图卷积神经网络加速器 HyGCN 的复现

陈家贤

摘要

受神经网络巨大成功的启发，人们提出了图卷积神经网络（Graph Convolutional Networks, GCNs）来分析图数据。GCNs 主要包括两个具有不同执行模式的阶段。聚合阶段，表现为图形处理，显示出动态和不规则的执行模式。组合阶段的行为更像神经网络，呈现出静态和规则的执行模式。GCN 的混合执行模式需要一种能够减轻不规则性和利用规则性的设计。此外，为了实现更高的性能和能源效率，设计需要利用聚合阶段的高顶点内并行性，组合阶段的高度可重复使用的顶点间数据，以及 GCN 的新特性所带来的融合逐个阶段执行的机会。然而，现有的架构未能满足这些需求。

在这项工作中，我们首先分析了 GCNs 的混合执行模式。在该特征分析的指导下，我们复现了一个 GCN 加速器 HyGCN，使用混合架构来有效地执行 GCNs。具体来说，首先，我们介绍了一个新的编程模型来利用我们的硬件设计的细粒度并行性。其次，我们介绍了一个带有两个高效处理引擎的硬件设计，以减轻聚合阶段的不规则性和利用组合阶段的规则性。此外，这些引擎可以利用各种并行性，有效地重用高度可重用的数据。第三，我们通过用于阶段间融合的引擎间管道和基于优先级的片外内存访问协调来优化整个系统，以提高片外带宽的利用率。与运行在英特尔至强 CPU 上的最先进的软件框架相比，HyGCN 实现了平均 275 倍的速度提升和 4112 倍的能耗降低。

关键词：图卷积神经网络；专用加速器；内存瓶颈

1 引言

受神经网络强大的学习能力的启发，图卷积神经网络（Graph Convolutional Networks, GCNs）被视为一种有效的深度学习模型用于表示和处理图数据[1-2]。GCNs 将图数据转换为低维空间，同时最大限度地保留结构和属性信息，然后构建神经网络进行后续训练和推理。最近，GCNs 吸引了工业界和学术界的大量努力，解决包括节点分类、链接预测、图聚类和推荐系统在内等问题。因此，GCN 逐渐成为数据中心，例如 Google、Facebook 和阿里巴巴等公司的新型工作负载。

GCNs 中两个主要执行阶段占据它的大部分执行时间，即聚合和组合过程。聚合阶段类似图处理，它严重依赖于随机和稀疏的图结构。每个顶点的处理需要聚合其所有源邻居的特征。然而，这些顶点邻居的数量和位置在不同顶点之间有很大差异。所以，每个顶点的聚合阶段的计算和内存访问模式是动态的和不规则的。而组合阶段更像神经网络，它使用多层感知器将每个顶点的特征向量转换为新的特征向量。由于神经网络层内每个神经元的连接模式相同，因此每个顶点的组合阶段的计算和内存访问模式是静态和规则的。聚合和组合阶段截然不同的计算和内存访问模式，对现有的计算机体系结构提出了严峻的挑战。为此，HyGCN[3]提出了一种基于混合计算引擎的 GCNs 加速器，用于提高 GCNs 的处理速度并降低能耗开销。

2 Motivation

聚合阶段在很大程度上依赖于图的结构，而图的结构本身是随机和稀疏的，这导致了大量的动态计算和不规则访问。聚合阶段的每个操作都需要从 DRAM 中获取比组合阶段多得多的数据，从而导

致了更高的 DRAM 访问能耗。此外，在聚合阶段，L2 和 L3 缓存的每千指令失误数（MPKI）非常高，这是由于每个顶点的邻居索引的高度随机性造成的。此外，间接和不规则的访问使聚合阶段的数据预取无效，因为如果不事先知道邻居的指数，就很难预测数据地址。这导致了大量无效的内存访问来预取数据。

组合阶段为每个顶点执行一个基于 MLP 的共享神经网络的矩阵向量乘法（Matrix Vector Multiplication，MVM），它具有静态和规则的计算和访存特征。组合阶段的每个操作只需要从 DRAM 中访问少量的数据。这是因为矩阵向量乘法是非常密集的计算，而且 MLP 的权重矩阵在顶点之间广泛共享。然而，在线程之间的共享数据复制和同步方面，观察到高达 36% 的执行时间。根据上述分析，GCN 中存在混合执行模式。合阶段执行的是动态和不规则的执行模式，受限于内存访问瓶颈，而组合阶段是静态和规则的，受限于计算能力瓶颈。

