2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO) | 979-8-3503-5057-9/24/$31.00 ©2024 IEEE | DOI: 10.1109/MICRO61859.2024.00050

SCALE: A Structure-Centric Accelerator for Message Passing Graph Neural Networks

Lingxiang Yin[*], Sanjay Gandham[*], Mingjie Lin, Hao Zheng[‡]
Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL, USA
{lingxiang.yin, sanjay.gandham, milin, hao.zheng}@ucf.edu

***Abstract*—Message passing paradigm has been widely used in developing complex Graph Neural Network (GNN) models, allowing for concise representations of edge and vertex-wise operations. Despite its pivotal role in theoretical advancement, the respective expression of edge and vertex operations, along with evolving GNN variants and datasets, has inevitably led to enormous computational complexity due to heterogeneous computation kernels. In particular, such inconsistent computation characteristics present new challenges in leveraging intermediate data reuse, ensuring both edge and vertex-wise workload balance, and sustaining system scalability.**

In this paper, we propose a structure-centric accelerator, SCALE, that can support a variety of message passing GNN models with improved parallelism, data reuse, and scalability. The central idea is to find latent similarities among GNN primitives such as shared dataflow structure, rather than strictly adhering to heterogeneous model structure. This serves as a hinge to homogenize inconsistencies in various GNN computation kernels. To accomplish this concept, SCALE consists of three unique designs, a novel systolic array-like architecture, a degree and vertex-aware scheduling, and a coherent dataflow tailored for fused graph and neural operations. The proposed systolic array-like architecture can support varying dataflows such as all-reduce, of distinct GNN operations improving parallelism, data reuse, and throughput. The degree and vertex-aware scheduling can remedy the workload imbalance encountered in vertex and edge-wise operations. Moreover, the proposed dataflow can unify the data movement of both graph and neural operators without extra communication and storage overheads. Our simulation results show that SCALE achieves 1.82× speedup and 38.9% energy reduction on average over the state-of-the-art GNN accelerators [1]–[4].

I. Introduction

Graph Neural Networks (GNNs) have emerged as a promising model to comprehend complex graph-structured data, showing satisfactory performance in various applications such as social networks [5]–[7], recommendation systems [8]–[10], and bioinformatics [11]–[14]. In light of recent theoretical advancements, GNN models now incorporate complex edge and vertex-wise operations to capture nuanced graph structures and semantics. Such complex operations are typically expressed using the message passing paradigm in deep learning libraries such as Deep Graph Library and Pytorch Geometric. Yet, despite its importance, message passing GNN poses escalating computational and communication challenges on the hardware due to intricate vertex and edge-wise operations.

Even though significant efforts [15]–[21] have been devoted to facilitating Graph Convolutional Networks (GCN), they are inefficient in handling message passing GNNs with

[*]Equal Contribution. [‡]Corresponding Author.

Table I: Comparisons of SCALE with other state-of-the-art accelerators.

Accelerator	Message Passing	Commu. Latency	Unified Dataflow	Data Reuse	Workload Balance	
					Aggr.	Update
AWB-GCN [2]	✗	Medium	◐^{1}	Low	◐^{1}	◐^{1}
GCNAX [1]	✗	High	◐^{1}	Medium	◐^{1}	◐^{1}
I-GCN [22]	✗	High	◐^{2}	Medium	◐^{2}	◐^{2}
ReGNN [4]	◐^{3}	Medium	✗	Medium	✗	✓
FlowGNN [3]	✓	High	✗	Low	✗	✓
SCALE	✓	Low	✓	High	✓	✓

◐^{1}: Optimized for sparse-dense matrix multiplications; ◐^{2}: Optimized for dense-dense matrix multiplications; ◐^{3}: No edge embedding support.

explicit edge operations as shown in Table I. Specifically, GCN computations can be simply accelerated in the form of sparse-dense matrix multiplication (SpMM) and general matrix multiplication (GEMM). Prior works, such as AWB-GCN [2] and I-GCN [22], either relied on runtime workload distribution or matrix preprocessing to overcome the sparsity issue involved in SpMMs. GCNAX [1] utilized loop fusion and reordering to optimize SpMM kernels, thus reducing excessive off-chip memory access. However, emerging GNN models like Graph Attention Networks (GAT) [23] involve complex attention score calculation over edges represented by sampled dense-dense matrix multiplication (SDDMM) [24]. Higher-Order Graph Convolutional Architectures [25] require information aggregation from non-adjacent vertices. Consequently, such intricate graph operations make it difficult to expedite GNN computations through graph reordering and SpMM optimizations.

To support message passing GNNs, FlowGNN [3] introduced a dataflow architecture to support edge and vertex operations in a pipeline fashion. These computation units are connected by a complex interconnection network to withstand erratic communication, leading to scalability concerns. Similarly, ReGNN [4] presented a dynamic redundancy-eliminated neighborhood message passing to improve graph data locality. However, its parallelism is also restricted by separate graph and neural operations with considerable communication overheads. More importantly, heterogeneous architectures emerge as a key bottleneck hindering the data locality of intermediate results and scalability.

Considering constantly evolving GNN models, we posit that adhering to their model structure is the primary issue that limits data locality and scalability, as each GNN operation exhibits distinct computation and dataflow patterns. To this end, we propose SCALE, a structure-centric accelerator that unifies the computation via a shared dataflow. The key idea is to uncover latent similarities among heterogeneous GNN kernels through exploiting dataflow structure. Tailoring dedicated dataflows to be consistent for both graph and neural operations enables their execution using a single computation fabric, thereby unifying the computation patterns. Consequently, it simplifies the communication complexity, improving intermediate data reuse, and system scalability.

