HyGCN: A GCN Accelerator with Hybrid Architecture

ABSTRACT

Inspired by the broad use of graph data and powerful learning capability of neural networks, graph convolutional neural networks (GCNs) are proposed to analyze graph datasets using neural networks. The convolutional layers occupy the major execution time of GCNs through two primary execution phases: *Aggregation* and *Combination*. The former behaves as graph processing while the latter acts more like the neural networks. In order to identify the bottleneck when performing GCNs, we conduct quantitative characterizations that evidence the inefficiency in the conventional architectures. This is caused by the distinct, even opposed, memory access and computation patterns of the two phases, as well as the serialized processing of them.

To address these issues, we propose the concept of GCN accelerator and implement it using a hybrid architecture. First, we build Edge- and MVM (matrix vector multiplication)-centric programming models for the dynamic & irregular *Aggregation* phase and the static & intensive *Combination* phases, respectively, to achieve the hardware transparency for programmers. Then, we design *HyGCN* with two efficient processing engines to respectively accelerate the two phases. In *Aggregation Engine*, besides the edge parallelism achieved by SIMD cores, we introduce the interval-shard graph partition to increase data reuse and the window sliding-shrinking method to decrease redundant accesses. In *Combination Engine*, we build multigranular systolic arrays to perform MVMs that can flexibly be used in an independent way for lower latency or in a joint way for lower energy. At last, we further optimize the overall system via orchestrating the inter-engine pipeline and off-chip memory access coordination. Through extensive evaluation experiments, our work achieves significant improvements compared with the state-of-the-art software framework running on Intel Xeon CPU. We also analyze the optimization techniques and design space to give more insights for future researches on GCN hardwares.

1. INTRODUCTION

Although deep learning has made a great success on Euclidean data (e.g. images), there is a boom of tasks required to analyze large graph datasets where the data are generated from the non-Euclidean domain [33, 40]. These graph-encoded data can capture abundant and complex relationships among billions of elements, such as friendship, topology and interaction. Typical graphs include social networks [19, 30], physical systems [6, 13], biological networks [14, 15], knowledge graphs [5, 17, 31], and so forth.

Inspired by the powerful learning capability of neural networks, graph neural networks (GNNs) are proposed as an effective category of models to represent and process graph data [33, 39, 40, 41]. GNNs convert the graph data into a low dimensional space while keeping both the structural and property information to the maximum extent, and then construct a neural network for the consequent training and inference. Recently, they attract substantial efforts from both the industrial and academic community [3, 13, 18, 20, 27, 37, 38] to solve problems including node classification [24], link prediction [13, 15], graph clustering [38], recommendation [12], and many others. Among existing GNN models, graph convolutional neural networks (GCNs) are the most broadly studied, which have great potential to extend current deep learning scope [26]. As a result, GCNs gradually become a new workload family member in data-centers, such as Euler of Alibaba [3], and Pytorch Biggraph of Facebook [27].

The convolutional layers occupy the major execution time of GCNs through two primary execution phases: *Aggregation* and *Combination* [14, 35, 41]. In the *Aggregation* phase, the feature vectors of its source neighbors are aggregated to one feature vector, which maintains most graph processing behaviors; while the *Combination* phase transforms the feature vector of each vertex to a new one using a multi-layered perceptron (MLP), which acts more like the neural networks. It seems that GCNs appear as a perfect hybrid paradigm that enables the usage of the neural network methodology for analyzing the graph data, however, the efficient processing of GCNs naturally suffers from huge challenges due to this fusion. The underlying reason is that the graph processing and neural network execution have distinct, even almost opposed, memory and computation patterns.

The *Aggregation* phase heavily relies on the graph structure that is inherently random and sparse. The processing of each vertex requires features from all its source neighbours. Unfortunately, the amount and location of these source neighbours vary significantly among vertices. This induces plenty of dynamic and irregular accesses and computations, and thus the data cannot be well shared to utilize the cache hierarchy. On the contrary, the *Combination* phase performs a static and regular transformation on each vertex, which is friendly for the memory subsystem. However, the computation of matrix-vector multiplications (MVMs) is very intensive. Moreover, since the parameters (e.g. weights and biases) for each vertex are fully shared in this phase, the data copy and synchronization between threads cause inefficient waits. To sum up, the *Aggregation* phase requires more design efforts on the memory optimization to address the irregularity for better cache utilization while the *Combination* phase needs a better parallelism and efficiency to perform the intensive MVM computations and reduce the wait time.

Conventional general-purpose processors cannot perform GCNs with high performance. First, although modern computer architectures can employ complex caching techniques to offset the processor-memory disparity, one of the required premises is the regular access pattern. Unfortunately, the abundant dynamic and irregular accesses in the *Aggregation* phase ruin the memory locality and the predictability of memory accesses [11]. Consequently, caches and prefetchers suffer from a low hit rate, leading to many costly DRAM

accesses with high latency and energy [28]. Second, although modern computer architectures can leverage the out-of-order (OoO) execution mechanism to exploit parallelism, the memory bound of *Aggregation* phase makes OoO in vain and the deterministic execution of *Combination* phase eliminates the demand for OoO. Furthermore, due to the parameter sharing between vertices in the *Combination* phase, the data copy and synchronization between threads increase the wait time. At last, besides these intra-phase inefficiencies, the two phases are currently performed in serial, which further degrades the overall performance.

Facing the above challenges on conventional general processors and inspired by the great success of domain-specific accelerators for graph processing [16] and neural networks [9, 21], we propose the concept of GCN accelerator and implement it using a hybrid architecture. We design *HyGCN* with two efficient processing engines, i.e. *Aggregation Engine* and *Combination Engine*, to accelerate the *Aggregation* and *Combination* phases, respectively. The *Aggregation Engine* provides an efficient data-aware scheduling to perform the dynamic computation and irregular access; while the *Combination Engine* maximizes the parallelism and energy efficiency to perform the regular but intensive MVMs. On the basis of individual optimizations of these two phases with different patterns, we further optimize the overall system via the execution pipeline and memory access coordination.

