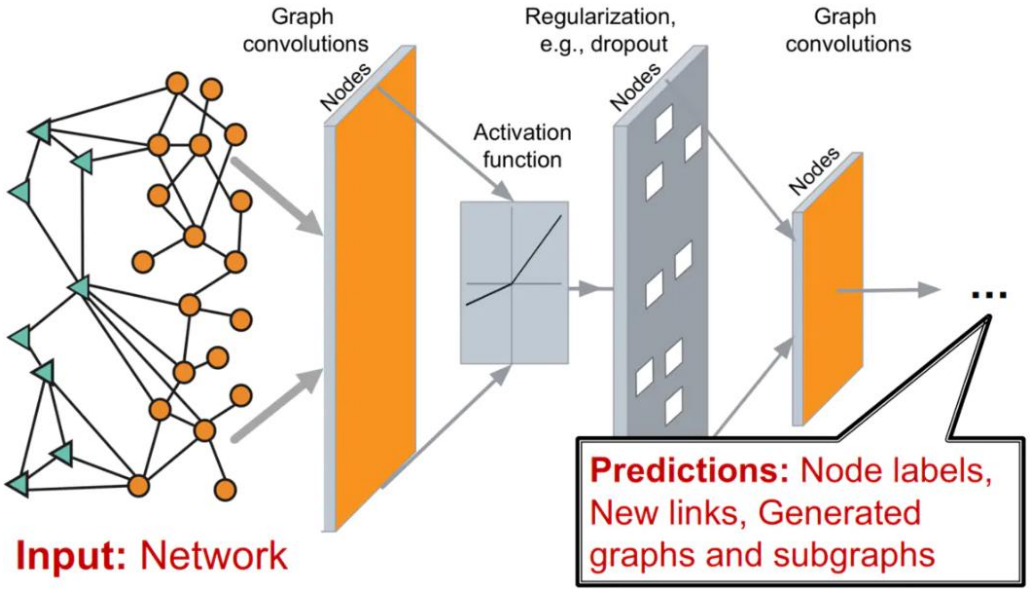


图 1.2 图卷积网络操作流程图

GCN 的计算过程大致如下（如图 1.2 所示）：

（1）图表示。将输入的图数据表示为邻接矩阵或者邻接表的形式。邻接矩阵表示了图中节点之间的连接关系，而邻接表则以每个节点为索引，存储了该节点相邻节点的列表。

（2）邻接矩阵归一化。对邻接矩阵进行归一化处理，常用的归一化方式是使用度矩阵的逆和邻接矩阵相乘，得到归一化后的邻接矩阵。

（3）特征初始化。为每个节点初始化一个特征向量作为节点的初始表示。

（4）图卷积层。GCN 的核心是多个图卷积层的堆叠。每个图卷积层都会更新节点的表示，同时保留了节点的邻居信息。图卷积操作包括聚合邻居节点信息、线性变换、非线性激活以及更新节点表示。

① 聚合邻居节点信息：对于每个节点，GCN 会聚合其邻居节点的表示信息。这可以通过将邻居节点的特征向量相加或平均来实现，类似于传统卷积神经网络中的池化操作。

② 线性变换：聚合后的邻居信息再与节点自身的特征向量进行线性变换，这通常使用与权重矩阵相乘的方式进行实现。

③ 非线性激活：将线性变换后的结果输入到非线性激活函数中，如 ReLU、

Softmax 等。

④ 更新节点表示：将非线性激活函数的输出作为节点的新表示，用于下一层的计算。

（5）堆叠卷积层。多个图卷积层堆叠形成一个深层的 GCN 模型。每一层的输出作为下一层的输入，逐层迭代更新节点表示。

（6）预测或分类：在 GCN 的最后一层输出上，可以根据具体的任务进行预测或分类。

1.2 GCN 计算问题分析

GCN 采用图卷积操作在图数据上进行信息传递和特征学习。图卷积操作结合了节点的特征和节点之间的连接关系，通过迭代更新节点特征向量来捕捉节点的局部和全局信息。如图 1.3 所示，GCNs 主要包括两个阶段：*Aggregation* 和 *Combination*，各阶段的执行模式并不相同（如表 1.1 所示），并且占据了 GCNs 的主要执行时间。

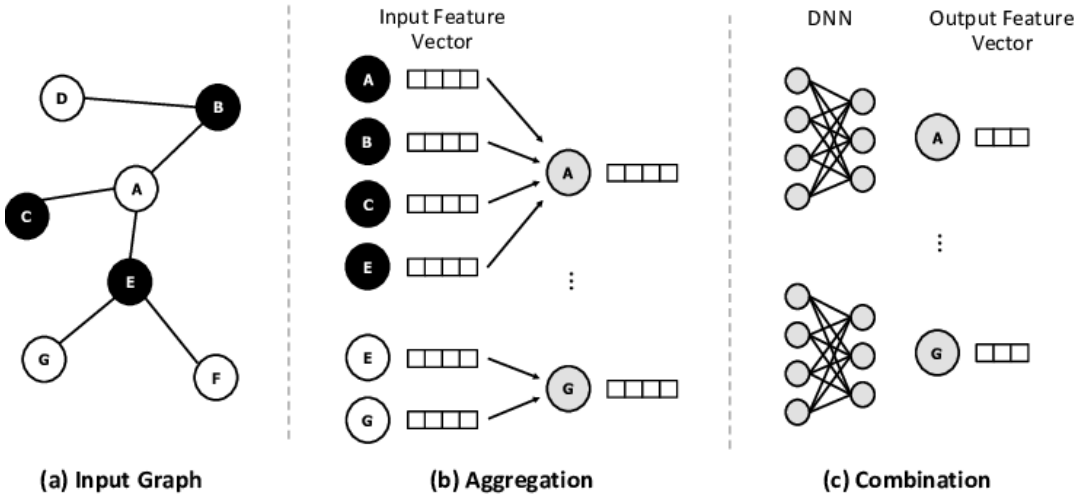


图 1.3 图卷积网络的 *Aggregation* 和 *Combination* 阶段

表 1.1 *Aggregation* 阶段和 *Combination* 阶段的不同执行模式

	Aggregation	***Combination***
访问模式	间接、不规则	直接、规则
数据可重用性	低	高
计算模式	动态、不规则	静态、规则
计算机密集性	低	高
运行瓶颈	内存	计算

GCN 在进行图卷积计算时通常表现出以下特征：

（1）数据局部性差，使得内存访问开销大

在 *Aggregation* 阶段，即对应式(1.1)中的 AX 过程，每个节点都会访问和汇聚邻居节点的信息。然而，现实世界中的图大多是稀疏的，即节点之间的连接较少，对应的邻接矩阵中大部分元素将是 0，从而导致大量空间浪费。并且由于图数据中不同节点的邻居节点数量不同，邻居节点在内存中的位置也存在较大差异。因此访问图数据的不同元素需要跳转到不同的内存位置，从而导致较大的内存访问开销。

（2）邻居节点数目不固定，使得并行计算时存在负载不均衡

图本身是一种不规则的数据结构，在 *Aggregation* 阶段，每个节点的表示是通过聚合其邻居节点的特征来计算的。然而，不同节点的邻居节点数量是不固定的，使得不同节点的计算工作量可能会有较大差异，这将导致严重的工作负载不平衡问题。

（3）节点特征维度大，使得计算复杂度高

在 *Combination* 阶段，即对应式(1.1)中的 XW 过程，涉及节点特征矩阵和权重矩阵之间的矩阵向量乘法(Matrix Vector Multiplication, MVM)。*Combination* 阶段使用多层感知机（Multi Layer Perceptron, MLP）将每个节点的特征向量转换为一个新的特征向量，MLP 通常用 MVM 表示。由于节点的特征矩阵维度通常较大，GCN 在进行特征变换时需要处理更多维度的特征矩阵，从而造成了更高的计算时间开销。

（4）卷积计算过程中存在冗余操作，使得数据重复读取次数多

首先，在传统 GCN 计算过程中，通常先加载数据集，将输入的图数据表示为邻接矩阵或者邻接表的形式。然后遍历邻接矩阵，对其进行归一化处理。在这个过程中，存在对邻接矩阵中数据值的重复读写。其次，在卷积计算过程中，GCN 会聚合其邻居节点的表示信息，并更新自身的特征向量，再通过激活函数传递到下一卷积层中。在这个过程中，存在对节点特征矩阵的重复读写。最后，随着卷积层数的增多，如果给每一个卷积层都分配一个矩阵存储中间计算结果，将会占用较大的存储空间。对数据的重复读取进一步限制了 GCN 计算效率的提升。

