To Appear in 2020 IEEE International Symposium on High Performance Computer Architecture

HyGCN: A GCN Accelerator with Hybrid Architecture

Mingyu Yan[†‡§], **Lei Deng**[§], **Xing Hu**[§], **Ling Liang**[§], **Yujing Feng**[†],
Xiaochun Ye[†*], **Zhimin Zhang**[†], **Dongrui Fan**[†‡], and **Yuan Xie**[§]

[†]State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences

[‡]School of Computer Science and Technology, University of Chinese Academy of Sciences

[§]University of California, Santa Barbara

ABSTRACT

Inspired by the great success of neural networks, graph convolutional neural networks (GCNs) are proposed to analyze graph data. GCNs mainly include two phases with distinct execution patterns. The *Aggregation* phase, behaves as graph processing, showing a dynamic and irregular execution pattern. The *Combination* phase, acts more like the neural networks, presenting a static and regular execution pattern. The hybrid execution patterns of GCNs require a design that alleviates irregularity and exploits regularity. Moreover, to achieve higher performance and energy efficiency, the design needs to leverage the high intra-vertex parallelism in *Aggregation* phase, the highly reusable inter-vertex data in *Combination* phase, and the opportunity to fuse phase-by-phase execution introduced by the new features of GCNs. However, existing architectures fail to address these demands.

In this work, we first characterize the hybrid execution patterns of GCNs on Intel Xeon CPU. Guided by the characterization, we design a GCN accelerator, *HyGCN*, using a hybrid architecture to efficiently perform GCNs. Specifically, first, we build a new programming model to exploit the fine-grained parallelism for our hardware design. Second, we propose a hardware design with two efficient processing engines to alleviate the irregularity of *Aggregation* phase and leverage the regularity of *Combination* phase. Besides, these engines can exploit various parallelism and reuse highly reusable data efficiently. Third, we optimize the overall system via inter-engine pipeline for inter-phase fusion and priority-based off-chip memory access coordination to improve off-chip bandwidth utilization. Compared to the state-of-the-art software framework running on Intel Xeon CPU and NVIDIA V100 GPU, our work achieves on average 1509× speedup with 2500× energy reduction and average 6.5× speedup with 10× energy reduction, respectively.

1. INTRODUCTION

Inspired by the powerful learning capability of neural networks, graph convolutional neural networks (GCNs) are proposed as an effective category of models to represent and process graph data [38, 45, 46, 47]. GCNs convert the graph data into a low dimensional space while keeping both the structure and property information to the maximum extent, and then construct a neural network for the consequent training and inference. Recently, GCNs attract substantial efforts from both the industrial and academic communities [3, 14, 18, 20, 28, 43, 44] to solve problems including node classification [25], link prediction [14, 16], graph clustering [44], and recommendation [12]. As a result, GCNs gradually become a new workload family member in data-centers, such as in Google [13], Facebook [28], and Alibaba [3, 47].

The convolutional layers occupy the major execution time of GCNs through two primary execution phases: *Aggregation* and *Combination* [15, 40, 47]. The *Aggregation* phase maintains most graph processing behaviors. It heavily relies on the graph structure that is inherently random and sparse. Processing of each vertex requires aggregating features from all its source neighbours. Unfortunately, the amount and location of these source neighbors vary significantly among vertices. As a result, the computational graph [26] and memory access pattern in the *Aggregation* phase of each vertex are dynamic and irregular. The *Combination* phase acts more like the neural networks. It transforms the feature vector of each vertex to a new one using a multi layer perceptron (MLP), which is usually expressed by a matrix-vector multiplication (MVM). Due to the identical connection pattern of each neuron within a neural network layer, the computational graph [26] and memory access pattern in the *Combination* phase of each vertex are static and regular. Besides, there are additional characteristics in these two phases that distinguish GCNs from conventional workloads. First, the length of vertex property is short and fixed in conventional graph analytics. However, in GCNs, the feature vector of each vertex is quite long and variable across layers, which introduces high-degree intra-vertex parallelism in *Aggregation* phase. Second, the parameters in conventional MLP-based neural networks are never shared, while they can be fully shared among vertices in GCNs, which induces abundant highly reusable inter-vertex data in *Combination* phase. Third, the two phases are executed alternatively. An inherent dataflow exists between phases, providing an opportunity to fuse the phase-by-phase execution.

To achieve high-performance and energy-efficient acceleration of GCNs, aforementioned characteristics have imposed new requirements on architecture design. First, not only can the GCN architecture alleviate the irregularity in *Aggregation* phase, but it can also exploit the regularity in *Combination* phase. Second, it needs to exploit the high-degree intra-vertex parallelism and highly reusable inter-vertex data. Third, it is able to efficiently fuse the execution of these two phases.

*Corresponding author is Xiaochun Ye and his email is yexiaochun@ict.ac.cn.

Unfortunately, existing architectures fail to implement GCN-specific characteristics. For CPUs, although they can employ complex caching and prefetching techniques to offset the processor-memory disparity by exploiting the regular access pattern [11], they fail to address the abundant dynamic and irregular data accesses in the *Aggregation* phase since the irregularity harms the predictability of memory accesses [10]. Besides, it is difficult to efficiently implement the reuse of the highly reusable parameter data between computing units in CPUs as like TPU [22] and Eyeriss [8]. Thus, the energy-hungry data accesses to cache introduce high energy consumption [8]. For GPUs, although they are well optimized for neural networks, they lack the ability to alleviate irregularity in *Aggregation* phase, which significantly hinders the performance improvement [30, 41]. Furthermore, although they leverage the regularity in *Combination* phase, the data copy and synchronization between threads for the parameter reuse are expensive. For graph analytics and neural network accelerators, they are only optimized to alleviate irregularity or exploit regularity, rather than both simultaneously. At last, all of them are short of the ability to efficiently fuse the execution of these two phases. In conclusion, existing architectures are not the ideal platforms to execute GCNs.

In this work, we first characterize the hybrid execution patterns of GCN workloads on Intel Xeon CPU. Next, guided by the characterization, we propose a GCN accelerator, *HyGCN*, using a hybrid architecture to efficiently perform GCNs. Specially, we first propose a programming model to achieve the hardware transparency for programmers and exploit fine-grained parallelism. It abstracts GCNs as edge-centric aggregation for the *Aggregation* phase and MVMs for the *Combination* phase. Second, we design *HyGCN* with two efficient processing engines, *Aggregation Engine* and *Combination Engine*, to accelerate the *Aggregation* and *Combination* phases, respectively. In *Aggregation Engine*, interval-shard graph partitioning and window sliding-shrinking methods are introduced to alleviate irregularity by increasing data reuse and decreasing unnecessary accesses for sparsity, respectively. Additionally, we implement a vertex-disperse processing method to exploit the edge parallelism and intra-vertex parallelism. In *Combination Engine*, to leverage the regularity, we build multi-granular systolic arrays to perform MVMs in parallel and reuse the shared parameters. Besides, they can be flexibly used either independently for lower latency or in combination for lower energy. Third, to improve the overall execution, on the basis of individual optimizations of these two phases, we build a fine-grained inter-engine pipeline to fuse the phase-by-phase execution and propose a priority-based memory access coordination for the off-chip data accesses between the two engines.

To summarize, we list our contributions as follows:

- We study an emerging domain, GCNs, from a computer architecture perspective and show that hybrid execution patterns exist in GCNs. Specially, the *Aggregation* phase in GCNs presents a dynamic and irregular execution pattern, while *Combination* phase is static and regular.
- We propose a GCN accelerator, *HyGCN*, using a hybrid architecture to efficiently perform GCNs. First, we build a programming model to enable our hardware design to exploit various parallelisms inherent in this domain. Next, we propose a hardware design to tackle irregularity and leverage regularity with *Aggregation Engine* and *Combination Engine*, respectively.
- We propose a flexible inter-engine pipeline and a priority-based memory access coordination to efficiently fuse the execution of *Aggregation* phase and *Combination* phase.
- We implement our architecture design in RTL and evaluate it using a detailed microarchitectural simulation. We use four well-known GCN models on six popular graph datasets. Compared to the state-of-the-art software framework *PyTorch Geometric* [15] running on Intel Xeon CPU and NVIDIA V100 GPU, our work achieves on average 1509× speedup with 2500× energy reduction and 6.5× speedup with 10× energy reduction, respectively.

2. BACKGROUND

GCNs follow a neighborhood aggregation scheme, where the feature vector of each vertex is computed by recursively aggregating and transforming the representation vectors of its neighbor vertices [18, 39, 47]. Fig. 1 illustrates the execution phases of GCN models. After k iterations of aggregation via the **Aggregate** function and transformation via the **Combine** function, a vertex is represented by its final feature vector, which captures the structural information within the vertex's k-hop neighborhood. Table 1 lists the notations used in GCNs. In this work, we mainly focus on undirected graphs and the inference stage rather than training.

Table 1: GCN Notations.

Notation	Meaning	Notation	Meaning
$\mathbf{G}$	graph $\mathbf{G}=(V,E)$	V	vertices of $\mathbf{G}$
E	edges of $\mathbf{G}$	D_v	degree of vertex v
$e_{(i,j)}$	edge between vertex i and j	$N(v)$ ($S(v)$)	(sampling subset of) v' neighbor set
A (A_{ij})	(element of) adjacent matrix	a_v	aggregation feature vector of v
h_G	feature vector of $\mathbf{G}$	W	combination weight matrices
h_v	feature vector of vertex v	b	combination bias vectors
X	initialized feature matrix	Z	embedding matrix
C	assignment matrix	ε	learnable parameter

Typically, the k-th layer/iteration of GCNs is formulated as

$$a_v^k = \textbf{Aggregate}\big(h_u^{(k-1)} : u \in \{N(v)\} \cup \{v\}\big), \qquad h_v^k = \textbf{Combine}\big(a_v^k\big). \tag{1}$$

where h_v^k is the representation feature vector of vertex v at the k-th iteration. Simply, the **Aggregate** function aggregates multiple feature vectors from source neighbors to one single feature vector, and the **Combine** function transforms the feature vector of each vertex to another feature vector using an MLP neural network. Note that the MLP parameters, including weights and biases, are shared between vertices.