The major contributions of this paper are as follows:

- **SCALE Accelerator Design:** The proposed SCALE accelerator can efficiently parallelize graph and neural operations with a coherent dataflow, thus obviating the communication complexity and storage overheads. Specifically, SCALE introduces a novel systolic array-like architecture to simultaneously fuse feature aggregation (in the form of reduce operations) and vertex update operations to eliminate graph irregularity with much-improved performance. In other words, our proposed architecture repurposes conventional systolic array architecture for the chained reduction and matrix multiplication with a coherent data movement pattern, enabling intermediate data reuse between heterogeneous operators with minimized communication distance. The proposed architecture compromises a shift-register array to meet the increased data bandwidth required by the fused operations, a novel PE architecture to directly accommodate the dependency between feature aggregation and vertex update, and a flexible ring interconnect to provide adaptive dataflow for both operations.
- **Degree and Vertex-aware Scheduling:** Unlike conventional graph processing, GNN computations require considerable execution time for both feature aggregation and vertex update. As such, both degree and vertex information should be considered for workload balance. SCALE incorporates a degree and vertex-aware scheduling policy, which can dynamically allocate equivalent edge and vertex quantity to different processing units while reforming the workload for message reduce and vertex update. This ensures the workload balance for both aggregation and update phases for GNNs.
- **Flexible Dataflow for Fused Graph and Neural Operations:** Given the dynamic variations in graph connectivity, feature size, and weight matrix, it requires a flexible dataflow to seamlessly orchestrate the data movement of the graph and neural operations. We propose a unique dataflow to fuse the inconsistent graph and neural operations to achieve unified data movement across computation units, optimizing the parallelism and data reuse.

We evaluate the proposed SCALE, and the evaluation result shows that our design achieves 1.82× speedup and 38.9% energy reduction on average over the state-of-the-art GNN accelerators [1]–[4].

II. Background and Motivation

A. Graph Neural Networks

Graph Neural Networks (GNNs) are a category of neural networks developed to handle graph-structured data. This category includes several variants, such as Graph Convolutional Networks (GCNs) [26], Gated Graph Convolutional Networks (G-GCN) [27], GraphSAGE [28], and Graph Isomorphism Networks (GINs) [29]. Given the complex and diverse graph operations, such as updating the edge and vertex embeddings, GNN computation patterns cannot be simply formulated as sparse-dense (SpMM) or general matrix (GEMM) multiplications. Instead, GNN models rely on a message passing scheme to perform such complex operations. In essence, most GNN models can likely be expressed using this message passing approach [30]–[33].

Message Passing in GNNs: Message passing in GNNs is a key mechanism to incorporate local neighborhood information and is generally structured into two main stages: the aggregation and update steps. Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, the aggregation step computes a representation for each vertex based on its neighbors' features:

$$m_v^k = \text{AGGREGATE}^k \left(\{h_u^{k-1} : u \in N(v)\}\right) \tag{1}$$

Here, m_v^k is the aggregated message for vertex v at layer k, $N(v)$ represents the neighbors of v, and h_u^{k-1} are the features of node u from the previous layer. The update step then integrates the aggregated message with its own features:

$$h_v^k = \text{UPDATE}^k \left(h_v^{k-1}, m_v^k\right) \tag{2}$$

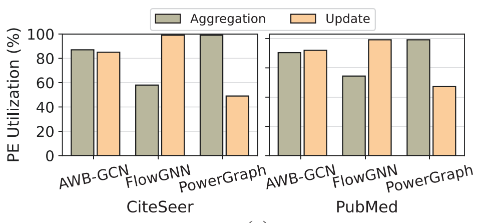
These two stages form the core of message passing and are foundational to various GNN models [34], [35]. The aggregation phase workload (AGGREGATE^k) is dependent on the degree of each vertex, while the update phase workload (UPDATE^k) relies solely on the number of vertices.

B. Motivation

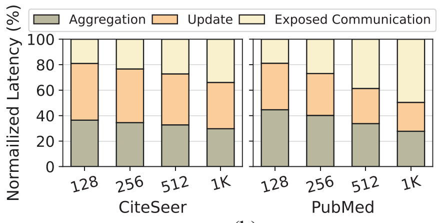
Despite recent efforts [2], [3], [36], [37], several challenges remain unsolved in accelerating message passing GNNs, limiting their parallelism, scalability, and data reuse. To fully uncover such challenges, we conduct an initial study across different message passing GNN models and graph datasets.

Structure-aware Workload Partitioning: Message passing paradigm involves extensive edge and vertex-wise operations, highly contingent on graph structure. Consequently, unlike edge or vertex-centric graph partitioning [3], [36], [38], it is imperative to ensure even distribution of edges and vertices among processing elements respectively. Based on our application profiling, as shown in Fig. 1(a), FlowGNN [3] and PowerGraph [36] present 40-50% PE under-utilization of both compute engines, in which only vertex or edge quantity is considered at the workload scheduling.

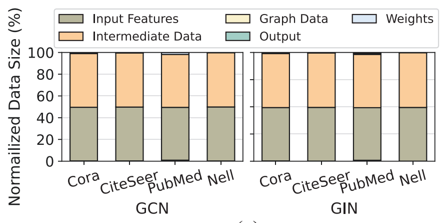
Scalability Concern due to Heterogeneous Kernels: Message passing GNN phases such as aggregation and update involve operating on irregular graph data and regular computation on tensors respectively. Given their distinct characteristics, architectures have to be tailored separately for each operation phase. These compute engines are connected by either a multi-stage or a crossbar network to handle the data communication of intermediate data. Therefore, as the network scales, the communication latency increases. For example, in Benes network [39], the hop count of each intermediate data is $2log_2N$.



(a)



(b)



(c)

Figure 1: (a) Unbalanced workload distribution in prior scheduling policies, (b) execution time breakdown in the pipeline of GNN execution, and (c) normalized size of various data types in sample GCN and GIN models.

However, the computation time for each intermediate result remains constant regardless of network size. Conventional architectures [3], [4], [40] typically employ pipelining to mask communication overheads with the computation. However, it remains difficult to efficiently overlap communication latency with computation due to their disproportionate scaling [41]. Based on our study, we observed that the communication time will not be overlapped when the PE size exceeds 128. This will eventually increase the overall execution time by 2-3× as shown in Fig. 1(b). Note that exposed communication refers to the portion of the communication latency that hinders the execution of the update phase.