综上所述，针对目前 GCN 计算过程中难以有效访问和处理大量的稀疏数据、忽略了节点特征聚合过程中存在的负载不均衡、无法充分降低高维度特征矩阵的计算复杂度、以及卷积计算过程中存在重复数据读取和冗余内存分配等问题，需要图数据的压缩存储、内存访问连续性设计、并行计算中的负载均衡、矩阵分块、SIMD 向量化、以及冗余操作融合等方面进行针对性的优化，从而达到充分利用系统资源，提升 GCN 计算效率的目的。

1.3 GCN 优化方法概述

我们将比赛所给的由两个卷积层构成的GCN计算过程形式化描述为式(1.3)，第一层激活函数使用 ReLU，第二层激活函数使用 LogSoftmax。

$$\hat{Y} = f(X, A) = LogSoftmax\left(\hat{A}ReLU\left(\hat{A}XW^0\right)W^1\right) \tag{1.3}$$

其中$\hat{A}$为归一化后的邻接矩阵，X为特征矩阵，W^0为第一层权重矩阵，W^1为第二层权重矩阵。ReLU 函数和 LogSoftmax 函数分别定义为

$$ReLU(x) = max(0, x) \tag{1.4}$$

$$LogSoftmax\left(X_{i,j}^l\right) = \left(X_{i,j}^l - X_{i,max}^l\right) - log\left(\sum_{c=0}^{F_l-1} e^{x_{i,c}^l - x_{i,max}^l}\right) \tag{1.5}$$

$$X_{i,max}^l = max\left(X_{i,0}^l, \dots, X_{i,F_{l-1}}^l\right) \tag{1.6}$$

为了减少 *Aggregation* 阶段节点特征聚合的时间开销，我们采用优先执行 *Combination* 阶段的方式，即$\left(A \times (XW)\right)$。通过减少特征长度，实现 *Aggregation* 阶段的计算加速。

我们的主要贡献如下：第一，针对 GCN 计算过程中的数据局部性差，难以有效访问和处理大量的稀疏数据的问题，我们引入了压缩稀疏行（Compressed Sparse Row, CSR）格式存储大量稀疏图数据，它通过仅存储非零元素及其对应的行索引和列指针，有效地压缩了稀疏矩阵的存储空间。同时在 CSR 格式中，所有的非零元素都被存储在一个一维数组中（values 数组），非零元素在内存中是连续存储的，当对这些元素进行遍历或操作时，可以减少内存访问的不连续性。此外，GCNs 的邻居节点特征汇聚过程中涉及大量矩阵，而这些矩阵通常以二维矩阵的形式存储。我们使用一维数组来模拟存储二维矩阵，保证数据在内存中的连续存储，进一步提升了数据访问效率。第二，针对 GCN 计算过程中由邻居节点数目不固定导致的并行计算时的负载不均衡问题，我们在 *Aggregation* 阶段设计了一种节点分散处理方法，该方法采用以边为中心的执行模式，并使用了 SIMD 向量化技术，将每个节点的特征向量内的元素聚合分配给所有核，从而充分提升节点特征聚合效率。第三，针对 GCN 计算过程中的节点特征维度大，难以高效实现特征矩阵和权重矩阵之间的 MVM 问题，我们利用矩阵分块、SIMD 向量化、寄存器加速、内存重排等技术对节点特征矩阵和权重矩阵之间的 MVM 过程进行充分优化，从而实现了 *Combination* 阶段的计算加速。第四，针对 GCN 计算过程中的冗余数据操作和重复内存分配问题，我们对 GCNs 计算过程中的固有数据流进行操作融合。首先，我们在图数据预处理阶段，将图数据存储为 CSR 格式的同时，直接对邻接矩阵进行归一化处理，并将归一化后的值存入 values 数组的对应位置，减少了邻接矩阵数据的重复读写。其次，为了提高计算效率，我们在图卷积操作后立即应用激活，分别将 ReLU 和 LogSoftmax 运算与 AX 函数

完全融合，进一步减少更新后的特征矩阵数据重复读写次数。最后，由于整个GCNs包含两个卷积层，每个卷积层的特征聚合及更新过程中会产生许多中间矩阵，我们对这些中间矩阵进行了重分配，避免使用多个中间矩阵，从而减少了内存开销。进一步加速了GCN的整体计算过程。

在给定的由 1024 个顶点和 4096 条边组成的图数据集上，顶点特征长度为64，第一层顶点特征长度和第二层顶点特征长度分别为 16 和 8。为消除随机误差，我们在相同平台上对两段程序代码运行 2000 次，并取平均值作为相应的执行时间。与运行在 Intel Xeon CPU 上的 Baseline 推理框架相比，我们的方案将执行时间由原来的平均 11.25ms 降低至平均 0.36ms，实现了约 31.28 倍的加速。为了评估我们所提出的 GCN 推理加速方法的泛化性，我们还使用 RMAT（https://github.com/farkhor/PaRMAT）随机生成了不同大小的图文件及节点特征矩阵，在由 20480 个顶点组成的图数据集上，顶点特征长度为 20480，第一层顶点特征长度和第二层顶点特征长度分别为 16 和 8。我们的方案将执行时间由原来的平均 194.77ms 降低至平均 8.08ms，实现了约 24.11 倍的加速；在由 1M 个顶点组成的图数据集上，顶点特征长度为 1M，第一层顶点特征长度和第二层顶点特征长度分别为 16 和 8。我们的方案将执行时间由原来的平均 5649.56ms 降低至平均 199.50ms，实现了约 28.31 倍的加速。我们的方案显著提高了数据局部性并减少了不必要的计算，同时能够达到与传统算法相当的测试精度。

2 设计思路和方法

GCN 是一种有效表示和处理图数据的模型，它通常包含两个具有不同执行模式的关键阶段：*Aggregation* 和 *Combination*。前一阶段主要用于将节点的邻居信息进行聚合。由于图本身是一种不规则的数据结构，不同节点的邻居节点数目和内存位置是不固定的，因此 *Aggregation* 阶段也会相应地表现出不规则的执行模式。后一阶段使用 MLP 更新特征向量，它具有与传统神经网络相似的规则模式。**为了实现更高推理性能的 GCNs，需要设计新型的 GCN 计算架构，以充分缓解 *Aggregation* 阶段的不规则性，同时利用 *Combination* 阶段的规则性，以及节点间的高并行度和高可重用性。**本节首先定量地描述和识别 GCNs 的混合执行模式，接下来解释 GCN 加速方案的设计动机，然后分别从 *Aggregation* 阶段内存访问和计算优化、*Combination* 阶段计算加速、固有数据流融合三个方面详细描述设计思路和方法。

2.1 GCN 计算过程定量描述

我们在 Intel Xeon CPU 上使用不同大小的数据集对基础 GCN 计算框架进行

定量表征，图文件及节点特征矩阵使用 RMAT（https://github.com/farkhor/PaRMAT）随机生成，不同阶段的执行时间占比如图 2.1 所示。

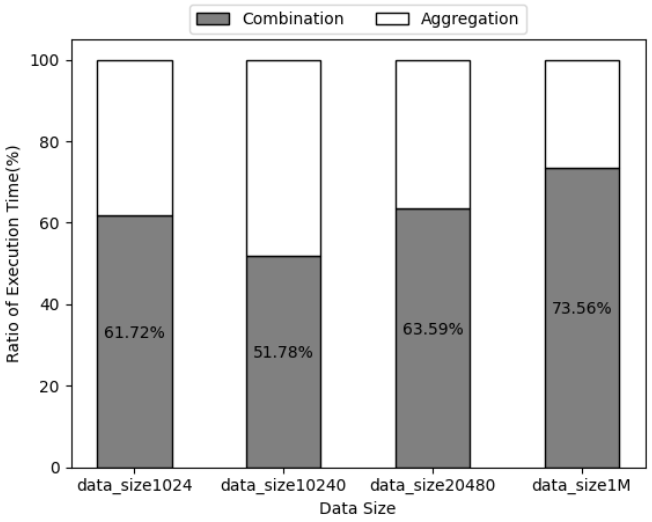


图 2.1 图卷积网络两个阶段的执行时间分解

根据执行时间分解占比图，可以看到 *Aggregation* 和 *Combination* 阶段都占用了大量的执行时间。因此，加速 *Aggregation* 和 *Combination* 阶段是提升 GCN 推理效率的关键。图 2.1 显示了它们在不同数据集上的执行时间比，由于数据集大小和特征向量的长度不同，*Aggregation* 和 *Combination* 阶段的占比也不同。