In order to decrease the computational complexity, the **Sample** function is usually applied before the **Aggregate** function to sample a subset from the neighbor vertices of each vertex [6, 18] as the new neighbors, specifically,

$$S(v) = \textbf{Sample}^k\big(N(v)\big). \tag{2}$$

Sometimes, the **Pool** function [44] follows the **Combine** function to transform the original graph into a smaller graph.

After several iterations, the features will be used for final prediction or classification. For the node classification problem, vertex feature vectors h_v^k at the last iteration are used for prediction. For the graph classification problem, a **Readout** function further aggregates the h_v^k at the last iteration to obtain

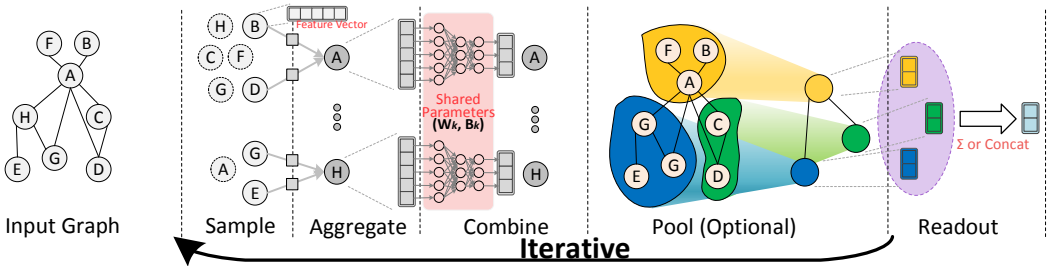

Figure 1: Illustration of the GCN model.

the entire graph's representation vector, i.e.

$$h_G = \textbf{Readout}\big(h_v^k \mid v \in G\big). \tag{3}$$

Next, we provide several typical GCN models as examples to explain the above operations in detail.

GCN is one of the most successful convolutional networks for graph learning [25, 38], which bridges the gap between spectral-based convolutions and spatial-based convolutions. Its inference model can be described as

$$a_v^k = \Big(\sum \frac{1}{\sqrt{D_v \cdot D_u}} h_u^{(k-1)} \mid \forall u \in \{N(v)\} \cup \{v\}\Big), \qquad h_v^k = ReLU(W^k a_v^k + b^k). \tag{4}$$

GraphSage further adopts uniform neighbor sampling to alleviate receptive field expansion that effectively trades off accuracy and execution time [18]. It is formulated as

$$a_v^k = Mean\big(\{h_v^{(k-1)}\} \cup \{h_u^{(k-1)}, \forall u \in S(v)\}\big), \qquad h_v^k = ReLU(W^k a_v^k + b^k). \tag{5}$$

GINConv is a simple neural architecture, and its discriminative power is equal to the power of the Weisfeiler-Lehman graph isomorphism test [39]. Vertex features learned by GINConv can be directly used for tasks like node classification and link prediction. We can perform this model as

$$a_v^k = (1+\varepsilon_k) \cdot h_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)}, \qquad h_v^k = MLP^k(a_v^k,\ W^k,\ b^k). \tag{6}$$

For graph classification tasks, the following **Readout** function is further used to produce the representation of the entire graph, given the representations of individual vertices. It concatenates across all iterations of GINConv to acquire the final graph representation as

$$h_G = \textbf{Concat}\big(\big(\sum_{v \in G} h_v^k\big) \mid k = 1, ..., K\big). \tag{7}$$

DiffPool provides a general tool to realize hierarchical graph-level transformation for a broad set of input graphs [44]. It can be inserted after the **Combine** function of any GCNs to transform the original graph to a smaller one (like the pooling layer in convolutional neural networks (CNNs)). In fact, Diffpool uses two extra GCNs to implement the graph transformation, which follows

$$C^{(k-1)} = softmax\big(GCN_{pool}^k(A^{(k-1)},\ X^{(k-1)})\big), \qquad Z^{(k-1)} = GCN_{embedding}^k(A^{(k-1)},\ X^{(k-1)}), \qquad X^k = C^{(k-1)^T} Z^{(k-1)},\ \ A^k = C^{(k-1)^T} A^{(k-1)} C^{(k-1)}. \tag{8}$$

After the DiffPool transformation, a new feature matrix X^k and adjacent matrix A^k are produced, which can be combined to construct a new and smaller graph. In the new graph, GCN_{pool}^k determines the number of vertices, and $GCN_{embedding}^k$ determines the length of vertex feature vector.

Summary. As explained above, we introduce several typical operations in GCNs: *Sampling*, *Aggregation*, *Combination*, *Pooling*, and *Readout*. Except for *Combination*, all the operations are graph structure-dependent, which involve graph processing. *Combination* usually is a typical MLP neural network (single layer or multiple layers). *Sampling* is used to sample a subset from neighbors, which can be done during preprocessing [20] or with random selection during runtime [18]. *Aggregation* aggregates the features from its 1-hop neighbors. *Pooling* acts like the pooling layer in CNNs to realize graph transformation by reducing the number of vertices and the length of feature vectors. *Readout* can be a simple summation [15] across vertices or further concatenation across iterations [39]. Therefore, *Readout* can be viewed as an extreme *Aggregation*. This work focuses on *Aggregation* and *Combination*, two major phases in GCNs.

3. MOTIVATION

In this section, we quantitatively characterize and identify the hybrid execution patterns in processing GCNs. Next, we explain our motivation behind designing a GCN accelerator.

3.1 Characterization on CPU

We conduct quantitative characterizations using a state-of-the-art GCN software framework *PyTorch Geometric* [15] on Intel Xeon CPU. The execution time breakdown of GCN (GCN) [25], GraphSage (GSC) [18], and GINConv (GIN) [39] on several datasets [23] is illustrated in Fig. 2. The profiling results of GCN [25] on the COLLAB dataset [23] are presented in Table 2. The details of system configuration and datasets are shown in Section 5.1.

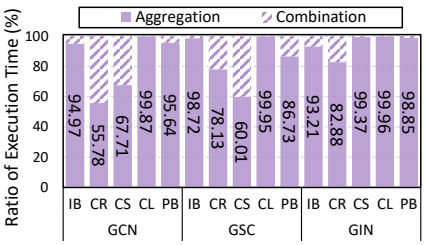

Figure 2: Execution time breakdown of the two phases.

Execution Time Breakdown. Both of *Aggregation* and *Combination* phases can occupy a significant amount of execution time, which implies that both need acceleration. Fig. 2 illustrates their execution time ratio on different models and

datasets. Their execution times differ due to the variable length of feature vectors and the execution flow of GCNs. For example, the long feature length of CR and CS datasets causes more time on *Combination* phase for GCN and GraphSage. Since GINCov executes *Aggregation* phase first, it spends more time on *Aggregation* phase without the reduction of feature length through *Combination* phase like the other two models do.

Hybrid Execution Pattern. The *Aggregation* phase heavily relies on the graph structure that is inherently random and sparse, which results in numerous dynamic computations and irregular accesses. From Table 2, it is observed that each operation in the *Aggregation* phase requires much more data to be accessed from DRAM than *Combination* phase, resulting in higher DRAM access energy. Besides, the extremely high numbers of misses per kilo-instruction (MPKI) of L2 and L3 caches in the *Aggregation* phase are caused by the high randomness of neighbor indices of each vertex. In addition, the indirect and irregular accesses render the data prefetching in the *Aggregation* phase ineffective, since it is difficult to predict the data addresses without knowing the indices of neighbors in advance. This causes abundant ineffectual memory accesses to prefetch data.

The *Combination* phase executes a MVM for each vertex with a shared MLP-based neural network, which performs static and regular computations and accesses. Table 2 illustrates that each operation in the *Combination* phase requires only small amount of data to be accessed from DRAM. This is because the MVMs are very compute-intensive and the weight matrix of MLP is widely shared between vertices. Nevertheless, up to 36% of execution time for shared data copy and synchronization between threads is observed.

Table 2: Quantitative Characterization on CPU.

	Aggregation	*Combination*
DRAM Byte per Ops	11.6	0.06
DRAM Access Energy per Ops	170nJ	0.5nJ
L2 Cache MPKI	11	1.5
L3 Cache MPKI	10	0.9
Ratio of Synchronization Time	—	36%

Table 3: Different Execution Patterns of *Aggregation* Phase and *Combination* Phase.

	Aggregation	*Combination*
Access Pattern	Indirect & Irregular	Direct & Regular
Data Reusability	Low	High
Computation Pattern	Dynamic & Irregular	Static & Regular
Computation Intensity	Low	High
Execution Bound	Memory	Compute

According to above analysis, hybrid execution patterns exist in GCNs, which are summarized in Table 3. The *Aggregation* phase performs dynamic and irregular execution pattern, bounded by memory, while the *Combination* phase is static and regular, bounded by computation.