为特定领域设计一个专门的加速器是解决现有架构效率低下的一个有效而普遍的解决方案，因为它可以根据特定的工作负载定制内存层次和计算单元。对于 GCNs，我们可以针对两个不同阶段进行分别优化，来构建一个混合架构的加速器。对于聚合阶段，可以提前获得图数据的知识，并安排访问以缓解不规则性。此外，每个顶点的计算也可以被调度，以利用边缘平行性和顶点内的平行性。对于组合阶段，我们从目前的神经网络加速器中得到启发，在参数共享的情况下有效地执行矩阵并行运算。除了这两个阶段的单独优化外，串行的阶段间数据流可以以更细的粒度进行流水线处理。此外，所有的片外内存访问可以被控制，以提高整体的内存访问效率。把所有这些放在一起，有巨大的机会来设计一个高效的 GCN 加速器，并具有高性能。

3 相关工作

3.1 传统编程框架

很多用于图分析和神经网络的软件框架已经被提出来，以减少编程工作负担，同时在现代通用架构上满足用户的高性能需求。然而，所有这些框架只对单一模式的工作负载起作用。因此，最近提出了大量的混合模式 GCNs 的软件框架。例如，PyTorch Geometric[4]利用消息传递框架来增强其表达能力和硬件优化的操作（如散点和矩阵乘法），从而可以加速 GCN 的工作负载。不幸的是，聚合阶段和组合阶段之间关于计算和访问的不同执行模式在传统平台上产生了处理的低效率。GCNs 需要专门的架构设计。

3.2 专用加速器

随着图形分析和神经网络工作负载的出现，很多硬件架构设计被提出来以加速这些工作负载。例如，Graphicionado[5]是为图分析量身定做的；而 Tensor Processing Unit (TPU) 则专注于神经网络的加速。然而，GCN 的行为不仅像图处理（聚合过程），也像神经网络（组合过程），导致了内在的混合设计要求。因此，目前的专业架用不能有效地执行 GCNs，因为它们只是处理两方面中的一个。

4 本文方法

4.1 基本数据流

构建编程模型（Programming Model，PM）可以充分利用并行机制，并为程序员实现硬件透明的编程框架。对于聚合，有基于 Gather 和 Scatter 两种处理方法。由于基于 Scatter 的方法通常会产生大量的原子操作，并且在处理完所有顶点后需要进行同步，所以并行程度会下降。相反，基于 Gather 的方法可以很容易地控制程序行为并保持执行的并行性。因此，HyGCN 在设计中选择了基于 Gather 的处理方式。然而，这种处理模式会导致密集的内存访问和顶点计算。为了解决这个问题，HyGCN 采用了以边为中心的 PM 来利用边级的并行性。每个顶点都有许多传入的边（邻居），它们可以在一个逐个边的流水线中被聚集起来。通过这种方式，每个顶点的工作负载可以被划分为子工作负载，并分配给每个计算单元进行并行处理。对于组合来说，情况相对容易一些。由于每个顶点的计算都像 MLP 一样，我们直接优化 MVM 操作。

4.2 本文方法概述

基于上述的数据流，图 1描述了 HyGCN 的架构。HyGCN 采用了一个混合架构来加速 GCNs 的处理，它主要包括两个引擎（聚合引擎和组合引擎）和一个内存访问处理器。一个通信接口（Coordinator）被引入以连接这两个引擎，并缓解它们之间的资源使用冲突，同时建硬件流水线来提高设备利用率。聚合引擎的目的是实现不规则访问和计算的有效执行。为了利用边级的并行性，HyGCN 设计了一个任务调度器（eSched），将边处理的工作负载分配给 Single Instruction Multiple Data（SIMD）处理核。同时，HyGCN 设计了一个稀疏性消除器，以避免与聚合顶点不共享边的顶点的冗余内存访问。组合引擎的设计是为了使常规访问和计算的效率最大化。为了提高处理的并行性和数据的重用性，HyGCN 采用了著名的脉动阵列设计，并对其进行了修改以兼容 GCN。

4.3 聚合引擎

为了优化聚合的计算，HyGCN 引入了顶点分散的处理模式。为了优化内存访问，HyGCN 采用了静态图分割方法来加强数据的重用，并采用了动态稀疏度消除技术来减少不必要的数据访问。

(1) 执行模式。HyGCN 使用顶点分散模式处理聚合过程。如图 2所示，它将每个顶点的顶点特征向量内的元素聚合分配给所有 SIMD 核心。如果一个顶点不能占据所有核心，空闲的核心可以分配给其他顶点。因此，所有的核心总是繁忙的，没有工作负荷的不平衡。此外，由于利用了顶点内的并行