Specifically, we divide our design into several steps. First, we propose a programming model to enhance the system programmability by abstracting GCNs as edge-centric aggregation for the *Aggregation* phase and MVMs for the *Combination* phase. Second, we design and optimize the two processing engines. We introduce a data-aware access scheduling to address the irregularity in *Aggregation Engine*, and use efficient systolic arrays to perform the intensive computations in *Combination Engine*. Third, we propose mechanisms of fine-grained pipeline and memory access coordination to improve the overall execution and off-chip data access, efficiently.
To summarize, we list our contributions as follows:

- We quantitatively characterize the GCN performance on the conventional general-purpose processor, and then identify the different execution patterns of the *Aggregation* phase and *Combination* phase.
- We propose a programming model for GCNs and design an accelerator, *HyGCN*, using a hybrid architecture with *Aggregation Engine* and *Combination Engine*. In the former engine, we introduce a data-aware scheduling to optimize the irregular memory access; in the latter engine, we use multigranular systolic arrays to perform the intensive MVMs efficiently.
- We coordinate the two engines and optimize the overall performance by proposing flexible inter-engine pipelines and an efficient memory access coordination.
- We implement our architecture design in RTL and evaluate it using a detailed microarchitectural simulation. We use four well-known GCN models on five popular graph datasets. Compared with the state-of-the-art *PyTorch Geometric* [14] running on Intel Xeon CPU, our work achieves three orders of magnitude speedup and three orders of magnitude less energy on average.

2. KNOWLEDGE OF GRAPH CONVOLUTIONAL NETWORKS

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, 34, 41]. Fig. 1 illustrates the execution phases of GCN models. After k iterations of aggregation via the **Aggregate** function and transformation via the **Combine** functions, a vertex is represented by its final feature vector, which captures the structural information within the vertex's k-hop neighborhood. Table 1 further gives the notations used in GCNs. In this work, we mainly focus on undirected graphs and the inference stage rather than training.

Table 1: Notations of GCNs.

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

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

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 [7, 18] as the new neighbors, specifically,

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

Sometimes, the **Pool** function [38] is inserted after the **Combination** function to transform the original graph into a smaller graph.

After several iterations, the graph 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 the entire graph's representation vector, i.e.

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

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

GCN is one of the most successful convolutional networks for graph learning [24, 33], which bridges the gap between spectral-based convolutions and spatial-based convolutions. Its inference model can be described as

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

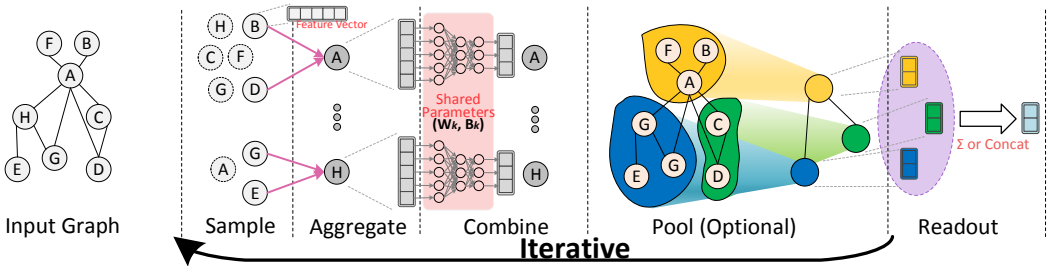

Figure 1: Illustration of the GCN model.

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 [34]. 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 by given 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 [38]. It can be inserted after the **Combination** 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

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

where A is the adjacent matrix of the graph, X is the matrix format of all feature vectors in the graph, C and Z are two intermediate matrices termed as assignment matrix and embedding matrix, respectively. 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 smaller graph. Actually in the new graph, GCN_{pool}^k determines the number of vertices, and $GCN_{embedding}^k$ determines the length of vertex feature vector.

Summary. In the above paragraphs, we introduce several typical operations in GCNs: *Sampling*, *Aggregation*, *Combination*, *Pooling*, and *Readout*. These operations can be divided into two categories depending on whether it involves graph processing or not. The graph structure-dependent operations include *Sampling*, *Aggregation*, *Pooling*, and *Readout*, while *Combination* is graph independent. *Sampling* is used to sample a subset from neighbors, which can be done during preprocessing [20] or with random selection during run-time [18]. *Aggregation* aggregates the features from the 1-hop neighbors, and *Combination* usually is a typical MLP neural network (single layer or multiple layers). *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 [14] across vertices or further with a concatenation across iterations [34]. Therefore, *Readout* can be viewed as an extreme *Aggregation*. In terms of the complexity, *Aggregation* and *Combination* are two major phases in GCNs, which is the design focus in our work.

3. CHARACTERIZATION & MOTIVATION

3.1 Characterization on General Processor

In order to identify the bottleneck when performing GCNs, we conduct quantitative characterizations using **GCN** [24] model and COLLAB [22] dataset. The programming tool is *PyTorch Geometric*(PyG) [14], and the execution platform is Intel Xeon CPU. Table 2 presents the profiling results which will be discussed from both the memory and compute aspects.