Compromised Intermediate Data Reuse: In addition, reusing intermediate data is critical in GNN acceleration as illustrated in Fig. 1(c), as intermediate data is predominant (approximately 50%) in overall GNN data including graph, input, weight, and output. Theoretically, the intermediate data reuse is proportional to graph data and feature size, whereas graph data reuse is subject to mutually-shared nodes [4]. While previous study [22] emphasized data reuse optimization in SpMM kernels, it remains a significant challenge to exploit intermediate data reuse in heterogeneous computation kernels at the register level. This requires a strategic design from both dataflow and architecture aspects.

III. Proposed Architecture Design

The key idea of SCALE is to identify a shared dataflow among distinct message passing operations. As such, a coherent dataflow can be leveraged to fuse multiple operations, enabling fine-grained parallelism and increased data reuse at the register level. Given that all operations are performed upon a shared graph structure, graph structure serves as the cornerstone to unify dataflows of GNN operations.

Fig. 2 illustrates SCALE's coherent dataflow for both GNN phases. Unlike conventional dataflows in the systolic array architectures, the proposed dataflow is similar to all-reduce operation in collective communication, which consists of 'reduce-scatter' and 'all-gather' operations [42]. SCALE first reorganizes the irregular computation involved in aggregating information from neighboring vertices during the aggregation phase into a linear chain of reduce computations. This enables the aggregation engine of SCALE to perform aggregation by shifting data in the forward direction adhering to a regular communication pattern, corresponding to the 'reduce-scatter'. The results of the aggregated vertices are then directly processed by the update engines while communicating

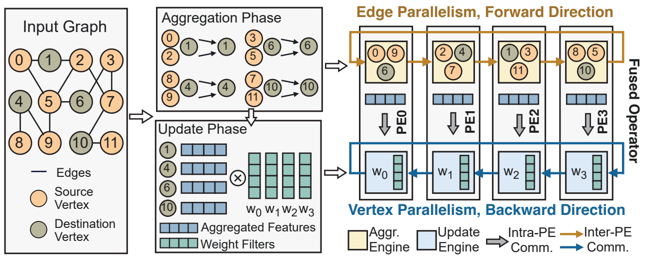
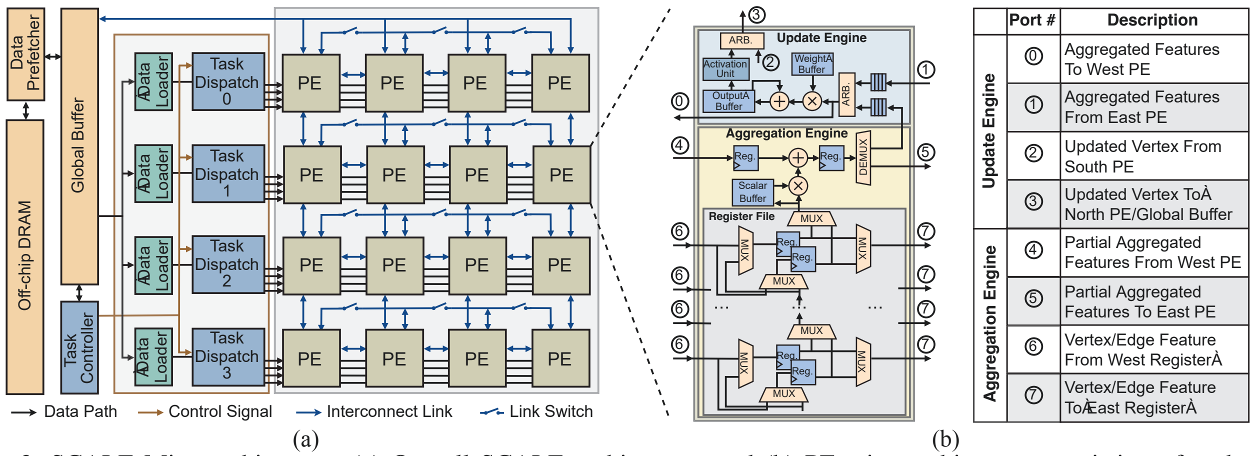


Figure 2: The proposed edge and vertex parallelism in SCALE.

in the backward direction, referring to the 'all-gather'. For example, vertices 0, 2, and 1 are loaded to PE0, PE1, and PE2 respectively. Vertex 0 will be forwarded to PE1 and aggregated with Vertex 2, and the intermediate result will be forwarded to PE2. Once the aggregation is completed at PE2, the aggregated feature will be forwarded to the update engine within PE2. The aggregation feature will be multiplied with distributed weight elements (w_0, w_1, w_2, and w_3) via backward direction. As such, all the intermediate results are exchanged and stored at the register level, increasing the data reuse. Meanwhile, the communication distance for each operation is one hop. SCALE exploits the edge parallelism at the aggregation phase, where multiple edge reduce operations are distributed and performed in parallel. On the other hand, it parallelizes the vertex update by distributing the weight matrix among processing units. Both edge and vertex parallelism are unified in one architecture via a coherent dataflow. As such, the aggregated features will be forwarded for vertex update within each PE to enable operator parallelism (e.g., aggregation and update).

A. *Overall SCALE Architecture*

Fig. 3(a) depicts the architecture of the proposed SCALE accelerator. SCALE consists of the following main components: a multi-bank global buffer, a task controller, data loaders, task dispatchers, and a flexible processing element (PE) array. The multi-bank global buffer is used to store the graph information (vertex and edge) and the weight matrix. The task controller schedules edge and vertex tasks and assigns them to the array of task dispatch units. The task dispatch unit orchestrates the data movement of their respective tasks from the global buffer to the PE array using the data loader to forward the data to the PE array. The PEs are connected by a flexible network, which can be dynamically sized to multiple PE sub-arrays. Each PE can support both message aggregation and vertex update phases at the same time. We will detail the design of the PE array to simultaneously support both message aggregation and vertex update phases in the following.



	Port #	Description
Update Engine	0	Aggregated Features To West PE
	1	Aggregated Features From East PE
	2	Updated Vertex From South PE
	3	Updated Vertex ToÀ North PE/Global Buffer
Aggregation Engine	4	Partial Aggregated Features From West PE
	5	Partial Aggregated Features To East PE
	6	Vertex/Edge Feature From West RegisterÀ
	7	Vertex/Edge Feature ToÀEast RegisterÀ

Figure 3: SCALE Microarchitecture: (a) Overall SCALE architecture, and (b) PE microarchitecture consisting of update and aggregation engine.