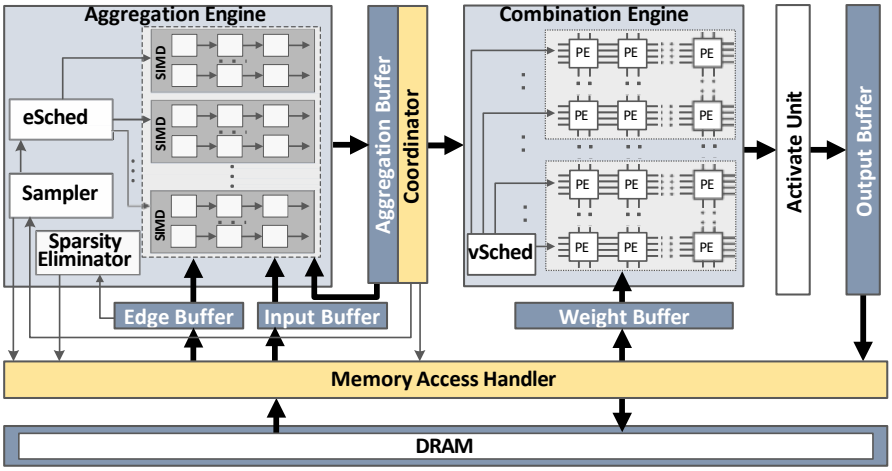

图 1: HyGCN 的架构示意图

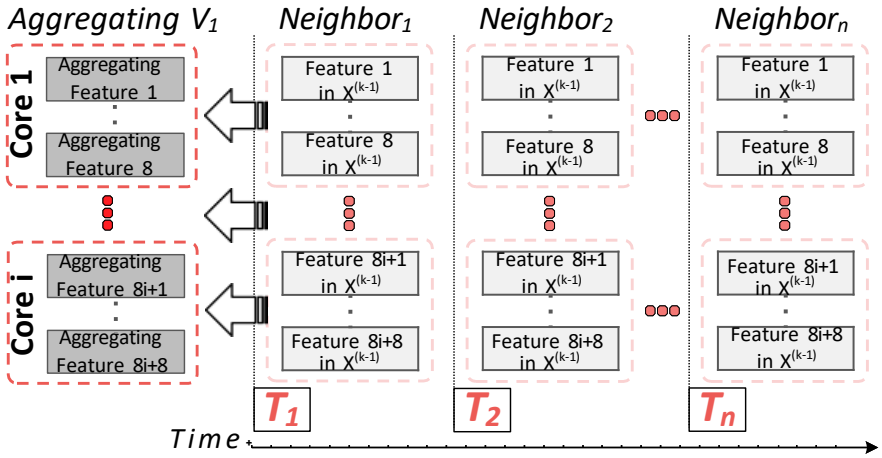

图 2: 基于顶点分散模式的聚合流程

性，单个顶点的延迟比一起处理多个顶点的延迟要小。此外，它还能在下面的组合引擎中立即处理每个顶点。

(2) 静态图分割。HyGCN 使用了顶点区间和边碎片的抽象来划分图数据，这也是下一小节中进行数据感知的稀疏消除的基础。我们不需要明确的预处理来生成间隔和碎片，因为我们直接采用压缩稀疏列（CSC）的数据格式作为输入。如图 3(a) 所示，16 个顶点被组织成几个区间（即从 I1 到 I4，每个区间有 4 个顶点），边缘被组织成 4×4 个碎片（即从 S(1, 1) 到 S(4, 1)，每个碎片最多有 16 条边）。区间和碎片是不相交的。

每个顶点的特征向量长度通常都很大，所以利用特征的局部性至关重要。我们将同一区间内的顶点分组（如 Ii），然后将其源邻居的聚合也按区间进行处理（即遍历 Ij）。基于这一流程，一个区间内所有顶点的特征访问被合并（见图 3(b)）。由此带来的好处是双倍的。首先，Ii 中的顶点在 Ij 中通常有重叠的邻居，因此，在进行特征聚合时，Ij 中加载的特征数据可以被重新使用。其次，当遍历所有的 Ij 时，Ii 的中间聚合结果被保留在缓冲区中，在进行特征更新时也可以重复使用。在实践中，分片的高度是由输入缓冲区的容量决定的，而分片的宽度是由聚合缓冲区的容量决定的。边缘缓冲区的大小同时影响到高度和宽度，因为它可以容纳每个分片的所有边。