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
Number of Threads	1	24
Ratio of Synchronization Time	—	36%

Memory Access and Cache Utilization. From the first two rows of Table 2, it is observed that each operation in the *Aggregation* phase requires much more data to be accessed from DRAM, resulting in higher DRAM access energy. The underlying reason for such DRAM-dominant phenomenon is due to low cache utilization and ineffectual accesses. On one hand, the misses per kilo-instruction (MPKI) for L2 and L3 caches in the the *Aggregation* phase are higher than 10 that is very high. This is caused by the high randomness and poor locality of neighbor indices for each vertex. Thus, most accesses jump down to DRAM with more cost. On the other hand, the indirect and irregular accesses make the data prefetching in the *Aggregation* phase blind, since it is difficult to predict the data addresses without knowing the indices of neighbors in advance. This results in abundant ineffectual

memory accesses to prefetch data. Besides, the complex messaging mechanism for graph *Aggregation* usually produces many intermediate data, which also increases the number of invalid accesses. Overall, these results are consistent with the model behaviors: the *Aggregation* phase performs dynamic and irregular execution pattern while the *Combination* phase is static and regular.

Parallelism Exploitation. Due to the severe irregularity of *Aggregation* phase, CPU usually uses only single thread to process the computations. Otherwise, the use of multi-threads will suffer from heavy overheads for frequent thread creation and imbalanced workloads, leading to even lower performance than single thread. In contrast, the *Combination* phase can be parallelized using multi-threads due to the more regular execution pattern. Nevertheless, since the parameters are greatly shared between vertices, we observe abundant waits for data copy and synchronization between threads. Besides, although the OoO technique is widely used in modern high-performance processors to enhance the parallelism, it fails in the scenario of GCNs. The *Aggregation* phase is too irregular and dynamic, and the computations are frequently bounded by indirect memory accesses, which invalidates the powerful OoO. Distinct from the *Aggregation* phase, the execution pattern of *Combination* phase is too regular and deterministic, which eliminates the need for OoO.

3.2 Need for GCN Accelerator

Given the above characterizations, we observe huge challenges when performing GCNs on conventional general-purpose processors. In this subsection, we explain the motivation of designing a GCN accelerator.

Table 3: Different execution pattern 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

Variable Design Requirement. We summarize the execution patterns of the two GCN phases in Table 3, according to the profiling results in Table 2. The *Aggregation* phase demands more efforts to orchestrate the memory access that bounds the overall performance. Differently, the *Combination* phase needs more attention to improve the intense computations with better parallelism and faster synchronization. Facing such opposed design requirements, it is even challenging to achieve high performance in a single specialized system, let alone on the bloated general-purpose processors.

Lack of Inter-phase Optimization. The conventional processors equipped with current programming framework for GCNs usually adopt coarse-grained set of functions and instructions, which results in phase-by-phase execution. This serialization compromises the design space with phase interaction, hindering the further performance improvement beyond the individual optimization for each phase.

Opportunities for Customization. Designing a specialized accelerator for a specific application domain is an efficient and prevalent solution to address the inefficiency on traditional architectures, since it can tailor the computation unit and memory hierarchy to adapt with the special type of workload. Here in the context of our work, we can build the 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 then schedule the accesses towards higher rate of data reuse and less redundant accesses; for the *Combination* phase, we draw inspirations from current neural network accelerators to efficiently perform MVMs with parameter sharing. Beyond the individual optimizations, the off-chip memory accesses from the two phases is controllable to improve the overall memory efficiency. Without the restrictions on general processors, now the serial inter-phase dataflow can be pipelined in fine grain. Putting all these together, there are huge opportunities to design an efficient GCN acceleration with high performance.

4. ARCHITECTURE DESIGN

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

4.1 Edge- and MVM-Centric PM

The goal of building a programming model (PM) is to achieve the hardware transparency for programmers without compromising the execution performance [41]. 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 all the computation of all vertices, the degree of parallelism will be degraded. 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 will lead to intensive memory access and vertex computation. To address this problem, we employ an edge-centric PM to exploit the edge-level parallelism. In this way, the workload for each vertex can be divided into subworkloads and assigned to each computation unit for parallel processing. For *Combination*, the situation is relatively easier. Since the computation of each vertex acts like the MLP neural network, 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 first read, 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 by the entire two engines, the matrix transposes can be done by the flexible *Aggregation* engine, and the matrix multiplications can be executed by the *Combina-*

tion 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, Figure 2 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 allowed.

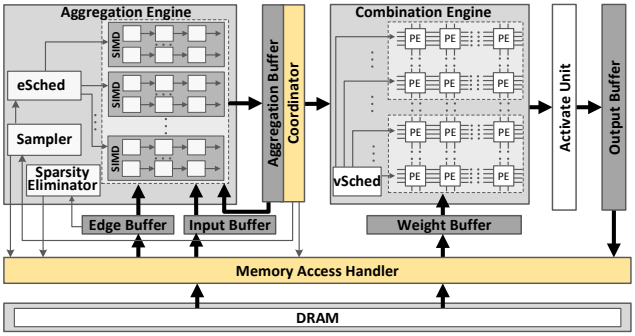

Figure 2: Architecture overview of *HyGCN*.

The *Aggregation Engine* aims to realize the efficient execution of random 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 sample operation, we introduce a *Sampler* into the *Aggregation Engine*. The *Sampler* selects edges from the edge list of each vertex using a uniform or pre-defined distribution in terms of index interval. The former indices for edge sampling are based on dynamical generation (see Fig. 3(a)) while the latter ones are predefined and can be read from off-chip memory like in [7, 20]. To reduce the latency of data access, we employ embedded DRAM (eDRAM) to cache various data for better rate of data reuse and locality. An *Edge Buffer* is used to cache edges for the exploitation of 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 the temporal locality. To hide the off-chip 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 neighbor vertices that do not connect to the aggregating vertex.