B. SCALE PE Array Architecture

The proposed PE array is a systolic array-like architecture, which simultaneously enables three types of parallelism, namely edge (i.e., feature aggregation), vertex (i.e., vertex update), and operator (i.e., the dependency between aggregation and update) parallelism. Each row of the PE array is interconnected in a ring-like topology. The wrap-up link consists of a set of link switches (i.e., transistors) to disable the signal propagation between segmented short links. By doing so, a long ring can be divided into several shorter rings. These small rings can be configured to handle various graph mapping and dataflow. The forward ring direction within this structure facilitates the reduce operation required for message aggregation. On the other hand, the ring's backward direction supports the vertex update. Any operator dependency is managed within each PE once the feature aggregations are completed. Consequently, the proposed PE array can achieve high parallelism without incurring extra storage or communication overheads.

However, several challenges arise when designing the proposed PE due to the irregular nature of the graph structure, coupled with varying degree numbers. Even though EnGN [15] attempts to perform ring-based reduce, it requires distributing the vertex features over the full length of the PE array because of the vertex-based mapping. Additionally, conventional designs often require aggregated features to be sent to global buffers or a centralized update engine. To overcome these limitations, we present two distinctive designs, aggregation and update engines within each PE, as illustrated in Fig. 3(b).

The aggregation engine is equipped with a dedicated multiply-and-accumulate (MAC) unit, along with a register array. The adder, multiplier, and scalar buffer within this structure are configurable, thereby enabling various aggregation operations essential for GNN models. Not only can each aggregation engine transmit the aggregated feature to an adjacent PE, but it can also forward them to the update engine housed within the same PE. In addition, each aggregation engine has a shift register array, comprising an array of double buffers. It has the capacity either to provide feature or weight data to MAC units or to forward the information to adjacent register arrays. The function of the update engine is to execute vector-vector multiplication, and it includes a weight buffer, an output buffer, and an activation unit for non-linear functions.

1) Aggregation Phase: The aggregation phase either leverages the edge or feature parallelism to perform multiple reduce operations simultaneously. To support this idea, the difficulty is two-fold: (1) The vertex features need to be distributed to the PE array, and each feature should be aggregated with its associated reduce operation; (2) the reduce operations increase data bandwidth requirements, where multiple features result in just one aggregated feature. To solve the mentioned challenges, we operate on the individual feature of the vertices rather than the entire feature vector reducing the bandwidth requirement across the aggregating PEs. Then, we utilize a task dispatcher to distribute the feature workload to the PEs and a shift register array with double buffers, storing feature values for multiple vertices and edges, to overlap the latency of feature distribution with feature aggregation.

For example, as shown in Fig. 4(a), four reduce operations will be performed on two PEs. Each reduce operation includes various vertices, in which source vertices send the features to the destination vertices (a, b, c, and d). These feature aggregations will be performed on the PE ring hop by hop. We first map features of each reduce operation to the aggregation engines. As such, the shift registers at each PE will receive two features. For example, features (a_0^0 and c_1^0) are loaded to the shift registers at PE0. After that, the register array pops the features up to the MAC units in the aggregation unit. Each aggregation unit will perform the reduce operation, receiving one operand from its upstream PE and another from its local register array. For example, a_0^0 is popped to the MAC units at cycle 1, and it will be forwarded to PE1 and added with a_1^0 at the next cycle. The accumulated result, m_a^0, will be further forwarded to the next PE until it finishes aggregating with all the source vertices. The final accumulated result corresponding to vertex a, M_a^0, will be sent to the update engine at PE1 during cycle 4. As such, the aggregated features can be directly sent to the next operation without incurring data

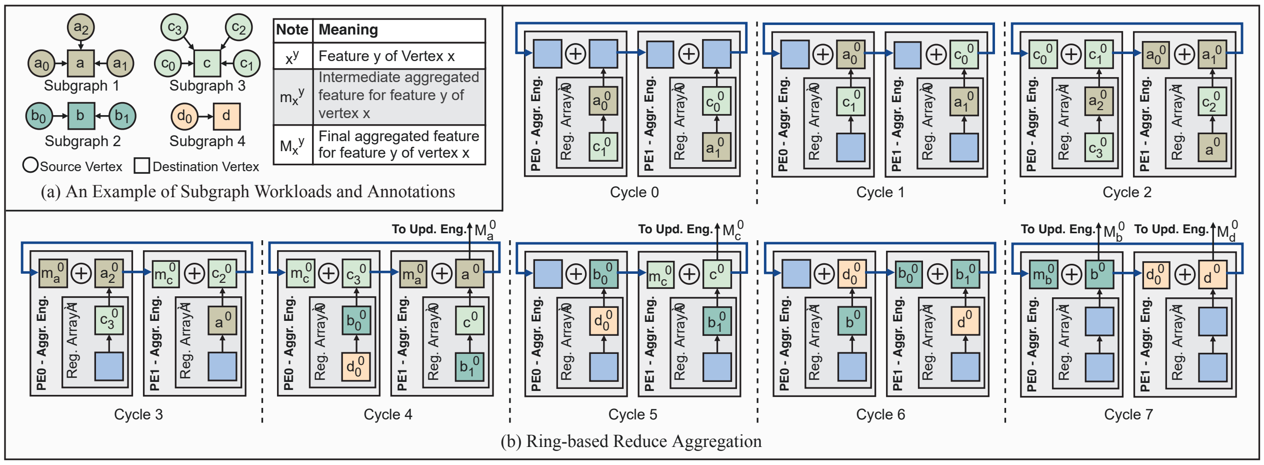


Figure 4: Aggregation phase example in the proposed SCALE architecture with a 1×2 PE ring.