在 *Aggregation* 阶段，邻居节点聚合操作严重依赖于固有的随机和稀疏图结构，从而导致大量的动态计算和不规则访问。邻居索引的高度随机性给缓存预取带来了巨大挑战，间接和不规则的访问使得 *Aggregation* 阶段的数据预取无效，因为如果不事先知道邻居的索引，就很难预测数据地址，使得预取数据的过程中产生大量无效的内存访问开销。

在 *Combination* 阶段，GCNs 使用共享的基于 MLP 的神经网络对每个节点执行 MVM，该神经网络执行静态和常规的计算和访问。MLP 的权重矩阵在节点之间广泛共享，并且 MVM 的计算量非常大。

综上所述，*Aggregation* 阶段的执行模式是动态且不规则的，它受到内存的限制，而 *Combination* 阶段的执行模式是静态和规则的，它的主要瓶颈在于计算。如何缓解 *Aggregation* 阶段的不规则性，同时利用 *Combination* 阶段的规则性是提升 GCN 推理速度的关键。本文接下来分别从 *Aggregation* 阶段的内存访问和计算优化、*Combination* 阶段的计算加速、以及 GCN 计算过程中的固有数据流融合三个方面来叙述我们的设计思路和方法。

2.2 *Aggregation* 阶段内存访问和计算优化

在 *Aggregation* 阶段，每个节点均会访问并汇聚其邻居节点的信息。鉴于图数据的稀疏性和不规则性，访问图数据的不同邻居节点必须跳转至不同的内存位置，因而导致了较大的内存访问开销和计算代价。针对这一问题，我们可采纳图数据压缩存储的策略，仅存储非零元素及其位置，从而大幅减少存储空间的开销。此外，优化内存访问模式以提升图数据中节点的邻居节点在内存中的连续性，同样是提高 Aggregation 阶段计算效率的重要手段。最后，充分利用节点内并行性，也为 *Aggregation* 阶段计算优化提供了潜在的价值。

2.2.1 CSR 存储格式

GCNs 在聚合邻居节点信息的过程中，通常需要遍历图的邻接矩阵以访问邻居节点信息。尽管邻接矩阵能够迅速判断两个节点之间是否有边相连，然而在实际应用中，图数据大多呈现稀疏性，即节点之间的边相对较少，导致邻接矩阵中大部分元素为零，仅有极少元素为 1，从而造成大量存储空间的浪费。因此，选择适当的图数据存储结构对于优化 GCN 计算效率至关重要。

常用的图数据存储格式包括邻接表、COO（Coordinate）、CSR（Compressed Sparse Row）、CSC（Compressed Sparse Column）、DOK（Dictionary of Keys）等，这些图数据表现形式的特点如表 2.1 所示。

表 2.1 图数据表示形式特点分析

	存储占用低	内存局部性好	数据访问效率高
邻接表	√	√	×
COO 格式	√	×	×
DOK 格式	√	×	×
CSC 格式	√	√	√
CSR 格式	√	√	√

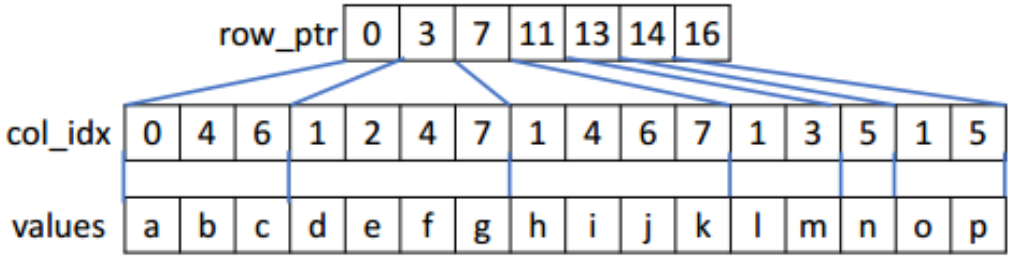
邻接表可以更好地处理稀疏图，但是在判断两个节点之间是否有边相连时需要遍历链表，因此数据查询效率较低；在 COO 和 DOK 格式中，稀疏矩阵的非零元素以三元组的形式存储，包括行索引、列索引及对应的值，减小了存储空间的占用，但在访问非零元素时，需要跳跃式地访问内存，从而导致较多的缓存缺失和额外的访问延迟。

相比之下， CSR 和 CSC 则提供了更好的内存访问局部性。CSR 是表示稀疏矩阵最广泛使用的数据结构之一。如图 2.2 a)所示，CSR 结构由 row_ptr、col_idx 和 values 三个数组组成。row_ptr[i]的值包含第 i 行第一个元素的索引。values[]

保存非零元素的实际值，col_idx[]保存相应的列索引。如图 2.2 b)所示，在 col_idx[]和 values[]中连续放置稀疏每行中的非零值，非零元素在内存中的存储位置相对连续，可以更好地利用缓存，从而提升邻居节点数据的内存访问效率。

	0	1	2	3	4	5	6	7
0	a				b		c	
1		d	e		f			g
2		h			i		j	k
3		l		m				
4						n		
5		o				p		

a) 稀疏矩阵



b) CSR

图 2.2 稀疏矩阵的 CSR 表示

2.2.2 内存连续性优化设计

GCNs 在 *Aggregation* 阶段汇聚邻居节点特征，节点特征通常采用二维矩阵的方式存储。然而这种二维矩阵的存储方式存在诸多弊端：第一，二维矩阵的每一行基本上都存储为一个单独的动态数组（矢量），外层向量保存指向这些单独行向量的指针（或引用），这意味着行的内存可能不是连续的，从而可能导致潜在的低效内存使用和缓存定位。第二，在二维嵌套结构中，外层向量中的每个元素都需要额外的内存来存储指向每一行向量的指针/引用。与一维向量相比，这会导致更高的内存开销。第三，访问二维向量中的元素需要两级索引。例如，要访问元素 matrix[i][j]，第一个索引 i 访问行向量，第二个索引 j 访问该行中的特定元素。这种间接访问方式在一定程度上降低了数据访问速度。

针对以上问题，我们使用一维数组来模拟存储二维矩阵，将二维矩阵的每一行依次拼接在一起，形成一个一维数组。矩阵中的第一个元素对应一维数组的第一个元素，矩阵中的第二个元素对应一维数组的第二个元素，依此类推。在内存布局方面，由于一维数组在内存中的布局是连续的，这意味着所有元素都存储在相邻的内存位置。这样可以获得更好的缓存局部性和更高效的内存访问，从而提

高性能。在内存开销方面，每个元素只消耗存储整数值所需的内存。因此，与二维嵌套向量表示法相比，它的内存开销通常较低。在数据访问方面，访问一维数组中的元素只需要一个索引。例如，要访问原始矩阵中的元素 matrix[i][j]，我们可以利用 matrix[i*N+j]进行访问，其中 N 为原始矩阵的行数。这使得数据访问更直接、更高效。

2.2.3 节点分散处理方法

传统 GCNs 在 *Aggregation* 阶段采用逐个邻居节点单独聚合的方式来实现邻居节点特征的聚合。然而，这种逐个聚合的方法未能充分利用单指令多数据（Single Instruction Multiple Data, SIMD）内核的并行处理能力。针对这一问题，我们引入了一种节点分散的处理模式优化 *Aggregation* 阶段的计算。由于 SIMD 内核有两种并行处理模式：以节点为中心的处理模式和以边为中心的处理模式。在以节点为中心的处理模式中，每个节点的工作负载被分配给单个 SIMD 核心。该模式可以在并发条件下产生节点的聚合特征，即周期性地处理一组节点。然而，单个节点的处理延迟很长，快速处理的节点必须等待慢速处理的节点，从而导致了工作负载不平衡。此外，它还忽略了每个节点的邻居节点聚合过程中的并行性（即节点内并行性）。因此，我们采用以边为中心的处理模式，具体如图 2.3 所示，我们设计了一种节点分散处理方法，将每个节点的特征向量内的元素聚合分配给所有核，它充分利用了节点内并行性，减小了节点处理延迟。