(3) 动态稀疏性消除。通过数据重用优化，我们进一步尝试减少冗余访问，因为图连接是稀疏分布的。为了消除稀疏性，HyGCN 提出了一种基于窗口的滑动和收缩方法。其关键思想是，我们首先向下滑动窗口（与边碎片的大小相同），直到顶行出现一条边缘，然后通过向上移动底行来缩小窗口大小，直到遇到一条边。

窗口滑动。图 3(c) 说明了窗口的滑动过程。对于每个顶点的间隔，顶部的碎片窗口逐渐向下滑动。直到它的顶行出现一条边，它才会停止。然后，一个具有相同大小的新窗口被创建，其顶行紧跟其前

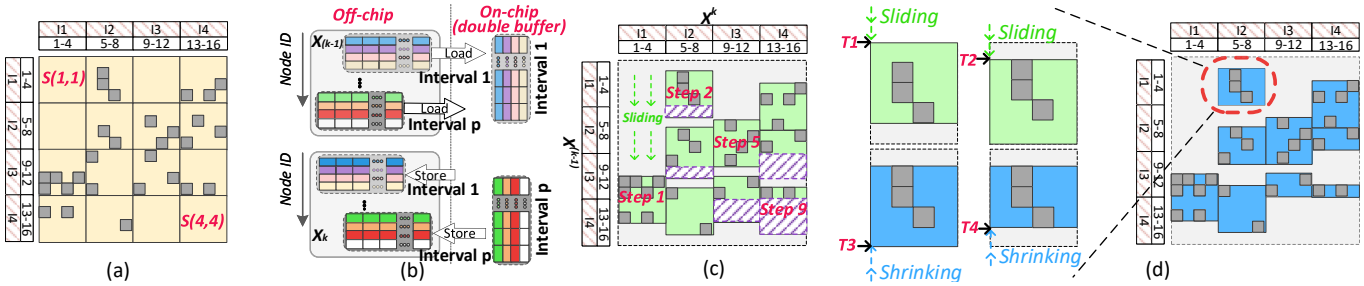

图 3: HyGCN 的静态图分割和动态稀疏消除技术

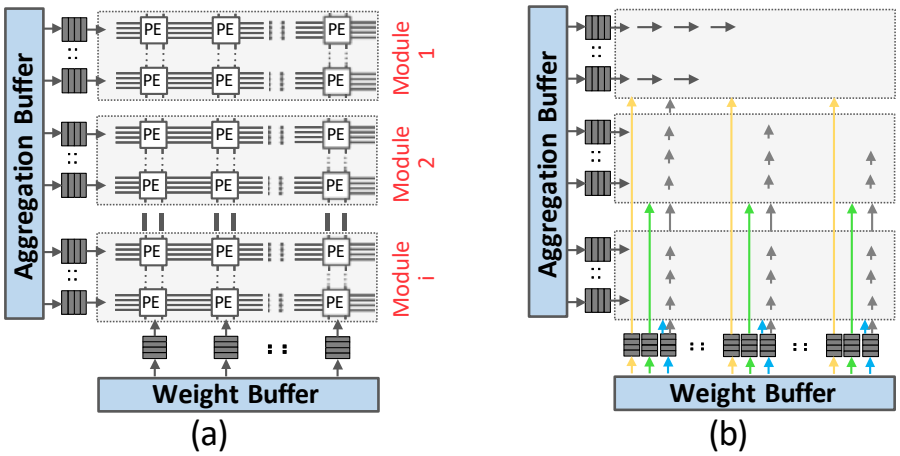

图 4: HyGCN 的组合引擎，脉动阵列模块

一个窗口的底行。每个窗口的停止标准都是一样的。通过这种方式，窗口不断出现，向下滑动，然后停止。所有窗口停止的位置都被记录为有效的碎片。

窗口收缩。尽管窗口的滑动可以捕捉到大多数有效边，但在底部（紫色虚线框内）仍然存在稀疏性。这是因为上述滑动方向是向下的。为了减少这部分的稀疏性，我们在此提出窗口收缩。具体来说，每个记录的窗口的底行向上移动，直到它遇到一个边，然后窗口收缩。图 3(d) 详细说明了一个窗口的滑动和收缩过程，并给出了最终记录的有效碎片。与之前的分区不同，由于窗口的收缩，最终碎片的大小通常是不同的。