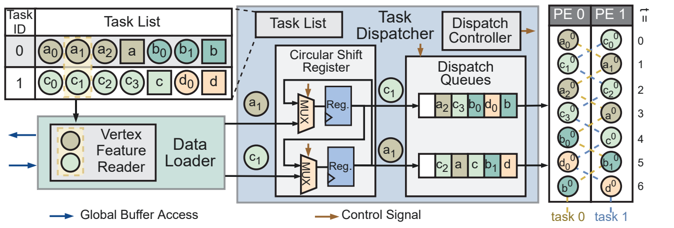
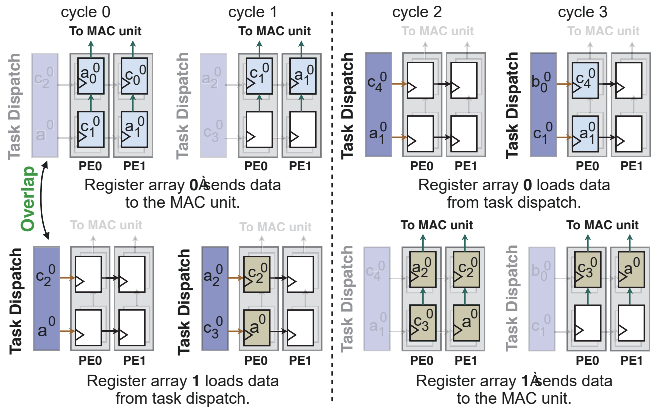


Figure 5: An example of task dispatcher feeding vertex features to the PE array.

movement across the memory hierarchy. This eliminates the communication complexity and memory operations. Note that using a ring interconnect between the PEs allows arbitrary-length subgraphs to be mapped onto the PE ring as large workloads (vertices with high degrees) can wrap around the PE ring multiple times. Fig. 4(b) shows that subgraphs 3 with four aggregations can be mapped to a PE ring of size 2.

Note that the SCALE microarchitecture supports a wider variety of aggregation functions employed by GNN architectures. A key property of the aggregation functions is permutation invariance [29]. This indicates that the ordering of the input features does not affect the output. Reduction operation is often associative and commutative, ensuring permutation invariance over the inputs [43]. By representing aggregation functions as a reduction operation over the features, SCALE ensures support for emerging GNN models.

Task Dispatcher: Fig. 5 illustrates the operation of a task dispatcher. The task dispatcher iterates over every source and destination vertex of all tasks in the task list and loads a portion of their respective features with the data loader. These features are sent through a circular shift register, similar to a barrel shifter, to reorder the vertex features. Note that the circular shift register is not the same as the shift register in the register array. The circular shift register is used to reorder the vertices, whereas the shift registers in the register array are used to supply the data to the PEs. For example, vertex features a_1^0 and c_1^0 are read from the global buffer by the data loader. This pair of features is sent through the circular shift register where it is shifted $i\%T_{n\_ring}$ times where i is the index in the task list and T_{n_ring} is the number of tasks assigned to a PE ring. In this example, i is 1, and T_{n_ring} is 2, leading to a shift



cycle 0
cycle 1
cycle 2
cycle 3
To MAC unit
Task Dispatch
Overlap
PE0
PE1
Register array 0 sends data to the MAC unit.
Register array 0 loads data from task dispatch.
Register array 1 loads data from task dispatch.
Register array 1 sends data to the MAC unit.

Figure 6: An example of overlapped task dispatch and aggregation operations with double buffered register array.

of 1. The reordered features are then pushed into the dispatch queues to be sent to the register array.

Shift Register Array: Naturally, the fused graph and neural operations, when performed simultaneously, increase the data bandwidth requirements. To overcome this problem, we propose a double buffer design in the shift register array. Fig. 6 illustrates the shift register array architecture, in which the register arrays of two PEs are interconnected. Specifically, the proposed shift register array is arranged in a mesh-like network and supports both horizontal and vertical data transfer. The features are propagated horizontally to fill up local registers. Once the features are fully loaded into PEs, they are delivered to the MAC units in the aggregation engine vertically. For example, as shown in Fig. 6, the task dispatcher has already loaded the features (a_i and c_i) associated with two reduce operations into a $2{\times}2$ register array. At cycle 0, features a_0^0 and c_0^0 are supplied to the MAC units. Afterward, features c_1^0 and a_1^0 will be propagated to the MAC units. In the meantime, the register array 1 starts preloading the feature data corresponding to the next set of computations. For example, at cycle 0, features c_2^0 and a^0 are loaded to the first column of the register array 1, and all of them will be shifted horizontally at the next cycle. To maintain the full utilization of the PE ring, the register array size should be set to at least N when generalizing the register array size to an $N \times N$ array. It is important to

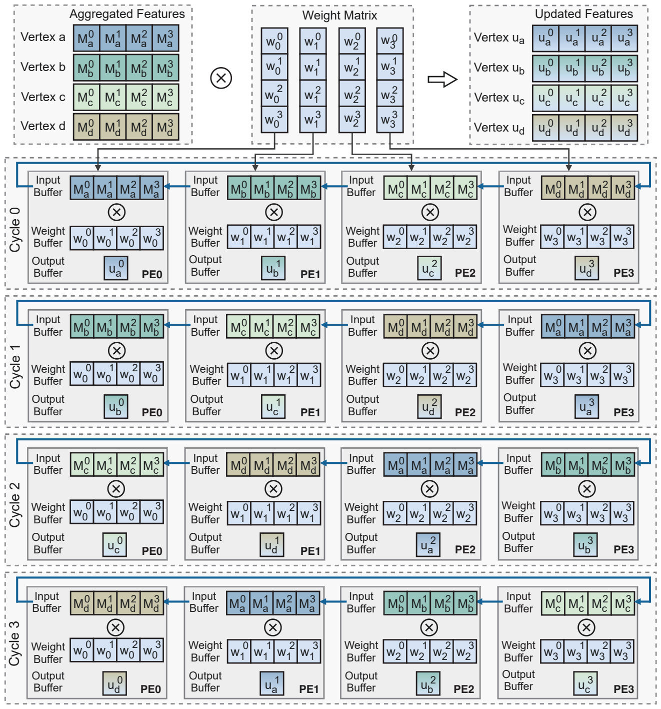


Figure 7: Vertex update example in SCALE architecture with a 1×4 PE ring.

note that the task dispatcher is only connected to the PEs in the PE array's leftmost column. To move data across the array, the dispatcher starts a shift operation while simultaneously pushing data into the register array.