Differences from Conventional Workloads. Beside hybrid execution patterns in GCNs, there are additional characteristics that distinguishes GCNs from conventional workloads. Specifically, in the *Aggregation* phase, the length of feature vectors is variable rather than fixed as in traditional graph analytics, which is determined by the input dataset and MLP structure. Moreover, the length of the feature vectors in each vertex is usually orders of magnitude longer than that of traditional graph analytics. This introduces high intra-vertex parallelism. In the *Combination* phase, the MLP parameters are fully shared by all vertices while non-reusable in traditional MLP models if not using the batching technique. This induces numerous highly reusable inter-vertex data. Besides, these two phases are executed alternatively to produce the final result, while conventional workloads iteratively perform only the graph traversal or the neural network propagation.

3.2 The Need for a GCN Accelerator

GCNs are showing great potential in various tasks [16, 18, 19, 38, 43, 46]. Many companies, such as Google [13], Facebook [28], and Alibaba [47] have deployed GCNs in data centers, which reflects the increasing importance and scope of upcoming applications. An efficient architecture is timely to achieve high performance and stimulate GCN development. Therefore, given the above characterizations, we explain our motivation of designing a GCN accelerator.

Design Requirements. Given the characteristics of GCNs, we present the design requirements to perform GCNs with high performance and energy efficiency. First, *Aggregation* phase demands efforts to alleviate the irregularity that degrades performance. On the other hand, *Combination* phase needs more attention to leverage the regularity to improve the intensive computations with better parallelism and faster synchronization. Second, the high-degree intra-vertex parallelism and the highly reusable inter-vertex data need to be exploited. Third, to achieve higher performance and energy efficiency, the execution of *Aggregation* phase and *Combination* phase need to be efficiently fused. Unfortunately, existing architectures fail to address these requirements, resulting in the following inefficiencies.

Inefficiencies of General-Purpose Processors. On CPUs, the irregularity in *Aggregation* phase makes GCNs ill-suited to current cache hierarchy design and data prefetching techniques. Besides, it is hard to efficiently reuse the highly reusable parameter data between compute units [8].

GPUs are inherently optimized for compute-intensive workloads with regular execution pattern [29] such as neural networks, but handling the *Aggregation* phase with irregular memory accesses suffers from low efficiency. Besides, the processing of *Combination* with strong parameter sharing needs costly data copy and thread synchronization.

Both CPUs and GPUs lack inter-phase optimization for GCN execution. To leverage the advantages of hardware-optimized functions [4, 31], current programming framework for GCNs usually adopts coarse-grained execution, which results in phase-by-phase execution. This compromises the design space with phase interaction, hindering the improvement beyond the individual optimization for each phase.

Inefficiencies of Conventional Accelerators. Specialized accelerators tailored to graph analytics or neural networks gain significant speedup and energy savings compared to general-purpose processors. Whereas, they are inefficient in processing GCNs due to following reasons: i) they are usually only designed to either alleviate irregularity or exploit regularity, while GCNs need both; ii) they fail to leverage the

new kinds of parallelism and data reuse to further improve performance; iii) single-paradigm design make them hard to fuse the execution of the two phases.

Opportunities for Customization. Designing a specialized accelerator for a specific domain is an efficient and prevalent solution to address the inefficiencies of existing architectures, since it can tailor the memory hierarchy and computation unit to the specific workload. For GCNs, we can build an accelerator with a hybrid architecture using different optimizations for the two phases. For the *Aggregation* phase, it is possible to obtain the knowledge of graph data in advance and schedule the accesses to alleviate the irregularity. Moreover, the computation for each vertex can also be scheduled to exploit edge parallelism and intra-vertex parallelism. For the *Combination* phase, we draw inspirations from current neural network accelerators to efficiently perform MVMs in parallel with parameter sharing. Beyond the individual optimizations of the two phases, the serial inter-phase dataflow can be pipelined in finer grain. Moreover, all off-chip memory accesses can be controlled to improve the overall memory access efficiency. Putting all these together, there are huge opportunities to design an efficient GCN accelerator with high performance.

4. ARCHITECTURE DESIGN

In this section, we design *HyGCN* to support the efficient execution of GCNs. We first introduce the programming model and then present details of the architecture design.

4.1 Edge- and MVM-Centric PM

The goal of building a programming model (PM) is to exploit available parallelisms and achieve hardware transparency for programmers [47]. For *Aggregation*, there are gather- and scatter-based processing methods. Since the scatter-based method usually produces large amount of atomic operations and requires a synchronization after the processing of all vertices, the degree of parallelism will be degraded [17]. On the contrary, the gather-based method can control the program behavior easily and preserve the execution parallelism. Therefore, we select the gather-based processing in our design. Nevertheless, this processing mode leads to intensive memory access and vertex computation. To address this problem, we employ an edge-centric PM to exploit the edge-level parallelism. Each vertex possesses many incoming edges (neighbors), which can be aggregated in an edge-by-edge pipeline. In this way, workload for each vertex can be divided into subworkloads and assigned to each computation unit for processing in parallel. For *Combination*, the situation is relatively easier. Since the computation of each vertex acts like the MLP, we directly focus on the MVM operations.

Our edge- and MVM-centric PM for GCNs is shown in Algorithm 1. At each vertex $v \in V$, the sampled neighbor indices are read first, which is a subset of all neighbors. Each index corresponds to an edge connecting v and a neighbor vertex u, i.e. $e(u, v)$. By traversing all sampled edges connected v, all the feature vectors of corresponding neighbors can be aggregated onto the feature vector of v. Then, a ***Combine*** function can start performing the *Combination* phase that is comprised of a series of MVMs. In this PM, the edge-level and MVM-level parallelism can be exploited.

Note that in Algorithm 1 we do not express the Pool and Readout operations explicitly since they are not always needed. In fact, the Pool operation can be represented by two GCNs and additional matrix operations. The GCNs can be performed entirely by the two engines, the matrix transposes can be executed by the flexible *Aggregation* engine, and the matrix multiplications can be executed by the *Combination* engine. The Readout operation can be expressed by an additional single vertex that connects all vertices in the graph, which can be accomplished by the *Aggregation* engine.

Algorithm 1: ***Edge- and MVM-centric Programming Model for Aggregation and Combination Phase***

```
initial SampleNum;
initial SampleIndexArray;
for each node v ∈ V do
    agg_res ← init();
                ◁ Edge-centric Parallelism
    sample_idxs ← SampleIndexArray[v.nid];
    for each sample_idx in sample_idxs do
        e(u,v) ← EdgeArray[sample_idx];
        agg_res ← Aggregate(agg_res, u.feature);
    end
                ◁ MVM-centric Parallelism
    v.feature ← Combine(agg_res, weights, biases);
end
```

4.2 Architecture Overview

Based on the proposed PM, Fig. 3 depicts the architecture of *HyGCN*. We construct the system using a hybrid architecture, which includes two engines (*Aggregation Engine* and *Combination Engine*) and one memory access handler. A communication interface (*Coordinator*) is introduced to bridge these two engines. Therefore, the interference between them is mitigated and their execution pipeline is established.

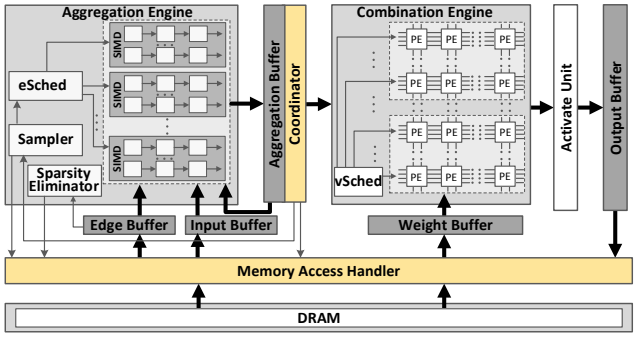

Figure 3: Architecture overview of *HyGCN*.

The *Aggregation Engine* aims to realize the efficient execution of irregular accesses and computations. To exploit the edge-level parallelism, a task scheduler (*eSched*) is designed to assign the edge processing workloads onto SIMD cores. To support the Sampling operation, we introduce a *Sampler* into the *Aggregation Engine*. The *Sampler* selects edges from the edge list of each vertex using a uniform or predefined distribution in terms of index interval. The former indices for edge sampling are based on dynamic generation while the latter ones are predefined and can be read from

off-chip memory like in [6, 20]. To reduce the latency of data access, we employ embedded DRAM (eDRAM) to cache various data to improve data reuse. An *Edge Buffer* is used to cache edges to exploit spatial locality in the edge array. An *Input Buffer* is used to cache the vertex features in X^{k-1} and an *Aggregation Buffer* is used to cache the intermediate aggregation results, to exploit temporal locality. To hide the DRAM access latency, both the *Edge Buffer* and *Input Buffer* adopt the double buffer technique. Specifically, we design a *Sparsity Eliminator* to avoid redundant feature loads of the vertices that share no edges with the aggregating vertex.

The *Combination Engine* is designed to maximize the efficiency of regular accesses and computations. In order to improve the processing parallelism and data reuse, we adopt the well-known systolic array design [22] and modify it to be compatible with GCNs. A *Weight Buffer* is used to cache the weight matrix to exploit their temporal locality, and an *Output Buffer* is used to coalesce the write accesses of the final features. Similarly, they also leverage the double buffer technique to hide off-chip access latency. The *Combination* engine takes the aggregation result of each vertex v from the *Aggregation* engine and the weight matrix from the *Weight Buffer* as inputs to execute the MVM operation. The *vSched* is responsible for the workload assignment. After the MVM operations, an activation operation is performed by *Activate Unit* to produce the new feature vector of vertex v. Different from normal systolic array, our systolic array is multi-granular that can be used as multiple smaller arrays or a whole large array under different optimization scenarios.