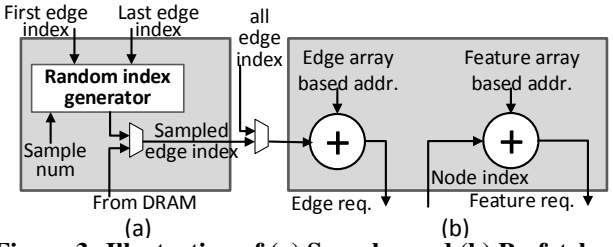

Figure 3: Illustration of (a) Sampler and (b) Prefetcher.

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 [21] 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 finally 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, we add a *Prefetcher* to explicitly prefetch graph data and weight data. For example, Fig. 3(b) illustrates the feature vector prefetching. *Prefetcher* can not only use the sampled edge indices from *Sampler* to prefetch the requested edges, but also prefetch the entire edge array. After receiving the prefetched edge $e(u,v)$, the neighbor vertex index can be used to prefetch the feature vector of that vertex immediately.

4.3 Aggregation Engine

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

4.3.1 Execution Mode

As shown in Fig. 4, the compute units in SIMD cores process edges in parallel. Usually, there are two processing modes. 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 a burst way, i.e. periodically processing a group of vertices. However, the processing latency of a single vertex is long, and the fast vertices have to wait for the slow vertices leading to workload imbalance. Therefore, we use the second one shown in Fig. 4 which assigns the workloads of each vertex

to all SIMD cores, termed as vertex-disperse mode. Although the total workloads do not change, the vertex-disperse mode generates the aggregated features vertex by vertex with low latency and less workload imbalance between SIMD cores, which enables the immediate processing of each vertex in the following *Combination Engine*.

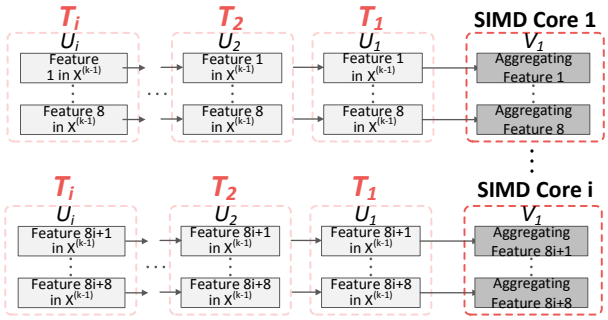

Figure 4: Vertex-disperse processing mode where the workloads of each vertex are assigned to all SIMD cores.

4.3.2 Graph Partition (Static)

We borrow the abstraction of vertex interval and edge shard from [10,25] 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.

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
```

The feature vector length of each vertex is usually large so that to exploit the locality of feature 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, each edge shard is usually 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.

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.

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 is shrinked. 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
```

Algorithm 4: ***GetOneEffectInterval (Get One Effectual Neighbor Interval)***

```
                ◁ 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;
```

Given the effectual shards after sparsity elimination, the

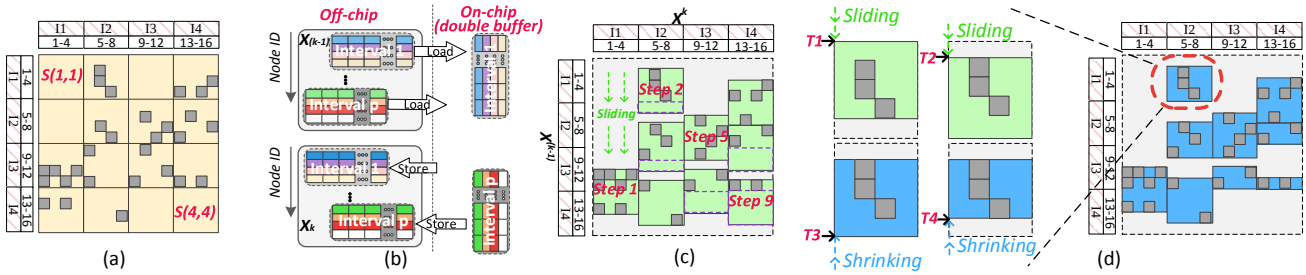

Figure 5: Static graph partition for data reuse and dynamic sparsity elimination to reduce redundant accesses: (a) interval-shard partition; (b) interval-wise feature access; (c) window sliding; (d) window shrinking.

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 remained neighbor vertices when performing the aggregation operation for each interval I_i are loaded, which eliminates plenty of redundant accesses.

4.3.4 *Difference from Graph Analytics*

The feature data in traditional graph analytics are small, usually one element for each vertex. By contrast, the feature of each vertex in GCNs is a vector with even thousands of elements. Thus, the feature data reuse from graph partition and redundant access reduction from sparsity elimination are considerable. When the sample operation is used in GCNs, the sparsity will be increased much 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 intense. Our design is based on the well-known systolic array. In order 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.

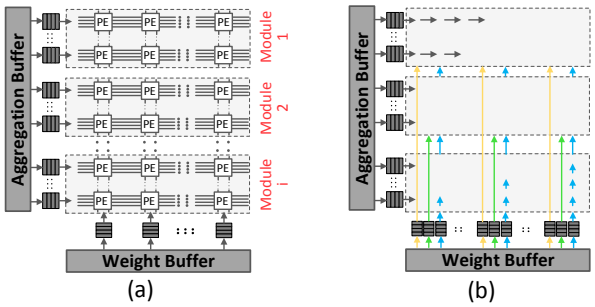

Figure 6: *Combination Engine* design: (a) multiple systolic modules; (b) different dataflow patterns.

4.4.1 *Independent Working Mode*

In this mode, the systolic modules work independently from each other. Each of them processes the MVM operations 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. The independent mode matches well with the vertex-disperse processing mode of *Aggregation Engine* in Fig. 4, where the aggregated features are produced quickly but sequentially.