2) *Update Phase:* The vertex update occurs once the accumulation of a vertex's features is complete. The fundamental computations of the vertex update are matrix-vector multiplications. In this process, feature vectors are multiplied with a weight matrix. Since the weight matrix is identical and shared across feature vectors, a weight-stationary dataflow is employed for the update phase. Specifically, we evenly distribute the computation of the update phase across a set of PEs by partitioning the weight matrix into equal chunks, where each PE holds one chunk. The weight matrix mapping is determined by matrix dimension and ring size, which will be handled by the task controller. This approach helps eliminate data duplication, enhancing the effective memory capacity. The weight matrix is pre-loaded into the PE array, while the aggregated feature is passed across PEs to facilitate computation with each weight partition. The data movement for vertex updates is orchestrated in a direction opposite to that of aggregation, fully utilizing both directions of the ring.

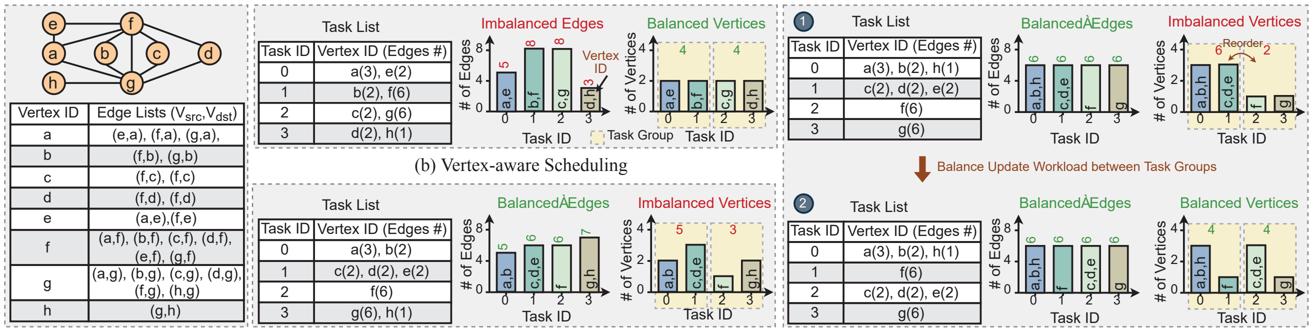
For example, as shown in Fig. 7, the feature vectors (a, b, c, and d) are received from the aggregation engine and are stored in the update engine. Each feature vector includes four elements (M_a^0, M_a^1, M_a^2, and M_a^3) and needs to be multiplied by the weight matrix. As the weight matrix is partitioned and distributed into the PE array, the weight vectors will be temporally stored at each PE for vertex update. The feature vector will move along the PE array to be multiplied with each weight vector. For example, the aggregated features (M_a^0, M_a^1, M_a^2, and M_a^3) will be multiplied by a weight vector (w_0^0, w_0^1, w_0^2, and w_0^3). The results (u_i^j, $i \in \{a, b, c, d\}$ and $j \in \{0, 1, 2, 3\}$) will be eventually sent back to global buffers through the vertical links in the PE arrays. The vertical links connect each PE with the corresponding PE in the row above it. As such, only the PEs of the topmost row are connected to the global buffer to write the updated features back to the global buffer. While the PEs of the other rows perform shift operations to send the updated feature to the row above it. This ensures design scalability as not all PEs need a direct connection to the global buffer. The aggregated feature vectors will be forwarded to the next hop. We note that each aggregated feature has to traverse $N - 1$ hops (N is the ring size) to accomplish the update operation.

3) *Workload Imbalance and Ring Size:* The efficiency of the proposed architecture is clearly influenced by the variability in vertex degree and weight matrix dimension size. Specifically, the edge parallelism in reduce operations is linked to the vertex degree since the number of features corresponds to the vertex degree. With a fixed PE array size, accommodating varying sizes of reduce operations becomes a complex task. Additionally, the size of the weight matrix determines the spatial parallelism, where each weight vector is divided among multiple PEs. These factors collectively contribute to the challenges of fusing the aggregation and update phases into a cohesive dataflow. Such integration demands meticulous design in terms of workload scheduling and PE array size, which will be discussed in the following sections.

IV. Proposed Scheduling Policy

As mentioned, the message passing GNNs follows a neighborhood aggregation scheme, where the feature vector of a vertex is computed by recursively aggregating and transforming feature vectors of its neighboring vertices [29]. This indicates that the computation associated with the aggregation phase depends on the number of neighboring vertices (represented by the edges of a vertex). Further, the update phases transform the feature representation of each vertex, indicating that the workload is proportional to the number of vertices. In distributing the workload across the proposed PE array, the ideal scenario would involve an even task allocation of edge and vertex to avoid workload imbalance at different stages. However, the varying degrees of vertices in power-law graphs pose a complicated challenge to the workload distribution.

In the context of workload scheduling, prior work typically falls into one of two categories: vertex-aware scheduling [44]–[46] and degree-aware scheduling [15], [47], [48]. Vertex-aware scheduling allocates an equal number of vertices to computing units. To illustrate this, we consider the example graph shown in Fig. 8(a). Vertex-aware partitioning forms four tasks, each containing an equal number of vertices but with differing degrees, as depicted in Fig. 8(b) with each task assigned to one PE. While this method balances the workload during the update phase, it leads to an unbalanced load in the aggregation phase, resulting in unbalanced PE utilization. However, degree-aware scheduling organizes vertices into tasks in such a way that the total degree of the vertices



Vertex ID	Edge Lists (V_{src},V_{dst})
a	(e,a), (f,a), (g,a),
b	(f,b), (g,b)
c	(f,c), (f,c)
d	(f,d), (f,d)
e	(a,e),(f,e)
f	(a,f), (b,f), (c,f), (d,f), (e,f), (g,f)
g	(a,g), (b,g), (c,g), (d,g), (f,g), (h,g)
h	(g,h)

(a) An Example Graph and Its Edge List Representation

Task List

Task ID	Vertex ID (Edges #)
0	a(3), e(2)
1	b(2), f(6)
2	c(2), g(6)
3	d(2), h(1)