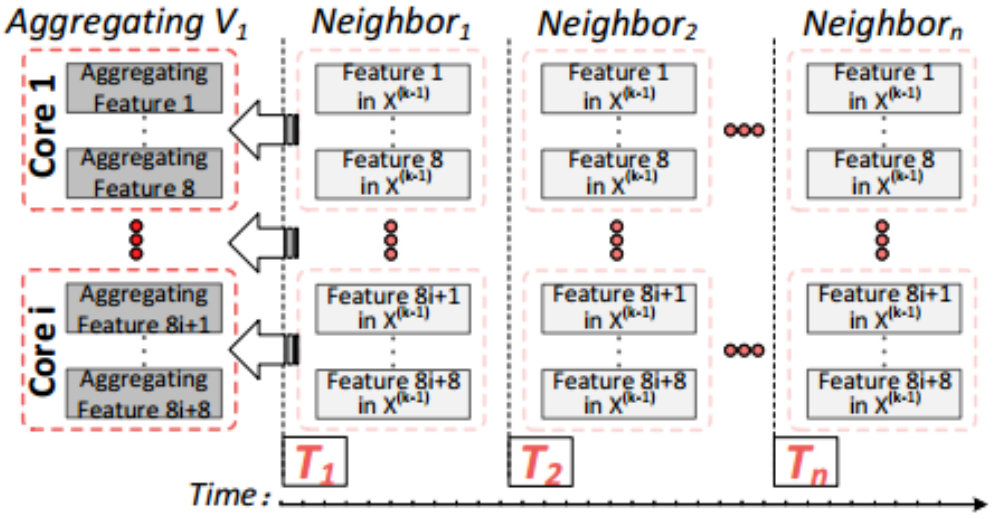


图 2.3 节点分散处理模式

2.3 *Combination* 阶段计算加速

在 *Combination* 阶段，它使用 MLP 将每个节点的特征向量转换为一个新的特征向量，MLP 通常用 MVM 表示。由于该过程涉及特征矩阵和权重矩阵之间的

矩阵乘法，对于一个具有 n 个节点的图，传统的 GCN 计算复杂度是 $O(n^3)$级别，这会带来较大的计算开销。此外，MVM 既是计算密集型也是访存密集型的任务，访存速度会严重制约矩阵计算的效率，朴素的矩阵乘法实现瓶颈在于访存指令的延迟，几乎每个浮点计算指令都要等访存指令。针对这一问题，我们充分利用了矩阵分块、SIMD 向量化、寄存器加速、内存重排、循环展开等方法提高处理并行性和数据重用性，从而最大限度地提升计算效率。

2.3.1 矩阵分块

矩阵分块是一种将参与乘法的大型矩阵拆分成更小的子矩阵，以便更高效地执行矩阵乘法的技术。矩阵分块技术通过将大矩阵分成足够小的子矩阵，并将子矩阵放入 Cache 中，使得更多的访存指令能从 Cache 直接获取数据，大大减小了平均访存延迟。

2.3.2 SIMD 向量化

SIMD（Single Instruction, Multiple Data）向量化技术是一种用于优化矩阵乘法等数值计算的重要方法。SIMD 允许在一条指令中同时对多个数据执行相同的操作，从而实现并行计算，提高计算效率。在现代 CPU 和 GPU 中，SIMD 指令集广泛支持，如 Intel 的 SSE（Streaming SIMD Extensions）、AVX（Advanced Vector Extensions），以及 ARM 的 NEON 等。SIMD 向量化，可以有效利用向量处理单元的能力，充分加速计算过程。

2.3.3 寄存器加速

在矩阵乘法优化中，数据寄存器加速思想是一种通过合理利用计算设备的寄存器（Register）来加速计算过程的优化技术。寄存器是位于 CPU 内部的高速存储区域，能够快速访问数据和执行指令。通过减少对内存的频繁访问，将数据存储在寄存器中，可以显著提高计算性能。

2.3.4 内存重排

内存重排（Memory Reordering）思想是根据每次循环的特点，调整数据在内存中的存储方式，将所有将会访问的元素排列到一起，减少跨区域访存次数，以提高数据访问效率和缓存的利用率，从而加速矩阵乘法的计算过程。内存重排通常涉及到数据在行、列和块的存储顺序上的调整，以适应计算设备的访问模式和缓存特性。

2.3.5 循环展开

循环展开技术旨在减少循环控制开销并充分利用硬件的并行性，从而提高计算性能。它通过将循环体中的多个迭代展开为多个独立的迭代，减少循环次数，从而减少循环控制的开销，提高计算效率。在矩阵乘法中，循环展开可以用于优化内部循环，即计算两个矩阵相乘的过程。

2.3.6 汇编指令集优化

汇编指令集是一种低级别的编程语言，能够直接操作计算机的硬件资源，包括寄存器、内存和算术逻辑单元。通过优化汇编指令集，可以更好地控制计算过程和资源使用，从而达到优化计算性能的目的。

2.4 固有数据流融合

对于由两个卷积层构成的 GCN，激活函数分别使用 ReLU 和 LogSoftmax。该 GCN 的计算过程可以表示为$\hat{Y} = f(X, A) = LogSoftmax\left(\hat{A}ReLU\left(\hat{A}XW^0\right)W^1\right)$。为了重用 GCN 计算过程中的固有数据流，我们首先在图数据预处理阶段，将图数据存储为 CSR 格式的同时，直接对邻接矩阵进行归一化处理，减少了邻接矩阵数据的重复读写。其次，我们考虑分别将 ReLU 和 LogSoftmax 运算与 AX 函数完全融合，进一步减少更新后的特征矩阵数据重复读写次数。最后，我们还对两个卷积层计算的中间矩阵进行了重分配，避免使用多个中间矩阵，进一步减少了内存开销。

3 算法优化

GCN 是一种用于图数据的深度学习模型，旨在对节点的特征进行学习和表示。GCN 推理过程中的 *Aggregation* 和 *Combination* 阶段都占用了大量的执行时间，其中 *Aggregation* 阶段表现出动态且不规则的执行特征，它受到内存的限制，而 *Combination* 阶段的执行模式是静态和规则的，它的主要瓶颈在于计算。本节将介绍我们对于这两个阶段的算法优化过程。我们采用优先执行 *Combination* 阶段的方式，通过减少特征长度，实现 *Aggregation* 阶段的计算加速。

3.1 *Aggregation* 阶段算法优化

在 *Aggregation* 阶段，我们首先利用 CSR 格式对原始稀疏图数据进行压缩存储，使得邻居节点信息在内存中连续存储，并且通过使用一维数组来模拟存储二维矩阵的方式进一步缓解了 *Aggregation* 阶段的不规则性。此外，我们还设计了一种节点分散处理方法，将每个节点的特征向量内的元素聚合分配给所有核，它充分利用了节点内并行性，减小了节点处理延迟。

（1）数据存储格式优化

Baseline 中采用邻接表存储图数据。邻接表是一种比较直观的图数据表示方法。对于每个顶点，使用链表或动态数组来存储与其相邻的顶点。Baseline 中的 edge_index 是一个 vector<vector<int>>类型，用于表示每个顶点的相邻顶点集合，而 degree 则记录每个顶点的度数（即与该顶点相连的边的数量）。由于构建邻接表的过程中，需要遍历图中的每条边，时间复杂度为$O(E)$。建立邻接表时，需要

存储每个顶点的相邻顶点列表和 degree 数组，空间复杂度为$O(\mathrm{V}+E)$。V 为顶点的数量，E 为边的数量。

而我们采用 CSR 格式队原始稀疏图数据进行压缩存储。CSR 是一种基于行的稀疏矩阵存储格式，将图的邻接矩阵转换成三个数组：row_ptr、col_index 和 edge_val。其中，row_ptr 存储每一行的起始索引，col_index 存储每个非零元素所在的列索引，edge_val 存储每个非零元素的值。建立 CSR 存储格式时，首先遍历每条边计算顶点度数，然后再次遍历每条边，循环填充 col_index 和 edge_val 数组，总共需要遍历每条边两次，时间复杂度为$O(E)$。由于 CSR 中需要存储 row_ptr、col_index、edge_val 和 degree 数组，空间复杂度为$O(\mathrm{V}+E)$。

邻接表在存储方面会比 CSR 格式占用更多的空间，因为需要额外的链表或动态数组来表示每个顶点的邻居。而 CSR 格式通过压缩行的方式，能够在稀疏图中节省大量存储空间。并且 CSR 存储格式使得非零元素在内存中的存储位置相对连续，可以更好地利用缓存，从而提升邻居节点数据的内存访问效率。

（2）内存连续性优化

Baseline 中使用了很多指针数组来分配二维矩阵，二维数组通常使用多个连续的一维数组来表示，导致每行可能被分配在不同的内存块中，可能存在内存碎片。并且二维数组的查找涉及行列嵌套关系，访问二维数组中的元素需要两级索引，这种间接索引方式将降低数据访问速度。