To improve the bandwidth utilization, a prefetcher is designed to explicitly prefetch graph data and parameter data. For example, the prefetching of the feature vectors is as follows. The prefetcher first prefetches the edges of current processing vertices. After receiving these edges, *Sparsity Eliminator* obtains the indices of neighbors from these edges and sends them to the prefetcher. The prefetcher uses them to prefetch the feature vectors immediately.

4.3 Aggregation Engine

To optimize the computation of *Aggregation*, we introduce a vertex-disperse processing mode. To optimize memory accesses, we employ a static graph partition method to enhance data reuse and a dynamic sparsity elimination technique to reduce unnecessary data accesses.

4.3.1 Execution Mode

There are two processing modes for SIMD cores to process edges in parallel. The first one is vertex-concentrated, where the workloads of each vertex are assigned to a single SIMD core. This mode can produce the aggregated features of vertices in burst mode, i.e. periodically processing a group of vertices. However, the processing latency of a single vertex (termed as vertex latency) is long, and the fast vertices have to wait for the slow vertices leading to workload imbalance. Furthermore, it also loses the parallelism that the aggregation of each element can be performed in parallel (i.e., intra-vertex parallelism). Therefore, we use the second processing mode, which is shown in Fig. 4. It assigns the aggregation of elements inside the vertex feature vector of each vertex to all cores, termed as vertex-disperse mode. If a vertex cannot occupy all cores, free cores can be assigned to other vertices. Thus, all cores are always busy without workload imbalance. Moreover, since the intra-vertex parallelism has been exploited, the vertex latency for a single vertex is smaller than processing multiple vertices together. Furthermore, it also enables the immediate processing of each vertex in the following *Combination Engine*.

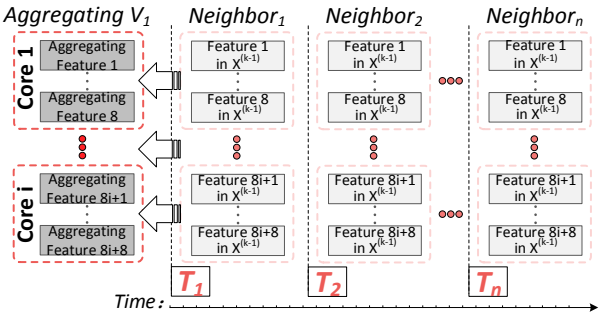

Figure 4: Vertex-disperse processing mode where the workloads of each vertex are assigned to all SIMD cores.

4.3.2 Graph Partitioning (Static)

We borrow the abstraction of vertex interval and edge shard from [9, 27] to partition graph data, which is the basis of our data-aware sparsity elimination in the next subsection. We do not need explicit preprocessing to generate the intervals and shards since we directly take the data format of compressed sparse column (CSC) as input. As exampled in Fig. 5(a), the 16 vertices are organized as several intervals (i.e. from I_1 to I_4, each with four vertices), and the edges are organized as 4×4 shards (i.e. from $S(1,1)$ to $S(4,1)$, each with 16 edges at most). The intervals and shards are disjoint.

The feature vector length of each vertex is usually large, so exploiting the locality of features is critical. We group the vertices within the same interval together (e.g. I_i) and then process the aggregation of their source neighbors also interval by interval (i.e. traverse I_j), as expressed in Algorithm 2. Based on this flow, the feature accesses of all vertices in an interval are merged (see Fig. 5(b)). The resulting benefits are twofold. First, the vertices in I_i usually have overlapped neighbors in I_j, therefore, the loaded feature data of I_j can be reused when performing feature aggregation. Second, when traversing all I_j, the intermediate aggregated results of I_i are remained in buffer which can also be reused when performing feature update. In practice, edge shards usually are not square as our simplified illustration in Fig. 5. The shard height is determined by the capacity of *Input Buffer*, while the shard width is determined by the capacity of *Aggregation Buffer*. The *Edge Buffer* size affects both height and width since it accommodates all edges of each shard.

4.3.3 Data-Aware Sparsity Elimination (Dynamic)

With the data reuse optimization, we further attempt to reduce the redundant accesses since the graph connections are sparsely distributed. To eliminate the sparsity, we propose a window-based sliding and shrinking approach. The key idea is that we first slide the window (with the same size of an edge shard) downward until an edge appears in the top row, and then we shrink the window size by moving the bottom row upward until an edge is met.

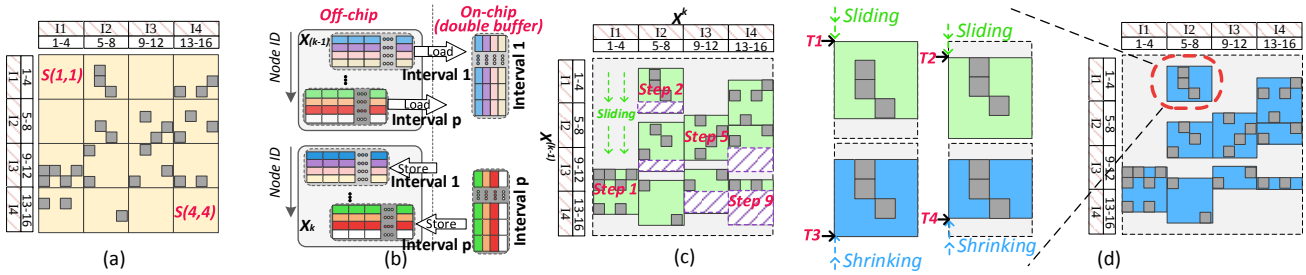

Figure 5: Static graph partition for data reuse and dynamic sparsity elimination to reduce redundant accesses: (a) interval-shard partition; (b) interval-wise feature access; (c) window sliding; (d) window shrinking.

Window Sliding. Fig. 5(c) illustrates the window sliding process. For each vertex interval, the top shard window gradually slides downward. It will not stop until an edge appears on its top row. Then a new window with the same size is created, whose top row follows the bottom row of its previous window. The stop criterion is the same for every window. In this way, windows continuously arise, slide downward, and stop. All the positions where windows stop are recorded as effectual shards.

Algorithm 2: ***Interval-wise Aggregation***

```
for each interval I_i in X^k do
    agg_res ← init();
    for each interval I_j in X^(k-1) do
        agg_res ← Aggregation(I_j, agg_res);
    end
    I_i ← Combination(agg_res);
end
```

Window Shrinking. Although the window sliding can capture most effectual edges, sparsity still exists on the bottom side (within the purple dashed boxes). This is because the above sliding direction is downward. To reduce this part of sparsity, we propose window shrinking here. Specifically, the bottom row of each recorded window moves upward until it meets an edge, and then the window shrinks. Fig. 5(d) illustrates the sliding and shrinking process of one window in detail and gives the final recorded effectual shards. Different from previous partition, the sizes of final shards are usually different due to the window shrinking.

Algorithm 3: ***Interval-wise Aggregation with Sparsity Elimination***

```
for each interval I_i in X^k do
    row_pos ← 1;
    agg_res ← init();
    do
        (I_j, row_pos) ←
          GetOneEffectInterval( X^(k-1), A, I_i, row_pos);
        agg_res ← Aggregation(I_j, agg_res);
    while (I_j != ∅);
    I_i ← Combination(agg_res);
end
```

Given the effectual shards after sparsity elimination, the execution flow of *Aggregation* follows Algorithm 3. The only difference from Algorithm 2 is that the each neighbor interval I_j is dynamically determined by window sliding and shrinking (see Algorithm 4). The starting row of each neighbor interval varies due to sliding and the interval length in the row dimension also varies due to shrinking. In this way, only the feature data of remaining neighbor vertices when performing the aggregation operation for each interval I_i are loaded, which eliminates plenty of redundant accesses.

Algorithm 4: ***GetOneEffectInterval***

```
                                  ◁ Window Sliding
while (edge(row_pos, v) == ∅ for ∀v ∈ I_i) do
    row_pos ← row_pos + 1;
end
win_start ← row_pos;
win_end ← row_pos + Window_height − 1;
row_pos ← win_end + 1;
                                  ◁ Window Shrinking
while (edge(win_end, v) == ∅ for ∀v ∈ I_i) do
    win_end ← win_end − 1;
end
I_effectual ← X^(k-1)[win_start : win_end];
return I_effectual;
```

Compared to traditional graph analytics, the feature data reuse from graph partitioning and redundant access reduction from sparsity elimination in GCNs are considerable efforts. This is because the feature of each vertex in GCNs is a vector with thousands of elements, while the feature data in traditional graph analytics are small, usually with one element for each vertex. Besides, our optimization achieves more when the Sampling operation is used, which increases sparsity since only sampled neighbors are required during *Aggregation*.

4.4 Combination Engine

The *Combination* operation at each vertex acts like a neural network, the execution of which is regular but compute-intensive. Our design is based on the well-known systolic array. To adapt it for the two processing modes of *Aggregation Engine* (see Fig. 4), we integrate multiple arrays rather than a single one, as shown in Fig. 6(a). A group of systolic arrays is assembled to form a systolic module. We allow a multigranular use of these systolic modules, including the independent working mode and cooperative working mode.