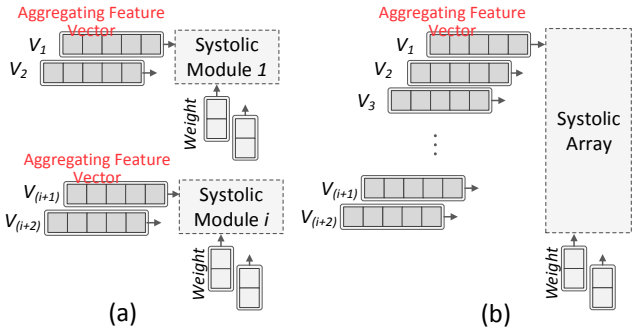

Figure 7: Different use of the systolic arrays: (a) independent working mode; (b) cooperative working mode.

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.

4.4.3 *Difference from Neural Networks*

In neural networks especially MLP, the weights cannot be shared without batching technique. By contrast, the weights are fully shared by different vertices in GCNs. No matter which working mode is selected in the *Combination Engine*, the weights can be reused in *Weight Buffer* when processing different vertices. In addition, the multigranular systolic array design is also special in our architecture in order to accommodate different application needs.

4.5 Inter-Engine Optimization

In this subsection, we orchestrate the execution pipeline and DRAM access of *Aggregation* engine and *Combination* engine, by the *Coordinator* module shown in Fig. 2.

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 fashions of pipeline: latency-aware pipeline and energy-aware pipeline.

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.

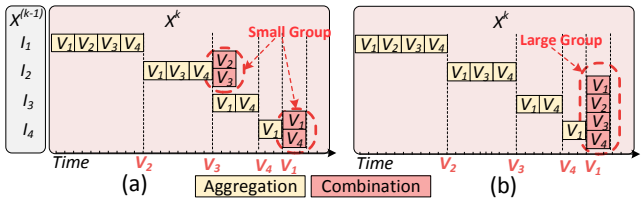

Figure 8: Timing illustration of different pipeline modes: (a) latency-aware pipeline; (b) energy-aware pipeline.

Energy-aware Pipeline. By contrast with the latency-aware pipeline, the energy-aware pipeline here uses the systolic module cooperative mode in the *Combination Engine*. The vertex-by-vertex processing changes to a burst fashion, 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.

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. Both the two engines access this memory at runtime, which causes a frequent switching of access locations, leading to inefficiencies.

Fig. 9 shows our solution. 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 concern to use this priority is based on the access sequence when processing a vertex. With the improved continuity, the utilization of row buffer can be significantly enhanced. Then, 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.

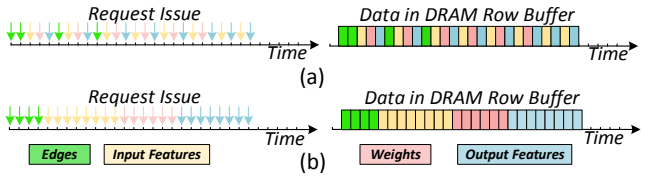

Figure 9: Coordination of off-chip memory access.

5. EVALUATION RESULTS

We first describe our experimental setup in Section 5.1. Then, to demonstrate the advantages of our design, we compare *HyGCN* to the state-of-the-art software framework *PyTorch Geometric*(PyG) in Section 5.2. Next, we give the detail analysis of our optimization techniques in Section 5.3. Finally, we present a scalability exploration to show the trade-off 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 customized cycle-accurate simulator to measure execution time in number of cycles. This simulator models the microarchitectural behaviors of each module in our architecture design. In addition, we implement a detailed cycle-accurate scratchpad memory model. It is integrated with Ramulator [23] to simulate the behaviors of memory accesses to DDR4.

CAD Tools. For the measurements of area, power, and critical path delay (in cycles) for each module, we implement each module with Verilog, and then synthesize them. 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.

eDRAM 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 [29, 32].

Table 4: System configurations.

	HyGCN	CPU (PyG)
Compute Unit	1 GHz @ 32 SIMD16 cores and 8 systolic modules (each with 4×128 arrays)	2.5 GHz @ 24 cores
On-chip Memory	128 KB (Input), 2 MB (Edge), 2 MB (Weight), 4 MB (Output) and 16 MB (Aggregation)	128 KB L1, 512 KB L2, and 60 MB L3
Off-chip Memory	136.5 GB/s (8 channels 2133 MHz DDR4)	

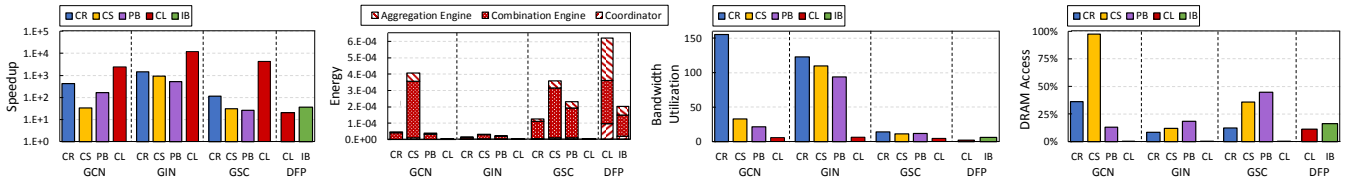

Figure 10: Comparison to PyG. (a) speedup, (b) energy, (c) bandwidth utilization, and (d) DRAM access.

Baseline Platform. To compare the performance and energy efficiency of *HyGCN* with state-of-the-art works, we evaluate *PyTorch Geometric*(PyG) [14] in a Linux workstation equipped with two Intel Xeon E5-2680 v3 CPUs, 400 GB DDR4, and 136.5 GB/s memory bandwidth. Table 4 shows the system configurations for above implementations.

Table 5: Datasets information [22, 36].