4.4 组合引擎

每个顶点的组合操作就像一个神经网络，它的执行是有规律的，但计算量很大。我们的设计是基于著名的脉动阵列。为了让它适应聚合引擎的两种处理模式，我们整合了多个阵列，而不是单一的阵列，如图 4所示。一组脉动阵列被组装起来，形成一个脉动模块。我们允许这些脉动模块的多角化使用，包括独立工作模式和合作工作模式。

(1) 独立工作模式。在这种模式下，各脉动模块相互独立工作，每个模块处理一小群顶点的 MVM 操作。在这种情况下，每个模块的权重参数直接从权重缓冲器中获取，并在模块内重复使用。这种模式的优点是顶点延迟较低，因为一旦顶点的聚合特征准备好了，我们就可以立即处理这一小组顶点的组合操作，而不需要等待更多的顶点。在这种模式下，聚合的特征会快速但有顺序地产生。

(2) 合作工作模式。除了单独工作，这些脉动模块还可以进一步结合在一起，同时处理更多的顶点。与立即处理顶点不同的是，这种模式要求在执行其组合操作之前将一大群顶点的聚合特征集合在一起。其优点是，权重参数可以从权重缓冲区流向下游的脉动模块，然后逐渐流向上游的脉动模块，这些参数被所有脉动阵列大大重用。这有助于减少能源消耗。无论在组合引擎中选择哪种工作模式，在处理不同顶点时，权重可以在权重缓冲区中得到重复使用。然而，在传统的神经网络中，特别是 MLPs，如果没有批处理技术，权重就不能被共享。多角脉动阵列的设计也是 HyGCN 的架构所特有的，以适应不同的应用需求。

4.5 引擎间的优化

片外内存访问的协调。实际应用中，很难确定两个引擎之间的内存带宽比例分配，因为实际工作负载通常在聚合和组合之间变化。HyGCN 只使用一个片外存储器。因为内存系统的分离会增加配置的开销，造成带宽的浪费。

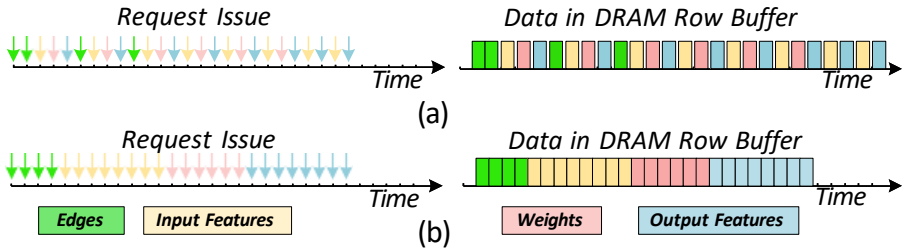

图 5: HyGCN 的引擎间内存优化

两个引擎在运行时都会访问这个存储器，这就造成了访问位置的频繁切换，导致了效率的降低。总共有四个缓冲区（聚合引擎的边缘缓冲区和输入缓冲区，以及组合引擎的权重缓冲区和输出缓冲区）将用于访问片外存储器。由于间隔处理和流水线机制，这些访问通常是并发的，如图 5(a) 所示。如果我们按顺序处理这些访问请求，不连续的地址会大大降低 DRAM 内行缓冲区的利用率。为了解决这个问题，我们预先定义了一个访问优先级（边 > 输入特征 > 权重 > 输出特征），以集中处理图 5(b) 所示的不连续的请求。使用这个优先级的动机是基于处理一个顶点时的访问顺序，访问请求是逐批执行的。因此，当前批次的低优先级访问在下一批次的高优先级访问之前被处理，而不是总是高优先级访问在先。有了改进的连续性，行缓冲区的利用率可以得到显著提高。接下来，我们将这些重新排序的地址重新映射到相同的通道和存储库索引。通过这种方式，可以进一步利用内存通道和存储库级别的并行性。

5 复现细节

5.1 复现概述

HyGCN 没有对源码进行开源，为此我们构建了一个模拟器对其性能进行测试。图 6为 HyGCN 的复现示意图。我们使用真实世界中的典型数据集作为输入，通过工作负载生成器生成对应的计算任务和内存访问请求。接着我们将计算任务和内存访问请求作为 HyGCN 模拟器的输入，来生成最终的性能结果输出，包括 GCN 的处理延迟和能耗等指标。HyGCN 模拟器的实现主要包括内存接口和计算逻辑。计算逻辑部分的模拟主要是对微架构组件的行为进行建模，包括聚合引擎和组合引擎的处理行为。内存接口是连接 HyGCN 加速器和片外存储器 DRAM 的桥梁纽带，它对混合引擎的访存请求进行协调、调度，并把请求转发给 DRAM。为了模拟内存请求的响应延迟，我们使用了 DRAMSim3[6]模拟器来对内存访问行为进行建模。