4.4.1 Independent Working Mode

In this mode, the systolic modules work independently from each other. Each of them processes the MVM opera-

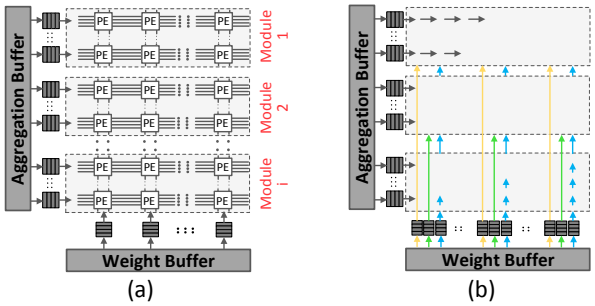

Figure 6: *Combination Engine* design: (a) multiple systolic modules; (b) different dataflow patterns.

tions of a small group of vertices, as illustrated in Fig. 7(a). The weight parameters for each module in this case are directly accessed from the *Weight Buffer* and just reused within module, as depicted in Fig. 6(b). The advantage of this mode is the lower vertex latency because we can process the *Combination* operations of this small group of vertices immediately once their aggregated features are ready, without waiting for more vertices. This mode matches well with the vertex-disperse processing mode of *Aggregation Engine* in Fig. 4, where the aggregated features are produced quickly but sequentially.

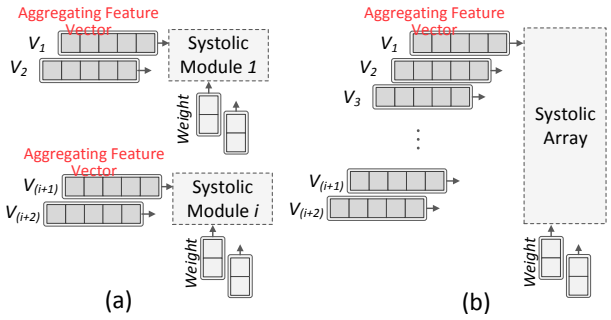

Figure 7: Different use of the systolic arrays: (a) independent working mode; (b) cooperative working mode.

4.4.2 *Cooperative Working Mode*

Besides working separately, these systolic modules can be further assembled together to simultaneously process more vertices, as shown in Fig. 7(b). Different from the immediate processing of vertices, this mode requires to assemble the aggregated features of a large group of vertices together before performing their *Combination* operations. The advantage is that, the weight parameters can flow from the *Weight Buffer* to the downstream systolic modules and then gradually to the upstream ones (see Fig. 6(b)), which are greatly reused by all systolic arrays. This helps reduce the energy consumption.

No matter which working mode is selected in the *Combination Engine*, the weights can be reused inherently in *Weight Buffer* when processing different vertices. However, in traditional neural networks, especially MLPs, the weights cannot be shared without batching technique. The multi-granular systolic array design is also specific to our architecture in order to accommodate different application needs.

4.5 Inter-Engine Optimization

To efficiently fuse the phase-by-phase execution, we orchestrate the execution pipeline and DRAM access of *Aggregation* engine and *Combination* engine by the *Coordinator*.

4.5.1 *Latency- or Energy-Aware Pipeline*

Ping-Pong Aggregation Buffer. To reuse the aggregation results produced by the *Aggregation* engine, we add an *Aggregation Buffer* between the two engines. This buffer can be written by the *Aggregation Engine* and can be read by the *Combination Engine*. Before the final aggregated results are generated, the *Aggregation Buffer* stores the partial results that will be read by the *Aggregation Engine* for feature accumulation. In order to increase the parallelism of these two engines, we implement a ping-pong buffering mechanism where the *Aggregation Buffer* is split into two chunks. In this way, the executions of aggregation and combination are decoupled, which enables an inter-engine pipeline.

To accommodate the needs of different applications, we provide two pipeline modes as follows.

Latency-Aware Pipeline. In this pipeline mode, the *Combination Engine* works in the systolic module independent mode. The aggregated features are produced vertex by vertex in the *Aggregation Engine*, and the following combination will be processed immediately once the aggregated features of a small group of vertices are ready. Therefore, the average processing latency for each vertex can be lower. The overall timing is illustrated in Fig. 8(a), where V denotes the vertices for aggregation, and I represents the neighbor intervals.

Energy-Aware Pipeline. The energy-aware pipeline uses the systolic module cooperative mode in the *Combination Engine*. The vertex-by-vertex processing changes to a burst mode, where a large group of vertices will be processed together every time. Although the vertex latency is longer, the energy consumption can be reduced due to the weight propagation in the merged systolic arrays without redundant accesses. Fig. 8(b) presents its timing sequence.

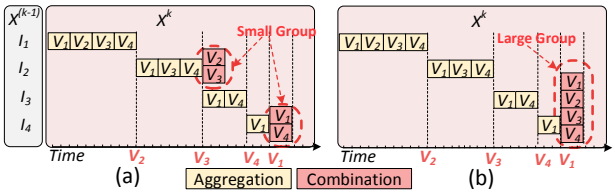

Figure 8: Timing illustration of different pipeline modes: (a) latency-aware pipeline; (b) energy-aware pipeline.

4.5.2 *Coordination of Off-chip Memory Access*

It is hard to determine the memory bandwidth ratio between the two engines since the practical workloads usually vary between *Aggregation* and *Combination*. Moreover, the separation of memory systems will increase the configuration overheads and cause bandwidth waste. This is the reason why we use only one off-chip memory.

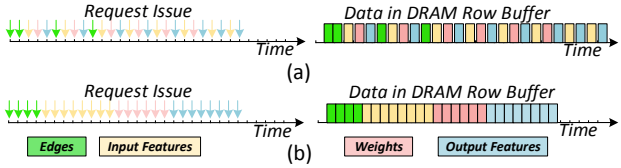

Figure 9: Coordination of off-chip memory access.

Both the two engines access this memory at runtime, which

causes a frequent switching of access locations, leading to inefficiencies. In total, there are four buffers (*Edge Buffer* & *Input Buffer* in *Aggregation Engine*, and *Weight Buffer* & *Output Buffer* in *Combination Engine*) that will be used for accessing the off-chip memory. Due to the interval processing and pipeline mechanism, these accesses usually come concurrently as shown in Fig. 9(a). If we sequentially handle these access requests, the discontinuous addresses greatly degrade the utilization of row buffer within DRAM.

To solve this problem, we predefine an access priority ($edges > input features > weights > output features$) to assemble the discontinuous requests shown in Fig. 9(b). The motivation in using this priority is based on the access sequence when processing a vertex. The access requests are executed batch-by-batch. Therefore, low-priority accesses in the current batch are handled before high-priority accesses coming at the next batch, rather than always high-priority accesses first. With the improved continuity, the utilization of row buffer can be significantly enhanced. Next, we remap these reordered addresses to index the channel and bank using low bits. In this way, the memory channel- and bank-level parallelism can be further exploited.

5. EVALUATION RESULTS

We first describe our experimental setup in Section 5.1. Next, to demonstrate the advantages of our design, we compare *HyGCN* to the state-of-the-art software framework in Section 5.2. Next, we give the detailed analysis of our optimization techniques in Section 5.3. Finally, we present a scalability exploration of our architecture in Section 5.4.

5.1 Experimental Setup

Methodology. The performance and energy of *HyGCN* are measured by using the following tools.

Architecture Simulator. We design and implement a cycle-accurate and execution-driven simulator to measure execution time in number of cycles. This simulator models the microarchitectural behaviors of each module, which is integrated with Ramulator [24] to simulate the behaviors of memory accesses to High Bandwidth Memory (HBM).

CAD Tools. For the measurements of area, power, and critical path delay (in cycles) for each module, we implement and synthesize each module in Verilog. We use the Synopsys Design Compiler with the TSMC 12 nm standard VT library for the synthesis, and estimate the power using Synopsys PrimeTime PX. The slowest module has a critical path delay of 0.9 ns including the setup and hold time, putting the *HyGCN* comfortably at 1 GHz clock frequency.

Memory Measurements. The area, power, and access latency of the on-chip scratchpad memory are estimated using Cacti 6.5 [1]. Since Cacti only supports down to 32 nm technologies, we apply four different scaling factors to convert them to 12 nm technology as shown in [33, 36]. The energy of HBM 1.0 is estimated with 7 pJ/bit as in [32, 41].

Benchmark Graph Datasets and GCN Models. Table 4 and Table 5 provide the information of the benchmark graph datasets and GCN models used in our evaluation. The datasets in Table 4 are standard ones in the GCN domain. They are actually not small although the number of vertices is smaller than that used in conventional graph analytics, due to the long length of feature vectors. On CPU, the datasets with more than one graphs are tested by assembling randomly selected 128 graphs into a large graph before processing for GCN, GSC, and GIN or batching the same number of graphs for DFP. On *HyGCN*, the testing methods remain the same with CPU except that the selected graphs for DFP are processed one by one rather than in a batched mode.

Table 4: Dataset information [23, 42].

Dataset	#Vertex	Feature Length	#Edge	Storage
IMDB-BIN (IB)	2,647	136	28,624	1.5MB
Cora (CR)	2,708	1,433	10,556	15MB
Citeseer (CS)	3,327	3,703	9,104	47MB
COLLAB (CL)	12,087	492	1,446,010	28MB
Pubmed (PB)	19,717	500	88,648	38MB
Reddit (RD)	232,965	602	114,615,892	972MB

Table 5: Configuration of convolution layers. Here $|a_v^k|$ denotes the length of feature vector a_v^k.

	#Sampling Neighbors	Aggregation & Combination (MLP)
GCN (GCN)	—	Add & $\|a_v^k\|$–128
GraphSage (GSC)	25	Max & $\|a_v^k\|$–128
GINConv (GIN)	—	Add & $\|a_v^k\|$–128–128
DiffPool (DFP)	GCN_{pool}	$\text{GCN}_{embedding}$
	Min & $\|a_v^k\|$–128	Min & $\|a_v^k\|$–128

Baseline Platform. To compare the performance and energy consumption of *HyGCN* with state-of-the-art works, we evaluate PyTorch Geometric (PyG) [15] on a Linux workstation equipped with two Intel Xeon E5-2680 v3 CPUs and 378 GB DDR4 memory and on an NVIDIA V100 GPU, denoted as PyG-CPU and PyG-GPU, respectively. Table 6 lists the system configurations for above implementations.