我们使用一维数组来模拟存储二维矩阵，将二维矩阵的每一行依次拼接在一起，形成一个一维数组。由于一维数组是线性排列的，元素在内存中连续存储，提高了内存利用率。此外，一维数组元素直接通过计算索引来访问，避免了额外的指针跳转操作，从而提高了访问效率。

（3）节点分散处理方法

Baseline 中执行邻居节点聚合时，遍历每个节点的邻居节点，然后对每个维度进行线性组合。这种实现方式可能导致计算复杂度较高，特别是当节点的邻居数较多、维度较大时，计算时间会随之增加。而且，由于嵌套循环，可能导致较多的循环迭代次数，进而影响性能。

我们引入了一种节点分散处理方法，它充分利用了每个节点的邻居节点聚合过程中的并行性。我们使用 OpenMP 的"#pragma omp simd"指令来进行向量化优化，从而充分利用硬件向量化指令，在计算每个邻居节点的变换结果时实现并行化，提高计算效率。特别是在较大的数据集和较大的维度情况下，这种优化可以明显加速计算过程。

在给定的由 1024 个节点和 4096 条边的图数据集上，为消除随机误差，我们在 Intel Xeon CPU 上将程序代码运行了 2000 次并取平均值，*Aggregation* 阶段加

速效果如图 3.1 所示。结果显示，我们的提出的方案能够将 *Aggregation* 阶段的执行时间由基础的 4.43ms 减少至 0.40ms，显著提升了 *Aggregation* 阶段的计算效率。

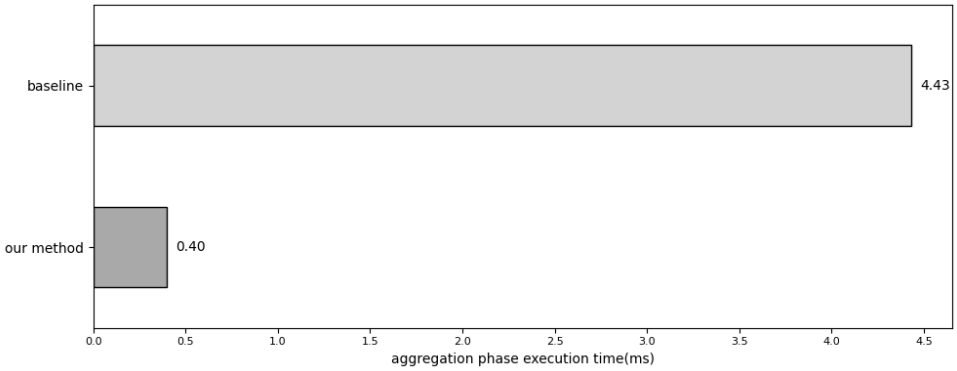


图 3.1 *Aggregation* 阶段加速效果图

3.2 *Combination* 阶段算法优化

在 *Combination* 阶段，为了提升数据重用度和计算效率，我们结合了常用的矩阵乘法优化思想，并根据硬件条件控制并发度，从而实现对 *Combination* 阶段的加速。

Baseline 中使用朴素矩阵乘法实现特征向量的转换，理论上要进行 $O(N^3)$量级的乘加计算，内存数据量只有 $O(N^2)$规模，内存访问次数却是 $O(N^3)$量级。MVM 既是计算密集型也是访存密集型的任务。如果访存速度和计算速度几乎相当的话，执行访存指令与浮点计算指令所耗时间应当相近。然而，我们了解到内存的访问带宽远远小于 CPU 进行浮点计算的吞吐量，且访存延迟远高于 CPU 进行浮点指令的延迟。虽然 CPU 的缓存相对较快，但其大小有限，无法容纳所有的矩阵数据。因此，朴素实现的瓶颈主要出现在访存指令的延迟上，导致每个浮点计算指令都必须等待访存指令的执行完成。

我们使用 OpenBLAS 库中的 cblas_sgemm()函数并行执行 MVM，它使用使用高度优化的底层实现来提升单精度浮点数矩阵乘法性能。底层优化通常包括使用高效的汇编代码，利用向量化指令（如 SIMD 指令）以及多线程并行处理，从而最大程度地利用计算机硬件的优势。此外，cblas_sgemm 函数会尽量让数据的访问模式更加连续和局部，以减少缓存未命中的情况，从而加快计算速度。同时，对于大规模矩阵乘法，cblas_sgemm 函数通常会使用矩阵分块，将大的矩阵划分为更小的块，这样可以使得每个小块都能够完全加载到缓存中，并充分利用局部性原理。

在给定的由 1024 个节点和 4096 条边的图数据集上，*Combination* 阶段加速

效果如图 3.2 所示，结果显示，我们的提出的方案能够将 *Combination* 阶段的执行时间由基础的 7.14ms 减少至 0.55ms，实现了 *Combination* 阶段的有效计算加速。

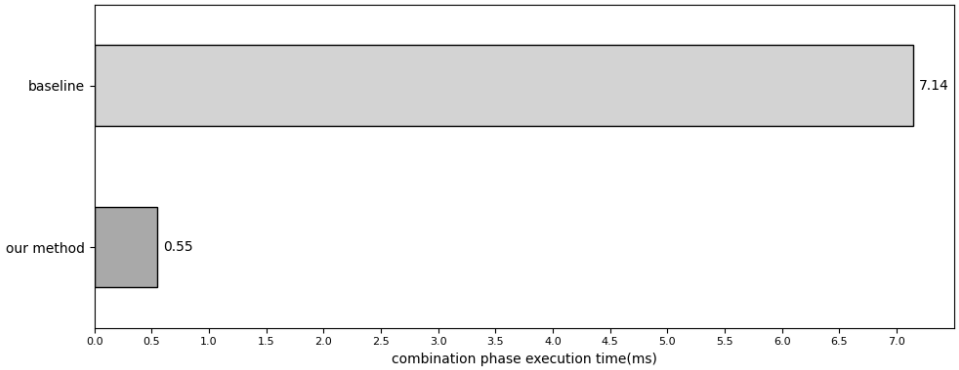


图 3.2 *Combination* 阶段加速效果图

3.3 固有数据流融合优化

除此之外，我们还对 GCNs 执行过程中的固有数据流进行了融合。首先，在在图数据预处理阶段，将图数据存储为 CSR 格式的同时，直接对邻接矩阵进行归一化处理，并将归一化后的值存入 values 数组的对应位置，减少了邻接矩阵数据的重复读写。其次，为了提高计算效率，我们在图卷积操作后立即应用激活，分别将 ReLU 和 LogSoftmax 运算与 AX 函数完全融合，进一步减少更新后的特征矩阵数据重复读写次数。最后，由于整个 GCNs 包含两个卷积层，每个卷积层的特征聚合及更新过程中会产生许多中间矩阵，我们对这些中间矩阵进行了重分配，避免使用多个中间矩阵，从而减少了内存开销。GCNs 的整体优化效果将在实验结果部分进行详细叙述。

4 详细算法设计与实现

我们针对 GCNs 中主导了执行时间的两个关键阶段：*Aggregation* 和 *Combination* 阶段分别进行了内存访问和计算上的优化。通过利用各种并行性，并且有效地重用高可重用性数据，我们的方法显著提高了数据局部性并减少了不必要的计算，能够达到与传统算法相当的测试精度。本节将详细描述我们的算法设计与实现。

4.1 *Aggregation* 阶段内存访问和计算优化的设计与实现

Aggregation 阶段需要访问和处理大量的稀疏数据，针对 *Aggregation* 阶段存在的图计算和内存访问模式的不规则性问题，我们从图数据存储、内存连续性优

化、以及节点分散处理三个方面实现了 *Aggregation* 阶段的内存访问和计算优化。

4.1.1 图数据存储格式 CSR 的实现

GCNs 处理的图数据大部分都是稀疏的，如何对图数据进行高效地存储和访问是提升节点特征聚合效率的重要手段。由 2.2.1 节可知，CSR 格式是一种常用的稀疏矩阵压缩表示方法，它使用以下三个数组来存储稀疏矩阵的非零元素：