为了得到 HyGCN 各个组件的处理延迟和功耗等参数，我们对硬件组件进行综合和建模。对于计算组件，我们使用硬件描述语言 Verilog HDL 对 HyGCN 的计算逻辑进行寄存器传输级别（Register-

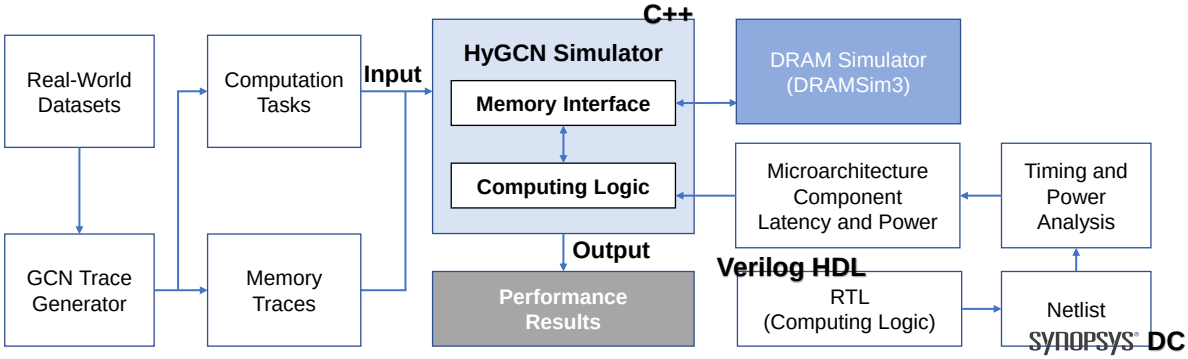

图 6: HyGCN 复现示意图

Transfer Level, RTL）抽象，接着使用 Synopsys Design Compiler 对 RTL 代码进行综合得到门级网表，并最终得到时序和功耗的分析结果。同时，对于 DRAM 和 SRAM 等存储单元，我们使用 CACTI[7]来对其延迟和功耗进行建模。各个组件的能耗和延迟等参数也是 HyGCN 模拟器的输入，用于得到最终的处理延迟和能耗等结果。

在代码实现上，HyGCN 模拟器项目包含硬件组件行为和 GCN 数据流建模的代码。主要的硬件器件包含缓冲模拟器（Scratchpad Memory，SPM）、内存接口协调器（Coordinator）、脉动阵列（Systolic Array）、SIMD 处理核。GCN 数据流由组合和聚合过程的计算任务和内存访问请求序列组成。具体的，HyGCN 模拟器的目录结构如下所示：

(a) dataflow

- aggr.cpp/aggr.h
- comb.cpp/comb.h
- event.cpp/event.h

(b) hardware

- aggr_buf.cpp/aggr_buf.h
- coordinator.cpp/corrdiniate.h
- simd.cpp/simd.h
- spm.cpp/spm.h
- systolic_arr.cpp/systolic_arr.h

(c) config.cpp/config.h

(d) graph.cpp/graph.h

(e) tool.cpp/tool.h

(f) hygcn.cpp/hygcn.h

5.2 计算逻辑

5.2.1 聚合引擎

聚合引擎在硬件上主要包含了 SIMD 处理核、专用缓冲寄存器 Edge Buffer、Input Buffer、Aggreatation Buffer。在数据流上，我们使用了有限状态机（Finite State Nachine, FSM）来实现计算和数据之间依赖关系和运行时状态的建模，达到了时钟周期级别准确的模拟。

5.2.2 组合引擎

组合引擎在硬件上主要包含了脉动阵列模块和专用缓冲寄存器 Weight Buffer。在数据流上，我们同样使用了有限状态机来实现计算和数据之间依赖关系和运行时状态的建模，达到了时钟周期级别准确的模拟。对于脉动阵列，我们使用了权重驻留（Weight-Stationary）数据流，来实现对权重数据的高效复用，减少片外内存的访问。

表 1: 典型图数据集

Dataset	# of vertices	# of edges	# of features	# of classes
Citeseer (CS)	3,327	9,104	3,703	6
Cora (CR)	2,708	10,556	1,433	7
DBLP	17,716	105,734	1,639	4
Pubmed (PB)	19,717	88,648	500	3
Reddit (RD)	232,965	114,615,892	602	41