Table 6: System configurations.

	PyG-CPU	PyG-GPU	*HyGCN*
Compute Unit	2.5 GHz @ 24 cores	1.25Ghz @ 5120 cores	1 GHz @ 32 SIMD16 cores and 8 systolic modules (each with 4×128 arrays)
On-chip Memory	60MB	34MB	128 KB (Input), 2 MB (Edge), 2 MB (Weight), 4 MB (Output) and 16 MB (Aggregation)
Off-chip Memory	136.5GB/s DDR4	∼900GB/s HBM∼2.0	256GB/s HBM∼1.0

Note: GPU's on-chip memory includes the register files, and L1 and L2 caches.

5.2 Overall Results

We first apply our algorithm optimization on PyTorch Geometric. And then, we compare our work (*HyGCN*) with PyG-CPU and PyG-GPU in terms of speedup, energy consumption, utilization of DRAM bandwidth, and DRAM access. Finally, the area and power of our design is presented.

• ***Algorithm Optimization on PyG Framework.*** To show the effect of our algorithm optimization on CPU and GPU platforms, we implement our algorithm optimization proposed in Section 4.3 on PyG framework. The graph is partitioned into multiple shards and they are executed shard by shard (see Fig. 5(a)). The number of partitions is determined by the capacity of L2 Cache and the length of feature vectors. Note that, PyG leverages the Pytorch Scatter library [4] for the acceleration of *Aggregation* on both CPU and GPU. It helps eliminate the sparsity and exploit the edge parallelism by executing each vertex's *Aggregation* in a hardware thread.

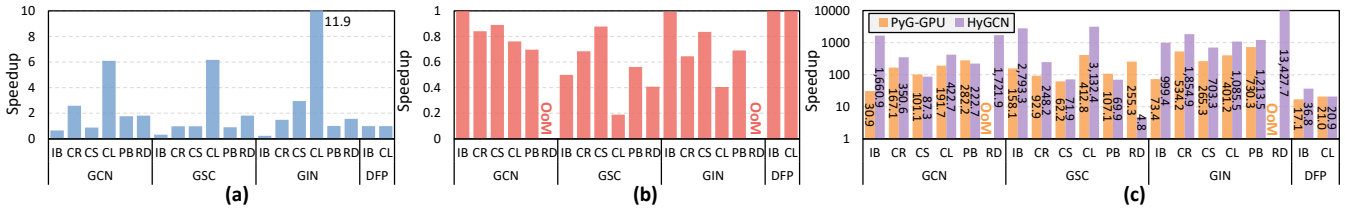

Figure 10: Comparistion to PyG-CPU and PyG-GPU: Speedup of our algorithm optimization on (a) CPU and (b) GPU; (c) Speedup over the optimized PyG-CPU. OoM means the evaluation fails in running on GPU due to out of memory.

Furthermore, the hardware-optimized libraries such as Intel MKL [21] and NVIDIA cuBLAS library [31] are used to accelerate *Combination* on CPU and GPU, respectively.

Fig. 10(a) shows the speedup of PyG-CPU with our algorithm optimization (PyG-CPU-OP) over the naive one without optimization. Thanks to the algorithm improvement, PyG-CPU-OP achieves 2.3× speedup on average. The performance benefits come from the reduction of frequent replacement of feature vectors since the reusable features after graph partition and the intermediate results of *Aggregation* are buffered in L2 Cache. Fig. 10(b) presents the same testing on GPU. The performance of PyG-GPU-OP degrades since only a small amount of vertices are processed for each graph partition, which cannot fully utilize thousands of hardware threads on GPU and miss the core advantage of GPU to hide the access latency through many parallel threads. As a result, it is inefficient for GPU to exploit our optimization to improve performance. The optimized PyG-CPU and the naive PyG-GPU are used as baselines in the following evaluation.

• ***Speedup.*** Fig. 10(c) depicts that *HyGCN* achieves average 1509× and 6.5× speedup compared with PyG-CPU and PyG-GPU, respectively. The performance improvement comes from the individual optimizations in *Aggregation Engine* & *Combination Engine*, and the inter-engine pipeline & coordination. First, the parallel processing in SIMD cores and systolic arrays speed up the computations. Second, the graph partition and sparsity elimination increase the feature reuse and decrease redundant accesses in *Aggregation Engine*, which saves DRAM bandwidth. Third, the weight parameters are reused efficiently in *Combination Engine*, which also helps better utilize the bandwidth. Finally, the inter-engine pipeline further optimizes the parallelism and the off-chip memory access coordination improves the DRAM access efficiency.

For PyG-CPU and PyG-GPU, abundant DRAM accesses and synchronization overheads lead to performance degradation. Specifically, the high randomness of neighbor indices results in poor locality of neighbors' feature vectors, causing many unnecessary DRAM accesses. From the perspective of computation, PyG-CPU and PyG-GPU leverage the hardware-optimized functions (such as scatter [4] and matrix multiplication [31]) to perform GCNs in a coarse-grained fashion. Although it is the best way to utilize CPU and GPU, it loses the inter-phase parallelism and produces redundant operations. The delay for data copy and synchronization between threads further degrades the performance.

In term of models, GIN achieves better performance than others. The underlying reason is that GIN executes *Aggregation* first on PyG-CPU and PyG-GPU, which introduces abundant computations and accesses since the feature vector size is an order of magnitude larger than that after *Combination*. By contrast, other models execute *Combination* first, which greatly reduces the feature length before performing *Aggregation*. This difference causes the inefficient execution of GIN on CPU and GPU, while our *HyGCN* can maintain the performance to a great extent due to the parallel processing and data reuse. For DFP, it includes three matrix multiplications (see Equation (8)) that can be efficiently executed on CPU and GPU. Therefore, our speedup when performing DFP is relatively lower. The GSC model consumes significant time on the Sampling operation in a preprocessing step, which is not included in the result of PyG-CPU and PyG-GPU. For example on the RD dataset, the preprocessing can cost up to 15 seconds while the execution time is only 0.65 second on PyG-CPU and 0.0025 second on PyG-GPU. In our work, the Sampling operation is executed together with *Aggregation* and considered in the reported result. Thus, the performance of our work is lower than PyG-GPU in Fig. 10(c) but the overall execution time ratio is 0.136 second v.s. 15.7 seconds.

• ***Energy Consumption.*** As Fig. 11 shows, *HyGCN* consumes only 0.04% and 10% energy on average compared to PyG-CPU and PyG-GPU, respectively. The energy consumption of all platforms includes the off-chip memory. Note that, although the results of PyG-CPU and PyG-GPU do not include the overhead of the Sampling operation, they are still costly. For example, the Sampling energy of GSC is 2715J on the RD dataset. In contrast, our work consumes only 1.79J compared to the total 2716J in PyG-GPU.

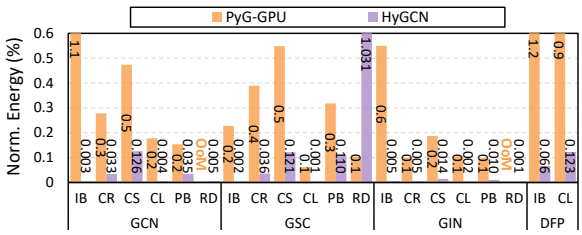

Figure 11: Normalized energy over PyG-CPU.

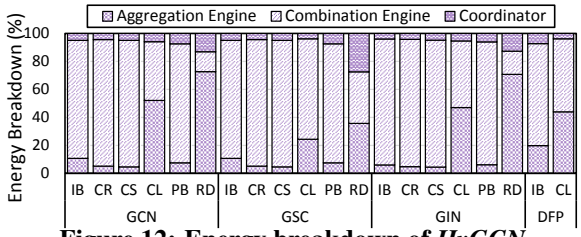

Figure 12: Energy breakdown of *HyGCN*.

As aforementioned, GIN causes additional computations and data accesses when performing *Aggregation*, which introduces extra energy consumption on PyG-CPU. Although *HyGCN* cannot reduce these computations, the optimizations of data reuse, sparsity elimination, and inter-engine pipeline can reduce redundant accesses to these additional

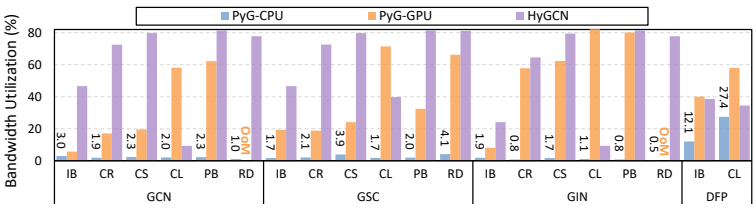

Figure 13: Bandwidth utilization of all platforms.

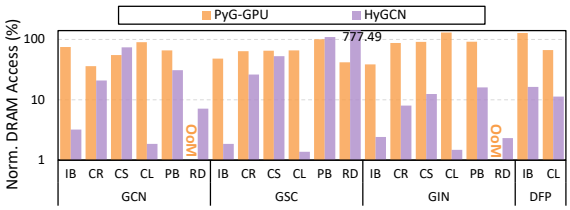

Figure 14: Normalized data access to PyG-CPU.