1) row_ptr: 这是一个长度为 v_num + 1 的数组，用于存储每个节点在 col_index 和 edge_val 数组中的起始位置。数组中的第 i 个元素表示节点 i 在 col_index 和 edge_val 中的起始索引，而最后一个元素表示最后一个节点的结束索引。row_ptr[0]=0，row_ptr 中的第 i 个元素即为第 i-1 个元素的值加上第 i-1 个元素的度数。
2) col_index: 这是一个长度为 e_num 的数组，用于存储稀疏图中边的邻居节点的索引。col_index[j]表示第 j 条边的目标节点索引。
3) edge_val: 这是一个长度为 e_num 的数组，用于存储稀疏图中边的权重值。edge_val[j]表示第 j 条边的权重值。

CSR 结构使用 row_ptr、col_index 以及 edge_val 三个数组来对图数据进行压缩存储，非零元素和邻居节点信息在内存中的存储位置相对连续，可以更好地利用缓存，从而减少随机内存访问开销，提高内存访问效率。

4.1.2 内存连续性优化的设计与实现

GCNs 的邻居节点特征汇聚过程中涉及大量矩阵，而这些矩阵通常以二维矩阵的形式存储。内存连续性优化旨在改善程序在内存中存储和访问数据的方式，以提高程序的数据访问性能。将指针数组表示的二维矩阵转换为一维数组是一种常见的内存连续性优化方法。在二维矩阵中，每一行都是一个独立的指针，指向存放该行数据的内存区域。这种布局方式导致数据在内存中的存储位置可能不连续，此外，二维矩阵在进行数据访问时，需要通过指针访问每一行的数据，涉及间接寻址，带来了较大的内存访问开销。而在一维数组表示中，所有元素都连续存储在内存中，无需通过指针进行间接访问。这种转换可以提高缓存命中率和内存访问效率，从而提高程序性能。

我们利用一维数组模拟存储二维矩阵的思想，对 GCNs 推理过程中涉及的大量二维矩阵均采用一维数组进行表示。对于二维矩阵 X[rows][cols]，我们通过一维数组中索引 index=i*cols+j 访问二维矩阵中的元素 X[i][j]，这进一步提高了内存连续性和缓存资源利用率。

4.1.3 节点分散处理方法的设计与实现

针对传统 *Aggregation* 阶段的计算忽略了每个节点的邻居节点聚合过程中的并行性、难以充分发挥 SIMD 内核并行处理能力的问题，我们设计了一种节点分

散处理方法，它充分利用了 SIMD 并行处理模式和节点内并行性。我们使用 256 位的 AVX(Advanced Vector Extensions) SIMD 指令，可以一次性处理 8 个单精度浮点数（float），利用 SIMD 内核中以边为中心的并行处理模式，将每个节点的特征向量内的元素聚合分配给所有核，这种处理方法充分改善了以节点为中心的处理模式中的负载不均衡现象，同时利用了邻居节点聚合过程中的节点内并行性，减小了节点处理延迟。算法 1 展示了节点分散处理方法的伪代码。

算法 1：节点分散处理算法

输入：

dim：输入特征向量维度

in_X：输入特征向量

输出：

out_X：输出特征向量

```
for i in range(v_num):
start = row_ptr[i]
end = row_ptr[i + 1]
sum = array of size dim, initialized to zeros
for j in range(start, end):     # 访问节点 i 的邻居节点
    nbr = col_index[j]
    val = edge_val[j]
    # SIMD 向量化，向量化宽度为 8
    val_vec = broadcast(val)
     for k in range(0, dim, 8):
        in_X_vec = load 8 elements from 'in_X' at index nbr*dim+k
        sum_vec = load 8 elements from 'sum' at index k
        # 执行向量化乘法和累加
        sum_vec = add(sum_vec, multiply(val_vec, in_X_vec))
        store 8 elements from 'sum_vec' into 'sum' at index k
```

为了充分利用 256 位的 AVX SIMD 指令，我们需要在编译选项中指定-march=native，该选项会根据处理器自动选择最佳的指令集，并且在支持 AVX 的处理器上将启用 AVX 指令集。同时，使用-O3 优化编译选项，以便在生成可执行文件时进行高级别的优化，从而提高程序的执行速度和性能。

4.2 *Combination* 阶段计算加速的设计与实现

Combination 阶段使用 MLP 将每个节点的特征向量转换为一个新的特征向量，针对 *Combination* 阶段计算复杂度高，无法重用高可重用性数据的问题，我们使用高性能矩阵乘法库 OpenBLAS 中的单精度浮点数矩阵乘法函数 cblas_sgemm()来加速矩阵乘法。OpenBLAS 是一个开源的、高性能的基础线性代数子程序库，针对各种 CPU 架构进行了优化。cblas_sgemm()的加速思想涉及了分块、多级优化、线程并行、内存对齐、寄存器优化以及 Cache 优化等手段，充分利用现代 CPU 的计算能力和硬件特性，从而实现了高效、快速的矩阵乘法运算。cblas_sgemm()函数原型如下：

```
void cblas_sgemm(const enum CBLAS_ORDER Order,
                 const enum CBLAS_TRANSPOSE TransA,
                 const enum CBLAS_TRANSPOSE TransB,
                 const int M,
                 const int N,
                 const int K,
                 const float alpha,
                 const float *A,
                 const int lda,
                 const float *B,
                 const int ldb,
                 const float beta,
                 float *C,
                 const int ldc);
```

cblas_sgemm()用于计算 C = alpha * A * B + beta * C，计算结果存储在矩阵 C 中。函数各参数的含义如下：

1. `Order`：表示矩阵数据存储的顺序。它是一个枚举类型（enum CBLAS_ORDER），可选值有两种：
    - `CblasRowMajor`：表示矩阵按行主序存储，即行优先顺序。
    - `CblasColMajor`：表示矩阵按列主序存储，即列优先顺序。
2. `TransA`：表示矩阵 A 的转置选项。也是一个枚举类型（enum CBLAS_TRANSPOSE），可选值有三种：
    - `CblasNoTrans`：表示不对矩阵 A 进行转置操作。
    - `CblasTrans`：表示对矩阵 A 进行转置操作。
    - `CblasConjTrans`：表示对矩阵 A 进行共轭转置操作（仅在复数运

算中使用）。

3. TransB：表示矩阵 B 的转置选项，含义与 TransA 相同，也是一个枚举类型。

4. M：表示结果矩阵 C 的行数。

5. N：表示结果矩阵 C 的列数。

6. K：表示矩阵 A 和 B 的公共维度，即 A 的列数和 B 的行数。

7. alpha：表示乘法的比例因子，用于控制乘法的强度。

8. A：表示输入矩阵 A 的指针，是一个一维数组。

9. lda：表示矩阵 A 的 Leading Dimension，即矩阵 A 的列数。

10. B：表示输入矩阵 B 的指针，也是一个一维数组。

11. ldb：表示矩阵 B 的 Leading Dimension，即矩阵 B 的列数。

12. beta：表示结果矩阵 C 的比例因子，用于控制结果矩阵的叠加强度。

13. C：表示输出结果矩阵 C 的指针，也是一个一维数组。

14. ldc：表示矩阵 C 的 Leading Dimension，即矩阵 C 的列数。

我们在 *Combination* 阶段利用 cblas_sgemm()函数加速矩阵乘法运算时，各参数设置如下：

```
cblas_sgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, v_num, out_dim, in_dim, 1.0, in_X, in_dim, W, out_dim, 0.0, out_X, out_dim);
```

4.3 固有数据流融合的设计与实现

首先，在传统 GCN 计算过程中，通常先加载数据集，将输入的图数据表示为邻接矩阵或者邻接表的形式（我们采用标准 CSR 的格式存储）。然后遍历邻接矩阵，对其进行归一化处理。在这个过程中，存在对邻接矩阵中数据值的重复读写，我们可以对该数据流进行融合，即在构建 CSR 存储格式的过程中，直接将归一化后的值存入 edge_val 数组中。

其次，在卷积计算过程中，GCN 会聚合其邻居节点的表示信息，并更新自身的特征向量，再通过激活函数 ReLU 和 LogSoftmax 运算传递到下一卷积层中。在这个过程中，存在对节点特征矩阵的重复读写，我们可以对该数据流进行融合，即在更新节点自身特征向量的同时进行激活，直接将激活后的结果写入结果特征矩阵的对应位置。