5.3 内存协调接口

内存协调接口（Coordinator）用于协调 HyGCN 中混合引擎的内存请求同片外内存 DRAM 之间的交互。在 HyGCN 模拟器中，我们使用了基于事件（Event）的片外内存访问请求。这种粗粒度的内存访问有利于充分聚集连续内存地址访问的请求，减少行缓冲区冲突，提高 DRAM 行缓冲区的局部性，提高内存访问性能。在内存协调接口中，我们按数据请求类型和优先级对数据分类排序，并使用批量处理的技术来聚集连续内存地址访问的请求。最终把调度后的内存请求转发给 DRAM 模拟器（DRAMSim3）进行处理。

6 实验结果分析

6.1 实验设置

测试数据集。实验测试中采用了一组具有代表性的 GCN 应用以及典型的图数据集。实验中使用了三个 GCN 应用程序，包括 GCN[2]、GraphSage[8] (GS) 和 Graph Isomorphism Networks[9] (GIN)。在默认配置中，每个 GCN 应用程序都有两层图卷积层，隐藏神经元大小为 128。表 1列出了用作 GCN 应用程序输入的图数据集，包括 Citerseer(CS)、Cora(CR)、DBLP、Pubmed(PB) , 和 Reddit(RD)[10]。

实验基准。我们使用 CPU 平台的软件框架 PyTorch Geometric[4] (PyG-CPU) 同 HyGCN 做对比，对推理延迟和处理能耗进行比较。PyTorch Geometric (PyG) 是一种高效的几何深度学习框架。因此，我们选择 PyG-CPU 同 HyGCN 进行比较。HyGCN 配备了 8GB 的高带宽内存、4.25 MB 缓冲内存、16 个 SIMD 单元每个具有 16 个内核，以及四个 8×64 MAC 脉动阵列模块，所有的运算单元都在 0.5GHz 的频率下工作。对于 PyG-CPU，CPU 平台配备了两个 Intel Xeon 4210R 处理器和总共 256GB 的 DDR4 内存。

6.2 实验结果

加速比。图 7显示了 PyG-CPU 和 HyGCN 之间的加速比，我们以 PyG-CPU 为基准，对处理延迟进行了标准化，得到 HyGCN 的加速比。从数据中可以看到，HyGCN 在处理速度上远远快于 PyG-CPU，平均快了 275 倍。性能的提升主要来源于 HyGCN 的混合处理引擎。这种异构的架构可以高效地处理受限于内存带宽的聚合过程，和受限于计算能力的组合过程。同时，稀疏消除技术有利于进一步减少冗余的内存访问，提高系统的整体性能。