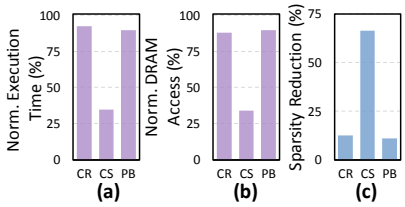

Figure 15: Effect of sparsity elimination on (a) execution time, (b) DRAM access, and (c) sparsity reduction.

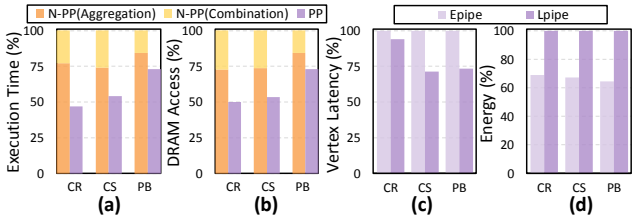

Figure 16: Effect of inter-engine pipeline on (a) execution time and (b) DRAM access, and the comparison of (c) vertex latency and (d) energy of *Combination Engine* under different pipeline modes.

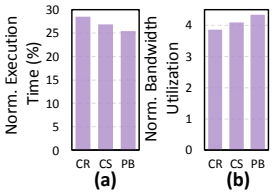

Figure 17: Effect of memory coordination on (a) execution time and (b) bandwidth utilization.

data. Among the architectural components, *Combination Engine* consumes most of the energy due to the intensive computation of MVMs as depicted in Fig. 12, while *Aggregation Engine* consumes more energy on high-degree graph datasets (i.e., CL and RD).

• ***DRAM Bandwidth Utilization.*** As seen in Fig. 13, *HyGCN* demonstrates 16× and 1.5× improvement on average on the utilization of DRAM bandwidth compared with PyG-CPU and PyG-GPU, respectively. The high bandwidth utilization of *HyGCN* and PyG-GPU derive from the high-degree parallelism. By contrast, PyG-CPU cannot sufficiently exploit the bandwidth, since there is only one thread most of time to reduce the heavy overheads of frequent thread creation. Our consistent lower bandwidth on the CL dataset is due to the higher data reuse, which benefits from denser connections.

• ***DRAM Access.*** Although the 16MB on-chip memory is much smaller than the 60MB L3 cache on CPU and 34MB on GPU, *HyGCN* accesses only 21% and 33% of off-chip data compared with PyG-CPU and PyG-GPU on average, respectively, as given in Fig. 14. This benefits from our data reuse optimizations, sparsity elimination, and the immediate processing between two engines. On the CL dataset for GCN, GSC, and GIN, multiple graphs are assembled to form a larger one before being processed, which results in intensive sparsity. *HyGCN* can efficiently eliminate the sparsity via window sliding and shrinking, thus avoiding unnecessary data accesses. Whereas, PyG-CPU and PyG-GPU produce many unnecessary accesses due to the irregularity in *Aggregation* phase and without the fusion of phase-by-phase execution. As aforementioned, the results of PyG-CPU and PyG-GPU do not include the data access of the Sampling operation. For example, the Sampling access volume of GSC is 56.5GB on the RD dataset. In contrast, our work only accesses 28GB data, compared with the total 58GB in PyG-GPU.

5.2.1 Power and Area

The total power and area of *HyGCN* are only 6.7 *W* and 7.8 mm^2, respectively. For the on-chip buffer, we use eDRAM to reduce both the area and energy consumption. For the computation precision, we use 32-bit fixed point that is enough to maintain the accuracy of GCN inference. Table 7 provides area and power breakdown in terms of buffer, computation, and control. The computation resources of two engines consume most of power (>64%) and area (>44%) to perform the edge-centric aggregation and MVMs-based combination. The *Coordinator* occupies ∼35% of the total area since it has a large *Aggregation Buffer*. The control overhead is small (only 1.2% power and <0.45% area) owing to the simple implementations of *eSched*, *Sampler*, *Sparsity Eliminator*, *vSched*, *Coordinator*, and *Memory Handler*.

Table 7: Layout characteristics of *HyGCN*

Module	Component	Power (%)	Area (%)
Aggregation Engine	Buffer	2.37	5.41
	Computation	3.85	1.43
	Control	0.48	0.18
Combination Engine	Buffer	14.4	15.13
	Computation	60.52	42.96
	Control	0.31	0.07
Coordinator	Buffer	17.66	34.64
	Control	0.41	0.19

5.3 Optimization Analysis

In this subsection, we analyze the effect of our optimization techniques including sparsity elimination, inter-engine pipeline, and off-chip memory access coordination. The benchmark model is GCN mentioned in Table 5.

5.3.1 Sparsity Elimination Optimization

We evaluate *HyGCN* with and without sparsity elimination. This experiment runs only *Aggregation Engine* to avoid the interference of other blocks. Fig. 15(a) shows that *HyGCN* achieves 1.1∼3× speedup with the optimization of sparsity elimination. The performance gain is due to fewer redundant DRAM accesses as reflected in Fig. 15(b), which benefits from eliminated sparsity as given in Fig. 15(c).

5.3.2 Inter-Engine Pipeline Optimization

First, we measure the overall performance with and without inter-engine pipeline optimization (PP v.s. N-PP). With the pipeline optimization, the execution time of GCN is reduced by 27%-53%, as shown in Fig. 16(a). On one hand, the *Aggregation Engine* and *Combination Engine* work in parallel with inter-engine pipeline. On the other hand, the DRAM accesses occupy most of the execution time (see Fig. 16(b)), therefore the inter-engine pipeline helps improve the performance by decreasing DRAM accesses of the intermediate aggregation results between two engines. It is observed from Fig. 16(b) that total DRAM accesses are significantly reduced to only 50%-73% with this pipeline optimization.

Second, we compare the vertex latency and energy of *Combination Engine* with energy-aware pipeline and latency-aware pipeline (Epipe v.s. Lpipe). From Fig. 16(c), the Lpipe reduces the average latency for each vertex by 7%-29% via the immediate processing without waiting for the aggregation results of many vertices. By contrast, as shown in Fig. 16(d), the Epipe saves energy consumption by 35% via assembling a large group of vertices to process together for reusing weight parameters aggressively. In practice, the application requirement determines the pipeline mode.

5.3.3 Memory Coordination Optimization

To show the effect of the memory access coordination, we present the execution time and bandwidth utilization with and without coordination in Fig. 17(a) and Fig. 17(b), respectively. With the memory access coordination for address continuity, the DRAM row buffers are better utilized and the channel-/bank-level parallelism is better exploited, which saves 73% of time and improves 4× bandwidth on average.

5.4 Scalability Exploration

The following evaluations are measured in GSC model.

• ***Sparsity Elimination with Sampling.*** The Sampling operation increases the sparsity, thus it has the potential to enlarge the benefits produced by sparsity elimination. In Fig. 18(a)-(c), horizontal axis sweeps the *sampling factor*. It indicates that only $\frac{1}{sampling\ factor}$ edges of each vertex are sampled to perform aggregation. As the increasing *sampling factor*, the performance is significantly improved on the PB dataset by reducing the DRAM accesses owing to the higher sparsity. For other datasets, since many edges have been removed, the *Combination* phase gradually dominates the execution time. Therefore, there is no significant speedup. Note that the *sampling factor* cannot be too high, as it might harm the accuracy of applications.

• ***Capacity of Aggregation Buffer.*** The size of the *Aggregation Buffer* affects the execution time, amount of data accesses, and even the effect of sparsity elimination. As the capacity of *Aggregation Buffer* increases from 2 MB to 32 MB, the exeuction time is decreased as shown in Fig. 18(d). This can be explained from two aspects: i) more intermediate aggregated feature data can be cached in on-chip buffer, leading to larger shard width when partitioning the graph and thus less execution loops; ii) larger shard means that the neighbor features can be reused more often, leading to less DRAM accesses (see Fig. 18(e)). However, larger shard also enlarges the window size during the sparsity elimination, which results in higher sparsity that cannot be eliminated (see Fig. 18(f)).

• ***Size of Systolic Module.*** In this experiment, we fix the number of total systolic arrays but change the size of each systolic module, and then to measure the cost of *Combination Engine*. Different from the systolic module with 4×128 systolic arrays in Table 6, here we treat 1×128 systolic arrays as a basic systolic module. Based on the initial 32 systolic modules, we gradually decrease the number of systolic modules under the restriction of fixed number of total systolic arrays. It is observed that longer latency for a vertex is consumed as the partition of systolic modules becomes more coarse-grained as shown Fig. 18(g)(bar). This is caused by the longer time to assemble a larger group of vertices to be processed together. Fortunately, the energy consumption can be reduced as shown Fig. 18(g)(red line) because the weight parameters are reused by more vertices within each larger systolic module. We only present the average energy result of these datasets for simplicity. In our architecture design, we set the systolic module with size of 4×128 arrays to achieve a good trade-off between the latency and energy costs.

6. DISCUSSION

In order to leverage our proposed PM, PyG needs to be significantly modified for its coarse-grain message-passing mechanism to stream *Aggregation* and *Combination* for each vertex. Note that although these two phases can be streamed after modification, it also misses the advantage of hardware-optimized operations, such as matrix multiplication operation [21,31]. Furthermore, further challenges exist with 1) inefficient memory subsystem due to workload-agnosticism [17], 2) difficulty in data reuse like systolic arrays [22], and 3) expensive on-line preprocessing for workload reorganization and streaming.

Following concerns make training unsuitable as a starting work to explore GCN hardware. First, training involves three passes with data dependency: forward, backward, and update, whose compute and memory patterns are more complex than that of inference with only the forward pass. Second, the gradient propagation in graphs is far more complicated than layer-by-layer propagation in neural networks. However, training accelerators can leverage our architecture to design the forward pass, and would need specialized blocks for other passes and an efficient memory hierarchy to connect them.