(b) Vertex-aware Scheduling

Task List

Task ID	Vertex ID (Edges #)
0	a(3), b(2)
1	c(2), d(2), e(2)
2	f(6)
3	g(6), h(1)

(c) Degree-aware Scheduling

1 Task List

Task ID	Vertex ID (Edges #)
0	a(3), b(2), h(1)
1	c(2), d(2), e(2)
2	f(6)
3	g(6)

2 Task List

Task ID	Vertex ID (Edges #)
0	a(3), b(2), h(1)
1	f(6)
2	c(2), d(2), e(2)
3	g(6)

(d) Degree and Vertex (DV)-aware Scheduling

Figure 8: Comparisons of different scheduling policies when assigning four tasks to a 2×2 PE array, and each task is assigned to the corresponding same PE ID. PE 0 and 1 form one ring, and PE 2 and 3 form the other ring. (a) an example graph, (b) vertex-aware scheduling, (c) degree-aware scheduling, and (d) our proposed degree and vertex-aware scheduling.

within each task is balanced. Fig. 8(c) provides an example of degree-aware scheduling, with four tasks created to have an equal number of edges. Although this approach ensures balance during the aggregation phase, it may create imbalances during the update phase due to the varying numbers of vertices in each task, which leads to unbalanced PE utilization.

A. Degree and Vertex-aware Scheduling

Finding a balance between the number of edges and vertices can be particularly challenging, as graph data often exhibit high irregularities. To tackle this challenge, we introduce a degree and vertex-aware task scheduling algorithm, in which a greedy heuristic is proposed for this bin-packing problem [49]. The process begins with degree-aware scheduling to create edge-balanced tasks. These tasks are then combined into equally distributed task groups using a modified vertex-aware scheduling approach. This two-step method ensures balanced edge and vertex workloads across all task groups.

Algorithm 1 outlines the pseudo-code for the proposed task scheduling method, which consists of two main steps. In the first phase, our goal is to organize vertices into tasks that are balanced in terms of edges. We approach this as a modified version of the bin-packing problem, where each "bin" or task has a size equivalent to the number of edges in that task. Unlike the traditional bin-packing problem, which seeks to minimize the number of bins, our version fixes the number of bins to the maximum number of tasks the hardware can handle, a limit set by the register array size. As outlined in lines 17-30 of the algorithm, we apply the first-fit heuristic, a common approach used for runtime bin packing. For SCALE, we set the number of tasks, T_n, and number of task groups, G_n, to be equal to the number of PEs and rings, respectively.

The second phase involves grouping these edge-balanced tasks into task groups, ensuring that the number of vertices within each group is balanced. It's important to note that the number of task groups is constrained by the number of available rings. To achieve the balance, we first sort the tasks according to the number of vertices in each task, as indicated in line 4. We then use a simple modulo operation in line 13 to identify the index for the task group and place the task there. This method allows us to pair tasks with high vertex workloads with those having low vertex workloads, resulting in task groups that have balanced vertex workloads.

Algorithm 1: Pseudocode for degree and vertex-aware scheduling

```
Input  : Graph: $G = (V, E)$; Number of Tasks: $T_n$; Number
         of Task Groups: $G_n$
Output: Workload balanced Task Groups: $G$
1  Function Hierarchical_Scheduling($G$, $Task\_Groups$)
2      // Create Edge-Balanced Tasks
3      $Task\_List = First\_Fit(G,\ T_n)$;
4      $Sorted\_Task\_List = Sort(Task\_List)$;
5      // Initializing Task Groups
6      $Task\_Groups = []$;
7      for $i$ in $0 \to G_n$ do
8          $Task\_Groups.append(new\ Task\_Group())$;
9      end
10     // Assign Tasks to Task Groups
11     for $Task\_num$ in $T\_n$ do
12         $Task = Sorted\_Task\_List[Task\_num]$;
13         $Task\_Group = Task\_Groups[Task\_num \% G_n]$;
14         $Place\ Task\ in\ Task\_Group$;
15     end
16     return $Task\_Groups$;
17 Function First_Fit($G$, $T_n$)
18     // Initializing Tasks
19     Tasks = [];
20     for $i$ in 0 $\to T_n$ do
21         $Tasks.append(new\ Task())$;
22     end
23     // Assign Vertices to Tasks
24     for $Vertex\ V$ in $Graph\ G$ do
25         for $T$ in $Tasks$ do
26             if $T.edges + V.edges \leq T.target\_edges$ then
27                 $Place\ V\ in\ T$;
28         end
29     end
30     return $Tasks$;
```

Fig. 8(d) illustrates the degree and vertex-aware scheduling for a 2×2 PE array. We create four tasks (Task 0 - 3), each containing a similar number of edges (i.e., 6). Take Task 0 as an example; it includes 6 edges from three vertices (vertex a, b, and h). These edges represent the reduce operations to be performed at each PE during the aggregation phase. We further combine the tasks into pairs, referred to as "task groups". Each group of tasks is assigned to one PE ring. For instance, Task 0 and Task 1 are grouped together as group 0, which has four vertices (vertex a, b, h, and f). This number represents the vertex updates for the group. With two PE rings, we form two task groups, ensuring that both the aggregation and update phases have a similar workload in both rings. It's worth noting that Task 0 includes three small-degree vertices and produces three aggregated feature vectors. Meanwhile, Task 1 includes a large-degree vertex and produces one aggregated feature. While the edge and vertex workloads among the task groups are balanced, there may be some vertex imbalance within a task group due to the greedy policy. Our architecture and mapping strategies, however, can tolerate this imbalance. As the weight matrix is distributed across PEs and the aggregated vertex features circulate through all PEs in a ring during the update phase, it ultimately equalizes the workload among PEs.

Following Algorithm 1, we implement the proposed scheduling algorithm in hardware allowing SCALE to schedule workloads during runtime. Task scheduling involves creating workloads and writing them to the task lists of each task dispatcher unit. To vary the scheduling latency, the task scheduler creates tasks over a subset of the graph vertices defined by batch size B. To minimize task scheduling overheads, we decouple the task scheduling and execution by using a double-buffered task list in the task dispatcher, allowing the task controller to schedule new tasks while the previously scheduled tasks are being executed. This allows SCALE to overlap the task scheduling latency with the task execution. By employing such a runtime mechanism, we avoid preprocessing the graph data. To hide the task scheduling latency with the task execution, SCALE uses a performance model, as described in IV-B, to estimate their latency and restricts the batch size parameter, B to ensure that the task scheduling does not hinder task execution.