最后，由于 GCN 包含两个卷积层，如果给每一个卷积层都分配一个矩阵存储中间计算结果，将会占用一定的存储空间。

对于第一层卷积层的XW运算，输入特征矩阵X_0和第一层权重矩阵W_1，计算结果存放在第一层中间特征矩阵X_1_inter中。

$$X_0 * W_1 = X_1\_inter$$

$$[v\_num \times F_0] * [F_0 \times F_1] = [v\_num \times F_1]$$

然后进行第一层卷积层的AX运算，输入稀疏邻接矩阵A和中间特征矩阵X_1_inter，计算结果存放在X_1矩阵中。

$$A * X_1\_inter = X_1$$

$$[v\_num \times v\_num] * [v\_num \times F_1] = [v\_num \times F_1]$$

对于第二层卷积层的XW运算，输入特征矩阵X_1和第二层权重矩阵矩阵W_2，计算结果存放在第二层中间特征矩阵X_2_inter中。

$$X_1 * W_2 = X_2\_inter$$

$$[v\_num \times F_1] * [F_1 \times F_2] = [v\_num \times F_2]$$

接下来进行第二层卷积层的AX运算，输入稀疏邻接矩阵A和中间特征矩阵X_2_inter，计算结果存放在X_2矩阵中。

$$A * X_2\_inter = X_2$$

$$[v\_num \times v\_num] * [v\_num \times F_2] = [v\_num \times F_2]$$

由于 GCN 是一个特征降维的过程，我们可以对两个卷积层计算的中间矩阵进行重分配，即将X_1_inter和X_2_inter合并，避免使用多个中间矩阵，从而减少了内存开销。

5 实验结果与分析

5.1 比赛给定数据集实验分析

我们在 Intel Xeon CPU 上使用比赛给定的数据集对我们的提出的优化算法和比赛所给 example 代码的执行效率进行了比较。程序输出结果如表 5.1 所示，为消除随机误差，我们在相同平台上对两段程序代码运行了 2000 次，并取平均值作为相应的执行时间。

表 5.1 给定数据集上的程序运行结果图

	Baseline	**Our methods**
最大顶点特征矩阵行	-16.68968964	-16.68968964
执行时间(ms)	11.250083210000012	0.35961826050000123

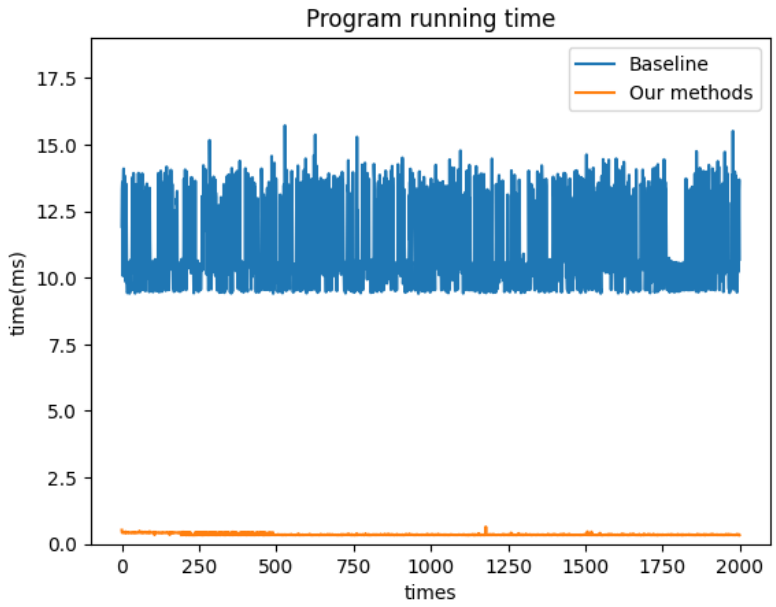


图 5.1 2000 次 GCN 推理执行时间对比图

结果显示，我们提出的方案在给定数据集上能够实现更高效的 GCN 推理速度。优化前平均执行时间为 11.25ms，优化后平均执行时间为 0.36ms，在 Baseline 的基础上将推理性能提升了约 31.28 倍，同时还能够达到与传统算法相当的测试精度。这是因为我们提出的 GCN 推理加速方法，它缓解了 *Aggregation* 阶段的不规则性，并同时利用了 *Combination* 阶段的规则性。

1）首先，我们的方案通过在 *Aggregation* 阶段利用缓存友好型的 CSR 格式存储大量稀疏图数据，使得邻居节点信息在内存中连续存放，减少了不必要的稀疏性访问。同时，使用一维数组来模拟存储二维矩阵的内存优化模式进一步提升了内存访问效率。此外，我们设计的节点分散处理方法，充分利用 SIMD 内核并行处理模式，将每个节点的特征向量内的元素聚合分配给所有核，从而充分利用了节点内并行性，提升了 *Aggregation* 阶段的计算效率。

2）其次，在 *Combination* 阶段，我们充分利用规则性，使用 OpenBLAS 库提供的 cblas_sgemm()函数并行执行 MVM 并重用共享参数。通过最大程度地利用计算机硬件的优势，实现了节点特征变换过程的有效加速。

3）最后，我们还对 GCNs 计算过程中的固有数据流进行操作融合，进一步加速了前向传播的计算过程。

图 5.1 展示了我们的优化算法与 Baseline 执行 2000 次 GCN 推理过程中的时间花销。为了更细致地展示我们的优化算法在执行 2000 次 GCN 推理过程中的具体表现，我们将其绘制在图 5.2 中。程序在前 500 次的执行时间仍然较长，因为此时还没有进行缓存预热，大约 500 次之后，执行时间稳定在 0.347ms。程序代码执行 2000 次的 GCN 平均推理时间约为 0.36ms。

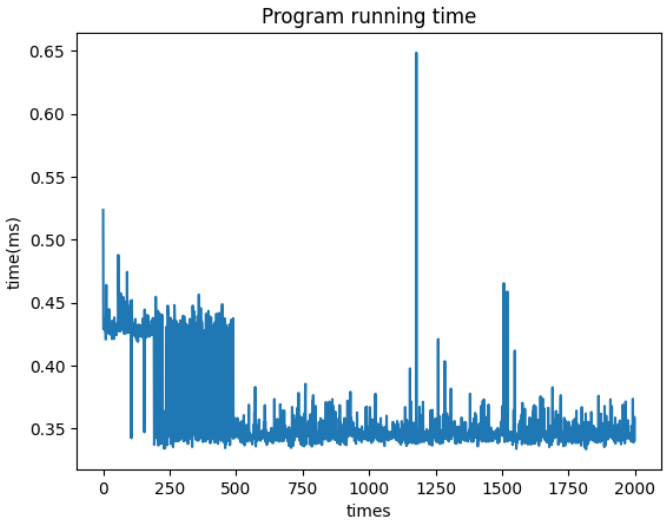


图 5.2 我们的方案执行 2000 次 GCN 推理过程图

5.2 RMAT 生成数据集实验分析

为了评估我们所提出的 GCN 推理加速方法的泛化性，我们还使用 RMAT（https://github.com/farkhor/PaRMAT）随机生成了不同大小的图文件及节点特征矩阵，并在这些数据集上对比了我们提出的优化算法和 Baseline 的执行效率，具体结果如图 5.2 所示，自行生成的数据集规模说明如表 5.2 所示。

表 5.2 随机生成的数据集规模

	图数据集大小	特征矩阵大小
dataset1	10240×10240	10240×64
dataset2	20480×20480	20480×64
dataset3	1M×1M	1M×64

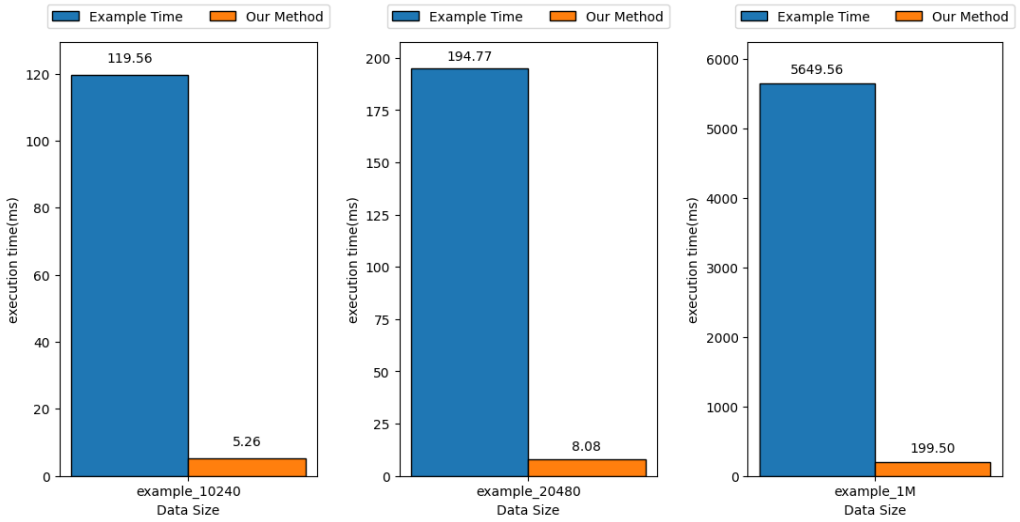


图 5.2 不同数据集规模下的 GCN 加速效果

当图数据集大小为10240×10240时，我们的方案将执行时间由原来的平均119.56ms 降低至平均 5.26ms，实现了约 22.73 倍的加速；当图数据集大小为20480×20480时，我们的方案将执行时间由原来的平均 194.77ms 降低至平均8.08ms，实现了约 24.11 倍的加速；当图数据集大小为$1M \times 1M$时，我们的方案将执行时间由原来的平均 5649.56ms 降低至平均 199.50ms，实现了约 28.31 倍的加速。进一步验证了我们所提出算法的有效性。