Dataset	#Graph	#Vertex	Feature Len.	#Edge
Cora (CR)	1	2,708	1,433	5,429
Citeseer (CS)	1	3,327	3,703	4,732
Pubmed (PB)	1	19,717	500	44,338
Dataset	**#Graph**	**Avg. #Vertex**	**Feature Len.**	**Avg. #Edge**
IMDB-BIN (IB)	1000	19.8	136	96.53
COLLAB (CL)	5,000	74.49	492	2,457.78

Table 6: 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	Mean & $\|a_v^k\|$–128
GINConv (GIN)	—	Add & $\|a_v^k\|$–128–128
DiffPool (DFP)	**GCN**$_{pool}$	**GCN**$_{embedding}$
	Mean & $\|a_v^k\|$–128	Mean & $\|a_v^k\|$–128

Benchmark Graph Datasets and GCNs. Table 5 and Table 6 provide the information of the benchmark graph datasets and GCN models used in our evaluation. 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 fashions remain the same with CPU except that the selected graphs for DFP are processed one by one rather than in a batching way. The latency is the execution time of a single graph on CR, CS, and PB or the assembled or batched graphs on other datasets.

5.2 Overall Results

In this section, we compare our work (*HyGCN*) with PyG on CPU in terms of speedup, energy consumption, utilization of DRAM bandwidth, and DRAM access. Besides, we show the area and power of our design.

5.2.1 Comparison to Software Framework

• ***Speedup.*** As shown in Fig. 10(a), *HyGCN* achieves 21∼12000× speedup compared with PyG on CPU. 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 make the computations fast. Second, the graph partition and sparsity elimination increase the feature reuse and decrease redundant accesses in *Aggregation Engine*, which save the DRAM bandwidth. Third, the weight parameters are well reused in *Combination Engine*, which also helps to better utilize the bandwidth. At last, the inter-engine pipeline further optimizes the parallelism and the off-chip memory access coordination improves the DRAM access efficiency.

While for PyG on CPU, abundant DRAM accesses and synchronization overheads lead to performance degradation. Specifically, the high randomness of neighbor indices results in poor locality, causing many DRAM accesses. Besides, the complicated messaging mechanism for graph aggregation usually produces many intermediate data that occupy extra DRAM bandwidth. From the perspective of computation, PyG performs GCNs in a coarse-grained fashion, which significantly loses the parallelism and produces redundant operations. The wait for data copy and synchronization between threads further degrades the performance.

Next, we discuss the performance variation between datasets. Since the CL dataset presents a high-degree distribution, we can achieve higher speedup via higher ratio of data reuse. By contrast, other datasets have higher sparsity than CL, thus show lower ratio of data reuse. Although the sparsity elimination can remove many redundant accesses, these datasets are still memory-bounded and their speedup is linearly correlated to the utilization of memory bandwidth.

In term of models, GIN achieves better performance than others. The underlying reason is that GIN executes the aggregation first on PyG, which introduces abundant computations and accesses since the feature vector size is a magnitude order larger than that after the combination. By contrast, other models execute the combination first, which greatly reduces the feature length before performing the aggregation. This difference causes the inefficient execution of GIN on CPU, while our *HyGCN* can maintain the performance to a great extent thanks to the parallel processing and data reuse. For DFP, it includes three matrix multiplications (see Equation (8)) that can be efficiently executed on CPU. There, our speedup when performing DFP is lower than running others.

• ***Energy Consumption.*** For CPU measurement, we use Intel PCM [4] to capture the energy consumption of cores and the DRAM controller. Note that our energy consumption does not include the DRAM controller since it could be placed off-die. As shown in Fig. 10(b), *HyGCN* consumes only 0.06%-0.0003% energy compared with PyG on CPU. Among the architectural components, *Combination Engine* consumes most of the energy due to the intensive computation of MVMs. As aforementioned, GIN causes more computations and data accesses when performing the aggregation, which introduces extra energy consumption on PyG. Although *HyGCN* cannot reduce these computations, the optimizations of data reuse, sparsity elimination, and inter-engine pipeline can reduce redundant accesses to these extra data.

• ***DRAM Bandwidth Utilization.*** As shown in Fig. 10(c), *HyGCN* demonstrates 2∼150× improvement on the utilization of DRAM bandwidth compared with PyG. The high bandwidth utilization of *HyGCN* derives from the high-degree parallelism. By contrast, PyG cannot sufficiently exploit the bandwidth, since there is only one thread in most of time to

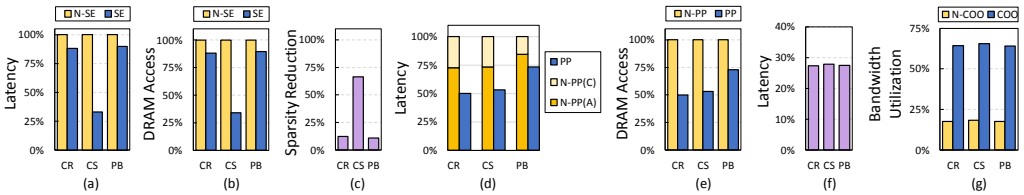

Figure 11: Optimization analysis: i) effect of sparsity elimination on (a) latency, (b) DRAM access, and (c) sparsity reduction; ii) effect of inter-engine pipeline on (d) latency and (e) DRAM access; iii) effect of off-chip memory coordination on (f) latency and (g) DRAM bandwidth utilization.

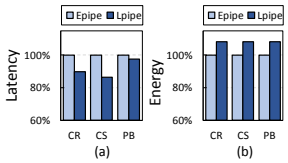

Figure 12: Comparison of (a) latency and (b) energy of *Combination Engine* under different pipeline modes.

reduce the heavy overheads of frequent thread creation. Our consistent lower bandwidth on CL dataset is due to the higher data reuse, which benefits from denser connections.