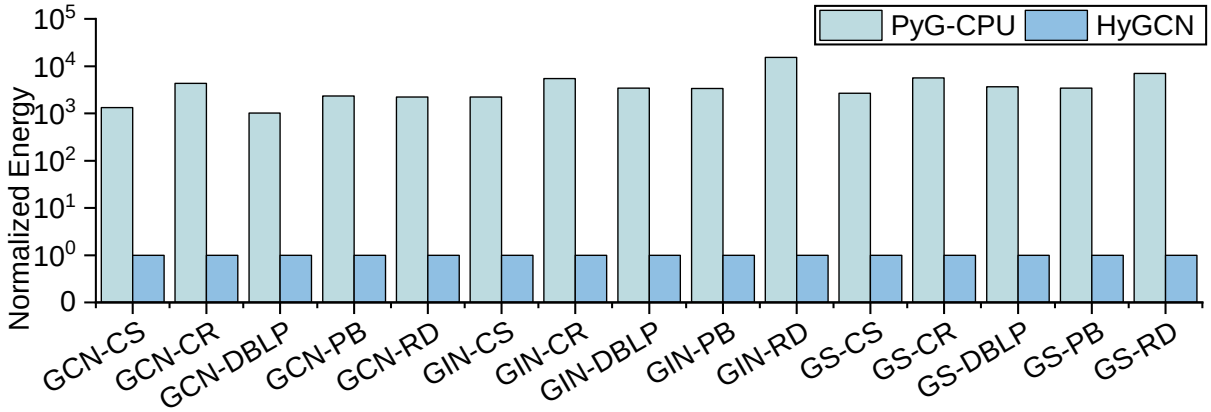

图 8: PyG-CPU 和 HyGCN 的能耗比较

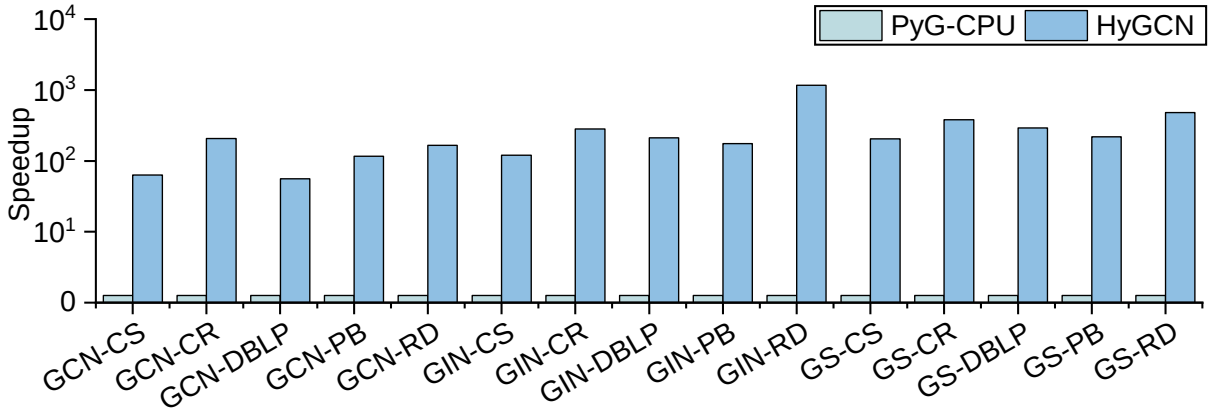

图 7: PyG-CPU 和 HyGCN 的处理加速比

能耗。图 8显示了 PyG-CPU 和 HyGCN 之间的处理能耗比较，我们以 HyGCN 为基准，对处理能耗进行了标准化，得到 PyG-CPU 和 HyGCN 能源消耗的对比。从数据中可以看到，HyGCN 在处理能耗上远远少于 PyG-CPU，平均节约了 4112 倍。能耗的减少一方面来源于 HyGCN 的混合处理引擎。这种异构的架构可以高效地处理受限于内存带宽的聚合过程，和受限于计算能力的组合过程，可以大大缩短处理的时间，这有助于减少静态/漏电能耗。另一方面，稀疏消除技术有利于减少不必要的数据迁移，可以有效减少能耗；而内存接口的调度优化技术，可以提高行缓冲区的命中率，减少行冲突导致的 DRAM 存储库激活操作，这有助于进一步减少能源消耗。

7 总结与展望

我们复现了一种具有混合处理引擎的 GCN 加速器，HyGCN。我们成功复现模拟了 HyGCN 里面的三个主要技术点：基于 SIMD 的聚合引擎、基于脉动阵列的组合引擎以及协调两种混合引擎数据访问的内存接口。未来，我们将基于该模拟测试更多的性能指标，包括内存带宽的使用率和行缓冲区的命中率等，从内存访问的角度来进一步探索实现性能提升的突破口。

参考文献

[1] LU Y C, KIRAN PENTAPATI S S, ZHU L, et al. TP-GNN: A Graph Neural Network Framework for Tier Partitioning in Monolithic 3D ICs[C]//Design Automation Conference (DAC). 2020: 1-6.

[2] KIPF T N, WELLING M. Semi-Supervised Classification with Graph Convolutional Networks[C]// International Conference on Learning Representations (ICLR). 2017: 1-14.

[3] YAN M, DENG L, HU X, et al. HyGCN: A GCN Accelerator with Hybrid Architecture[C]//International Symposium on High Performance Computer Architecture (HPCA). 2020: 15-29.

[4] FEY M, LENSSEN J E. Fast Graph Representation Learning with PyTorch Geometric[C]//International Conference on Learning Representations (ICLR). 2019: 1-9.

[5] HAM T J, WU L, SUNDARAM N, et al. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics[C]//nternational Symposium on Microarchitecture (MICRO). 2016: 1-13.

[6] LI S, YANG Z, REDDY D, et al. DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator [J]. Computer Architecture Letters, 2020, 19(2): 106-109.

[7] CACTI[Z]. https://www.hpl.hp.com/research/cacti. 2008.

[8] HAMILTON W L, YING Z, LESKOVEC J. Inductive Representation Learning on Large Graphs[C]//Advances in Neural Information Processing Systems (NIPS). 2017: 1024-1034.

[9] XU K, HU W, LESKOVEC J, et al. How Powerful are Graph Neural Networks?[C]//International Conference on Learning Representations (ICLR). 2019: 1-17.

[10] Pytorch Geometric[Z]. https://github.com/pyg-team/pytorch_geometric. 2021.