6 程序代码模块说明

我们的程序代码使用了一些 C++标准库，如<vector>、<fstream>、<sstream>等，以及 OpenBLAS 和 AVX（Advanced Vector Extensions）指令集的库<cblas.h>。

程序代码各函数说明如下：

1.readGraph()

- 输入：

  fname：图文件名

- 输出：

  无

- 功能描述：

  从输入文件名 fname 指定的文件中读取图数据，将图的顶点数和边数存储在全局变量 v_num 和 e_num 中，并将图的边信息存储在全局向量 raw_graph 中。

2.readFloat()

- 输入：

  fname：浮点数文件名

  dst：目标内存指针

  num：浮点数数量

- 输出：

  无

- 功能描述：

  从文件中读取浮点数数据，并将数据存储在指针 dst 所指向的内存空间中。

3.initFloat()

- 输入：

  dst：目标内存指针

num：浮点数数量

- 输出：

  无

- 功能描述：

  初始化一个浮点数数组，并将所有元素置为 0。

4.raw_graph_to_CSR()

- 输入：

  无

- 输出：

  无

- 功能描述：

  将原始的图数据（raw_graph）转换为压缩稀疏行 CSR 的格式。它构建了三个全局向量 row_ptr、col_index 和 edge_val，其中 row_ptr 存储了每个顶点的边在 col_index 和 edge_val 中的起始索引，col_index 存储了每个边的目标顶点，而 edge_val 存储了每个边的权重值（边的标准化值）。

5.XW()

- 输入：

  in_dim：in_X 的列数/W 的行数

  out_dim：W 的列数/out_X 的列数

  in_X：输入矩阵

  out_X：输出矩阵

  W：输入矩阵

- 输出：

  无

- 功能描述：

  节点特征变换，执行矩阵乘法运算 out_X = in_X * W，这里我们使用 OpenBLAS 库中的函数 cblas_sgemm 来执行矩阵乘法。

  OpenBLAS 库中的函数 cblas_sgemm 来执行矩阵乘法。

6.AX_relu()

- 输入：

  dim：in_X 的列数

  in_X：输入矩阵

  out_X：输出矩阵

- 输出：

  无

- 功能描述：

  执行稀疏矩阵乘法、ReLU 激活函数。对于图中的每个节点，聚合其邻居节点的特征，然后执行 ReLU 激活函数。

7.AX_LogSoftmax()

- 输入：

  dim：in_X 的列数

  in_X：输入矩阵

  out_X：输出矩阵

- 输出：

  无

- 功能描述：

  执行稀疏矩阵乘法和 LogSoftmax 激活函数。对于图中的每个节点，聚合其邻居节点的特征，然后执行 LogSoftmax 激活函数。

8.MaxRowSum()

- 输入：

  X：输入特征矩阵

  dim：输入特征矩阵维度

- 输出：

  max_sum：最大的顶点特征矩阵行

- 功能描述：

  计算矩阵每行元素之和的最大值。

9.freeFloats()

- 输入：

  无

- 输出：

  无

- 功能描述：

  释放分配的浮点数数组的内存。释放 X0、W1、W2、X1、X2 和 X_inter 指向的内存空间。

10.cleanup()

- 输入：

  无

- 输出：

  无

- 功能描述：

  在计算结束后进行清理操作。清空 degree、row_ptr、col_index 和 edge_val 的内容，并调用 freeFloats()函数释放浮点数数组的内存。

11.main()

- 输入：

  argc：命令行参数的数量

  argv：指向指针的指针参数，代表命令行参数的内容。每个指针指向一个字符串，表示一个命令行参数的具体内容。接收的 7 个具体参数为：输入顶点特征长度、第一层顶点特征长度、第二层顶点特征长度、图结构文件名、输入顶点特征矩阵文件名、第一层权重矩阵文件名、第二层权重矩阵文件名。

- 输出：

  max_sum：最大的顶点特征矩阵行

  l_timeMs：执行时间

- 功能描述：

  主函数。首先读取图数据，并加载输入顶点特征长度、第一层顶点特征长度、第二层顶点特征长度 F0、F1、F2，以及输入特征矩阵 X0 和权重矩阵 W1、W2。然后调用一系列函数来执行 GCN 的计算。最后输出结果和程序的运行时间。

除此之外，还有 raw_graph_to_CSR_single()、AX_relu_single()、AX_LogSoftmax_single()函数，这三个函数的实现与功能同 raw_graph_to_CSR()、AX_relu()、AX_LogSoftmax()函数，不同点是未使用 openmp 的并行化。

并行计算在提高性能、加速计算速度和解决大规模问题方面具有显著优势。但是，在并行计算中，可能出现资源竞争和冲突，多个处理单元之间需要进行数据同步和通信这可能导致额外的开销和延迟。因此需要平衡并行计算所带来的性能提升和额外的开销。我们发现在小数据集（如比赛所给的 1024 size 数据集）上，不使用 openmp 并行化反而能取得更高的计算性能。然而在大型数据集上（如我们使用 RMAT 随机生成 1M size 数据集），使用 openmp 并行化能够带来更大的计算加速。因此，我们针对不同规模的数据集，设计了不同的卷积函数：当节点数大于等于 500K 的时候，选择启用多线程，即执行 raw_graph_to_CSR()、AX_relu()、AX_LogSoftmax()。当数据集规模小于 500K 时，卷积过程中执行 raw_graph_to_CSR_single()、AX_relu_single()、AX_LogSoftmax_single()。

7 详细程序代码编译说明

程序代码的编译指令为：

```
g++-9 -o ../makka_pakka.exe source_code.cpp -O3 -march=native -I /home/gy/include/ -L/home/gy/lib -lopenblas -fopenmp
```

各编译选项说明如下：

1. `g++-9`:指定使用版本为 9 的 g++编译器进行编译。
2. `-o ../makka_pakka.exe`: 指定输出文件的名称和路径，将输出的可执行文件将被放置在上一级目录（即当前目录的父目录）中，文件名为 makka_pakka.exe。
3. `source_code.cpp`: C++源代码文件。
4. `-O3`: 令编译器使用更高级别的优化。
5. `-march=native`: 编译时使用本地 CPU 的特定指令集进行优化，编译器将根据当前编译代码的机器的 CPU 架构来生成特定的指令，以获得更好的性能。
6. `-I /home/gy/include/`: 指定头文件的搜索路径。由于我们的程序代码中使用了 OpenBLAS 库，并且 OpenBLAS 库安装路径为`/home/gy`，实际运行过程中需将该路径改为 OpenBLAS 库的对应安装路径。
7. `-L/home/gy/lib`: 指定库文件的搜索路径。由于我们的程序代码中使用了 OpenBLAS 库，并且 OpenBLAS 库安装路径为`/home/gy`，实际运行过程中需将该路径改为 OpenBLAS 库的对应安装路径。
8. `-lopenblas`: 链接 OpenBLAS 库。
9. `-fopenmp`: 启用 OpenMP 多线程支持，将任务并行化以提高程序性能。

8 详细代码运行使用说明

程序代码运行环境为：

- CPU：Intel(R) Xeon(R) Gold 6330 CPU @ 2.00GHz
- 操作系统：Ubuntu 21.10
- g++或 gcc 编译器：9.4.0

代码运行具体步骤说明如下：

1.首先进入源代码所在目录

```
cd cgc_makka_pakka/makka_pakka
```

2.编译代码

```
make
```

3.返回上层目录

```
cd ..
```

4.执行可执行文件并传入参数

```
./makka_pakka.exe  64  16  8  graph/1024_example_graph.txt
embedding/1024.bin weight/W_64_16.bin weight/W_16_8.bin
```