• ***DRAM Access.*** As shown in Fig. 10(d), although the 16 MB on-chip memory is much smaller than the 60 MB L3 cache on CPU, *HyGCN* accesses only 30% of off-chip data compared with PyG on average. This benefits from our data reuse optimizations, sparsity elimination, and the immediate processing between two engines. On 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 avoid unnecessary data accesses. Whereas, PyG is not aware of the sparsity, producing abundant accesses that are introduced by blind prefetch.

5.2.2 Power and Area

For the computation precision, we use 32-bit fixed point that is enough to maintain the accuracy of GCN inference. We construct *Aggregation Egine* using adders and shifters while construct *Combination Engine* using multiplier-and-accumulators, adders, and comparators. For the on-chip buffer, we use eDRAM to reduce both the area and energy consumption. The total power and area of *HyGCN* are only 6.7 W and 7.8 mm^2, respectively.

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* (1 GHz, 6.7 W and 7.8 mm^2), implemented in TSMC 12 nm technology.

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 6 and the results are shown in Fig. 11-12.

5.3.1 Sparsity Elimination Optimization

We conduct an experiment to evaluate *HyGCN* with and without sparsity elimination (SE v.s. N-SE). This experiment runs only *Aggregation Engine* to avoid the interference of other blocks. As depicted in Fig. 11(a), with the optimization of sparsity elimination, *HyGCN* achieves 1.1∼3× speedup. The performance improvement owes to the much less redundant DRAM accesses, which is reflected in Fig. 11(b). As shown in Fig. 11(c), the eliminated sparsity reaches 66% .

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%-50%, as shown in Fig. 11(d). 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. 11(d)), therefore the inter-engine pipeline helps improving the performance by decreasing DRAM accesses of the intermediate aggregation results between two engines. It is observed from Fig. 11(e) that total DRAM accesses are significantly reduced to only 50%-73% with this pipeline optimization.

Second, we compare the latency and energy of *Combination Engine* with energy-aware pipeline and latency-aware pipeline (Epipe v.s. Lpipe). From Fig. 12(a), it is obvious that the Lpipe reduces the average execution time for each vertex by 3%-14% via the immediate processing without waiting for the aggregation results of too many vertices. By contrast, as shown in Fig. 12(b), the Epipe saves the energy consumption by 8% via assembling a large group of vertices to process together for reusing weight parameters aggressively. In practice, the application requirement determines the pipeline fashion.

5.3.3 Memory Coordination Optimization

To show the effect of the memory access coordination, we present the execution latency and bandwidth utilization with and without coordination (COO v.s. N-COO) in Fig. 11(f) and Fig. 11(g), 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. In this way, it is able to save 72% of execution time and increase 3.6× DRAM bandwidth on average.

5.4 Scalability Exploration

5.4.1 Sparsity Elimination with Sampling

The sample operation in GCNs is helpful to increase the sparsity, thus has potential to enlarge the benefits produced by sparsity elimination. In Fig. 13(a)-(c), we measure the

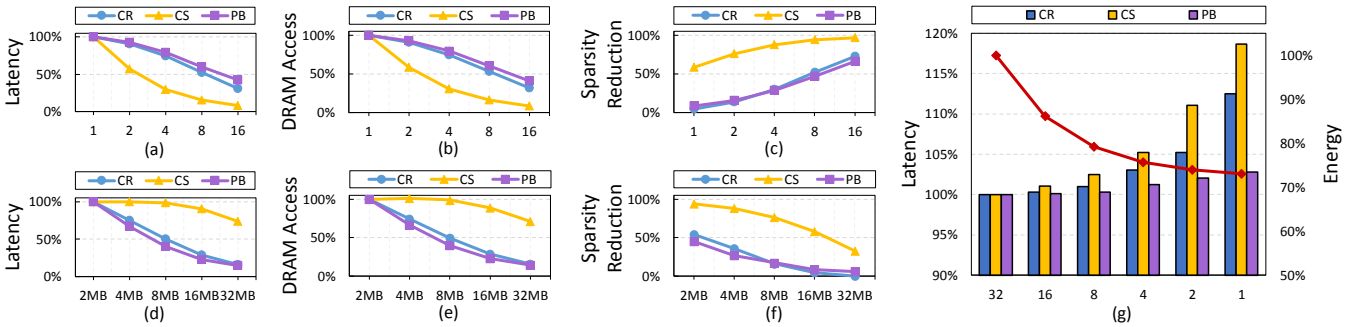

Figure 13: Scalability exploration: i) sparsity elimination with different sampling factor affecting the (a) latency, (b) DRAM access, and (c) sparsity reduction; ii) capacity of aggregation buffer affecting the (d) latency, (e) DRAM access, and (f) sparsity reduction; iii) size of systolic module affecting the (g) latency and energy of *Combination Engine*.

performance when running GSC model with variable sampling ratio. The number in horizontal axis is the *sampling factor*, which indicates that only $\frac{1}{sampling\,factor}$ edges of each vertex are sampled to perform aggregation. It is observed that the increasing *sampling factor* significantly improves the latency by reducing the DRAM accesses owing to the higher sparsity. Note that the *sampling factor* cannot be too high, which might harm accuracy of the application.

5.4.2 Capacity of Aggregation Buffer

The size of the *Aggregation Buffer* will affect the latency, 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 latency of running GCN model is decreased as shown in Fig. 13(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. 13(e)). However, larger shard also enlarges the window size during the sparsity elimination, which results in higher sparsity that cannot be eliminated (see Fig. 13(f)). Note that in this experiment, the *Edge Buffer* should also increase with the *Aggregation Buffer* to match the enlarged edge shard.