7. RELATED WORK

Plenty of software frameworks for graph analytics and neural networks have been presented to release the programming efforts while achieving high performance on modern general-purpose architectures [5,34,35,37]. However, all of them only work well for the single-pattern workloads. Therefore, a large number of software frameworks for hybrid-pattern GCNs are proposed recently [2, 15, 28, 43, 47]. For instance, PyTorch Geometric [15] leverages message-passing framework to enhance its expression ability and the hardware-optimized operations (e.g. scatter and matrix multiplication) so that the GCN workloads can be accelerated. Unfortunately, the distinct execution pattern regarding computation and access between the *Aggregation* phase and the *Combination* phase produces processing inefficiencies on traditional platforms. GCNs demand specialized architecture design.

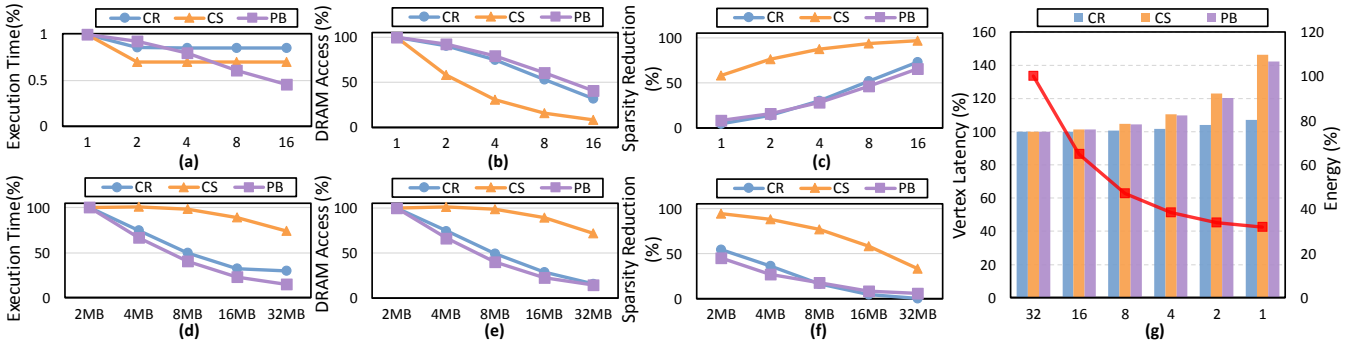

Figure 18: Scalability exploration. i) sparsity elimination with different *sampling factor*: (a) Execution time, (b) DRAM access, and (c) sparsity reduction; ii) capacity of *Aggregation Buffer*: (d) execution time, (e) DRAM access, and (f) sparsity reduction; iii) size of the systolic module: (g) vertex latency and energy of *Combination Engine*.

With the emergence of graph analytics and neural networks workloads, a lot of hardware architecture designs are proposed to accelerate these workloads [7, 8, 17, 22, 33]. For example, Graphicionado [17] is tailored for graph analytics; while TPU [22] focuses on the acceleration of neural networks. However, GCNs behave not only like the graph processing (*Aggregation*) but also like neural networks (*Combination*), leading to intrinsic hybrid design requirement. Therefore, current specialized architectures cannot efficiently perform GCNs since they just handle one of the two sides.

8. CONCLUSION

GCNs are becoming widely adopted for analyzing graph data and are comprised of *Aggregation* and *Combination* phases. In this work, we identify that the execution patterns of these two phases are distinct, even almost opposite, which requires separate design requirements. Besides, the high intra-vertex parallelism in *Aggregation* phase, the highly reusable inter-vertex data in *Combination* phase, and the opportunity to fuse phase-by-phase execution introduced by the new features of GCNs need to be leveraged for better performance. To this end, we propose a GCN accelerator, *HyGCN*, with hybrid architecture. First, we build edge- and MVM-centric programming model to exploit various parallelisms and enable hardware transparency. Next, we propose the hardware design with two efficient engines to optimize the two phases correspondingly. The latency- and energy-aware inter-engine pipelines are orchestrated to improve the overall latency and energy according to system needs. The off-chip memory accesses between the two engines are carefully coordinated to improve the efficiency. Finally, through comprehensive evaluations, *HyGCN* demonstrates significant improvements compared to the software framework running on CPU and GPU. We believe our work will stimulate more attention on specialized hardware for increasingly important GCNs.

Acknowledgments

We thank the anonymous reviewers of HPCA 2020 and the sealer in Scalable Energy-efficient Architecture Lab (SEAL) for their constructive and insightful comments. This work was supported by the National Key Research and Development Program of China (Grant No. 2018YFB1003501), the National Natural Science Foundation of China (Grant No. 61732018, 61872335, and 61802367), the Strategic Priority Research Program of Chinese Academy of Sciences (Grant No. XDA 18000000), the Innovation Project Program of the State Key Laboratory of Computer Architecture (Grant No. CARCH4408, CARCH4412, and CARCH4502), the National Science Foundation (Grant No. 1730309, 1725447 and CCF 1740352), and SRC nCORE NC-2766-A.

9. REFERENCES

[1] Cacti. [Online]. Available: http://www.hpl.hp.com/research/cacti/

[2] Deep graph library. [Online]. Available: https://docs.dgl.ai

[3] "A distributed graph deep learning framework." [Online]. Available: https://github.com/alibaba/euler

[4] "Pytorch extension library of optimized scatter operations." [Online]. Available: https://github.com/rusty1s/pytorch_scatter

[5] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[6] J. Chen, T. Ma, and C. Xiao, "Fastgcn: Fast learning with graph convolutional networks via importance sampling," *CoRR*, vol. abs/1801.10247, 2018.

[7] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. ACM, pp. 269–284.

[8] Y. H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379.

[9] Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, and H. Yang, "Nxgraph: An efficient graph processing system on a single machine," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, May 2016, pp. 409–420.

[10] I.-H. Chung, C. Kim, H.-F. Wen, and G. Cong, "Application data prefetching on the IBM blue gene/q supercomputer," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, pp. 1–8.

[11] F. Dahlgren, M. Dubois, and P. Stenstrom, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 7, pp. 733–746, July 1995.

[12] H. Dai, Z. Kozareva, B. Dai, A. J. Smola, and L. Song, "Learning steady-states of iterative algorithms over graphs," in *ICML*, 2018.

[13] DeepMind. Graph nets library. [Online]. Available: https://deepmind.com/research/open-source/graph-nets-library

[14] D. Duvenaud, D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 2224–2232.

[15] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[16] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in Neural Information Processing Systems 30*, pp. 6530–6539.

[17] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct. 2016, pp. 1–13.

[18] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems 30*, pp. 1024–1034.

[19] W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *IEEE Data Eng. Bull.*, vol. 40, pp. 52–74, 2017.

[20] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Advances in Neural Information Processing Systems 31*, pp. 4558–4567.

[21] Intel. Mkl. [Online]. Available: https://software.intel.com/en-us/mkl

[22] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. ACM, pp. 1–12.

[23] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann, "Benchmark data sets for graph kernels," 2016. [Online]. Available: http://graphkernels.cs.tu-dortmund.de

[24] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, pp. 45–49, Jan. 2016.

[25] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[26] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. CasÃğaval, "How much parallelism is there in irregular applications?" in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '09. ACM, pp. 3–14.

[27] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USENIX Association, pp. 31–46.

[28] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large-scale graph embedding system," *CoRR*, vol. abs/1903.12287, 2019.

[29] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[30] A. H. Nodehi Sabet, J. Qiu, and Z. Zhao, "Tigr: Transforming irregular graphs for gpu-friendly graph processing," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 622–636.

[31] NVIDIA. cublas. [Online]. Available: https://developer.nvidia.com/cublas

[32] M. O'Connor, "Highlights of the high-bandwidth memory (hbm) standard," in *Memory Forum Workshop*, 2014.

[33] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 166–177.

[34] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[35] N. Sundaram, N. Satish, M. M. A. Patwary, S. R. Dulloor, M. J. Anderson, S. G. Vadlamudi, D. Das, and P. Dubey, "Graphmat: High performance graph analytics made productive," *Proc. VLDB Endow.*, vol. 8, no. 11, pp. 1214–1225, Jul. 2015.

[36] O. Villa, D. R. Johnson, M. Oconnor, E. Bolotin, D. Nellans, J. Luitjens, N. Sakharnykh, P. Wang, P. Micikevicius, A. Scudiero, S. W. Keckler, and W. J. Dally, "Scaling the power wall: A path to exascale," in *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2014, pp. 830–841.

[37] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the gpu," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: ACM, 2016, pp. 11:1–11:12.

[38] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019.

[39] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *CoRR*, vol. abs/1810.00826, 2018.

[40] K. Xu, C. Li, Y. Tian, T. Sonobe, K. Kawarabayashi, and S. Jegelka, "Representation learning on graphs with jumping knowledge networks," *CoRR*, vol. abs/1806.03536, 2018.

[41] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 615–628.

[42] P. Yanardag and S. Vishwanathan, "Deep graph kernels," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*. ACM Press, pp. 1365–1374.

[43] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '18. ACM, pp. 974–983, event-place: London, United Kingdom.

[44] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *Advances in Neural Information Processing Systems 31*, 2018, pp. 4800–4810.

[45] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *CoRR*, vol. abs/1812.04202, 2018.

[46] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun, "Graph neural networks: A review of methods and applications," *CoRR*, vol. abs/1812.08434, 2018.

[47] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *CoRR*, vol. abs/1902.08730, 2019.