5.4.3 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 4, here we treat 1×128 systolic arrays as a basic systolic module. Based on the initial 32 systolic modules, we gradually increase the number of arrays in each systolic module, i.e. decreasing the number of systolic modules under the restriction of fixed number of total systolic arrays. The latency-aware pipeline technique and GCN model are used in this experiment. It is observed that longer execution time is consumed as the partition of systolic modules becomes more coarse-grained as shown Fig. 13(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. 13(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. RELATED WORK

GCN Software Frameworks. There are a plenty *GCN* software frameworks presented to release the programming efforts while achieve high performance in modern architectures [2, 14, 27, 37, 41]. Unfortunately, the distinct pattern of computation and access between the *Aggregation* phase and *Combination* phase produces processing inefficiencies on traditional platforms. GCNs call for specialized architecture design and optimizations.

Accelerators for Graph Analytics and Neural Networks. With the emerging of graph analytics and neural networks for real-world applications, a lot of hardware architecture designs are proposed to accelerate these workloads [8, 9, 16, 21, 29]. However, GCNs behave like not only the graph processing (*Aggregation* phase) but also neural networks (*Combination* phase), which require different design requirements. Therefore, current specialized architectures cannot efficiently perform GCNs since they focus on only one side.

7. CONCLUSION

Recently, *GCNs* are widely adopted to analyze graph datasets using neural networks, which mainly include two execution phases: *Aggregation* phase and *Combination* phase. In this work, we identify that the computation and access patterns of these two phases are distinct, even almost opposed, which requires inconsistent design requirements. To this end, we propose the concept of GCN accelerator and implement it using a hybrid architecture. First, we build Edge- and MVM-centric programming models for the two phases respectively to achieve hardware transparency. Then, we design *HyGCN* architecture with efficient *Aggregation Engine* and *Combination Engine* to optimize the two execution phases correspondingly. The latency- and energy-aware inter-engine pipelines are orchestrated to improve the overall latency and energy according to actual needs. The off-chip memory accesses between the two engines are carefully coordinated to improve the efficiency. At last, through extensive evaluation experiments, *HyGCN* demonstrates significant improvements compared with current software frameworks running on CPUs. The analysis of optimization techniques and exploration of design space are also provided to present our design insights. Our work will stimulate more researches on specialized hardware for increasingly important GCNs.

8. 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] Intel performance counter monitor - a better way to measure CPU utilization | intel software. [Online]. Available: https://software.intel.com/en-us/articles/intel-performance-counter-monitor

[5] Knowledge inside search–google. [Online]. Available: https://www.google.com/intl/en_us/insidesearch/features/search/knowledge.html

[6] P. Battaglia, R. Pascanu, M. Lai, D. J. Rezende, and K. kavukcuoglu, "Interaction networks for learning about objects, relations and physics," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. USA: Curran Associates Inc., 2016, pp. 4509–4517. [Online]. Available: http://dl.acm.org/citation.cfm?id=3157382.3157601

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

[8] 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. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541967

[9] 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.

[10] 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.

[11] 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. [Online]. Available: http://ieeexplore.ieee.org/document/6468514/

[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] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969442.2969488

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

[15] 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. [Online]. Available: http://papers.nips.cc/paper/7231-protein-interface-prediction-using-graph-convolutional-networks.pdf

[16] 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.

[17] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto, "Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, pp. 1802–1808. [Online]. Available: http://dl.acm.org/citation.cfm?id=3172077.3172138

[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. [Online]. Available: http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs.pdf

[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. [Online]. Available: http://papers.nips.cc/paper/7707-adaptive-sampling-towards-fast-graph-representation-learning.pdf

[21] 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. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[22] 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

[23] 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. [Online]. Available: https://doi.org/10.1109/LCA.2015.2414456

[24] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016. [Online]. Available: http://arxiv.org/abs/1609.02907

[25] 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. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387884

[26] Y. LeCun, "1.1 deep learning hardware: Past, present, and future," in *2019 IEEE International Solid-State Circuits Conference-(ISSCC)*. IEEE, 2019, pp. 12–19.

[27] 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. [Online]. Available: http://arxiv.org/abs/1903.12287

[28] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, "Enabling practical processing in and near memory for data-intensive computing," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: ACM, 2019, pp. 21:1–21:4. [Online]. Available: http://doi.acm.org/10.1145/3316781.3323476

[29] 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.

[30] B. Perozzi, R. Al-Rfou, and S. Skiena, "DeepWalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. ACM, pp. 701–710, event-place: New York, New York, USA. [Online]. Available: http://doi.acm.org/10.1145/2623330.2623732

[31] R. Trivedi, H. Dai, Y. Wang, and L. Song, "Know-evolve: Deep temporal reasoning for dynamic knowledge graphs," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, pp. 3462–3471. [Online]. Available: http://dl.acm.org/citation.cfm?id=3305890.3306039

[32] 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.

[33] 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. [Online]. Available: http://arxiv.org/abs/1901.00596

[34] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *CoRR*, vol. abs/1810.00826, 2018. [Online]. Available: http://arxiv.org/abs/1810.00826

[35] 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. [Online]. Available: http://arxiv.org/abs/1806.03536

[36] 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. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2783258.2783417

[37] 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. [Online]. Available: http://doi.acm.org/10.1145/3219819.3219890

[38] 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. [Online]. Available: http://papers.nips.cc/paper/7729-hierarchical-graph-representation-learning-with-differentiable-pooling.pdf

[39] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *CoRR*, vol. abs/1812.04202, 2018. [Online]. Available: http://arxiv.org/abs/1812.04202

[40] 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. [Online]. Available: http://arxiv.org/abs/1812.08434

[41] 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. [Online]. Available: http://arxiv.org/abs/1902.08730