B. Performance Model

The batch size, B, indicates the number of vertices operated by the task scheduler to allocate tasks during one pipeline phase. SCALE selects a suitable value for B such that the task scheduling does not hinder the first pipeline stage of task execution. Specifically, the task scheduling latency, t_{ts} must be less than the aggregation latency, t_{agg} of the tasks. SCALE implements an analytical model of task scheduling and task aggregation to determine their respective execution time for a given subgraph with an average degree of D_{avg}.

The task scheduling latency, t_{ts}, from Algorithm 1 can be written as the sum of time to create tasks and task groups. Operations involved in creating tasks, as indicated by the nested for-loops in Algorithm 1 (lines 24-29), require, on average, $B \times Log(T_n)$ on-chip memory access. Task group creation, as indicated by sorting tasks and a for loop in lines 4-15, takes $(T_n \times Log(T_n) + T_n)$ on-chip memory accesses. Thus the task scheduling latency can be written as

$$t_{ts} = ((B + T_n) \times Log(T_n) + T_n) \times t_{ocm}$$

Here, t_{ocm} is the access latency for on-chip memory. Similarly, the aggregation of a task involves the PEs performing $B \times D_{avg}$ reduce operation and inter-PE communication for each feature. Moreover, the same task list can be reused to perform the computation associated with F_n features of the vertex. Since these reduce operations are performed in parallel by T_n PE, the task scheduling latency can be written as

$$t_{agg} = \frac{B \times D_{avg}}{T_n} \times (t_{reduce} + t_{comm}) \times F_n$$

Here, t_{reduce} and t_{comm} are the latency to perform a reduce operation and inter-PE communication, respectively. If the t_{ts} is larger than t_{agg}, the aggregation engines of the PE will remain idle, leading to resource underutilization. Therefore, for a given accelerator configuration, SCALE chooses the batch size B carefully such that $t_{ts} < t_{agg}$. The sensitivity study is provided in Section VII-F.

V. Proposed Dataflow and Mapping

Both dataflow and mapping choices affect the performance by exploiting the temporal and spatial data locality via loop interchanging and spatial parallelism [50]–[52]. The weight matrix is of a relatively lower dimensionality compared to CNNs. Therefore, conventional CNN dataflow and mapping methods have limited applicability to message passing-based GNN acceleration.

In this work, the principal objective of the mapping strategy is to parallelize both feature aggregation and vertex update in a coherent dataflow. To attain this, the proposed flexible ring plays a critical role in achieving spatial parallelism. For instance, the weight matrix must be distributed across different PEs within the same PE ring, thereby mitigating data duplication and resource under-utilization. However, the dimensionality of the weight matrix differs among datasets. In the update phase of the GCN's second layer on the Cora dataset, a weight matrix of dimension 16×7 is utilized, while for Nell, the dimensions are 64×168. This necessitates a ring size adaptable to diverse matrix dimensions. For example, if the ring size S_{ring} is small, then the sum of weight buffers B_{weight} in the ring may be smaller than the entire weight matrix W. This would incur off-chip memory accesses to load portions corresponding to the weight matrix that is not present in the weight buffers. On the other hand, employing a ring size S_{ring} larger than the total available computations in the update phase would reduce the PE utilization rate. This is caused by certain update engines being idle due to the lack of available computations. Formally, we show the optimal range of ring size as

$$S_{ring} \in [\lceil \frac{W}{B_{weight}} \rceil, R_{weight} \times C_{weight}], S_{ring} \in \mathbb{Z}^{+} \quad (3)$$

where R_{weight} and C_{weight} are the row and column dimensions of the weight matrix, respectively.

Upon determining the ring size, the flexible PE array will be configured accordingly. As depicted in Fig. 9(a), if the ring size is set to 2, a row of the array is configured into two PE rings, resulting in the formation of eight rings to

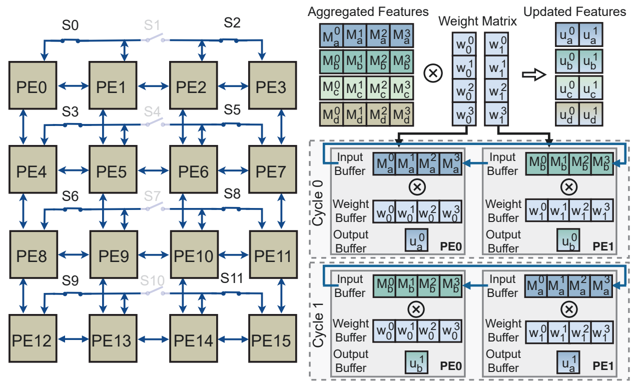


Figure 9: An example of the proposed ring configurations in which PE ring size is 2. (a) link switch configuration and (b) vertex update workloads on a 1×2 PE ring.

facilitate the concurrent execution of feature aggregation and vertex update. To illustrate this, Fig. 9(b) shows aggregated feature vectors, such as M_a^0, M_a^1, M_a^2, and M_a^3, are distributed among PEs. In parallel, each column of the weight matrix, like w_0^0, w_0^1, w_0^2, and w_0^3, is also distributed to PEs by the task controller. The aggregated feature vectors only need to pass through two PEs, where they are then multiplied by each corresponding weight partition. The resulting products, such as u_a^0, u_b^0, u_b^1, and u_a^1, can be directly written back to the global buffer. If an outer product were chosen, it would require additional buffers at each PE to hold the intermediate data. This method would cause the data movement to be inconsistent with the aggregation phase and would also introduce additional computations to accumulate all the partial results.

Once the optimal ring size is determined, the PE array will be configured. As the size of the ring is determined by the dimension of the weight matrix used in the update phase of each layer, we reconfigure the rings between the execution of layers. Note that GNNs have fewer layers than other neural networks (e.g., CNN), so the reconfiguration overheads, involving simple switch toggling, are negligible even if SCALE reconfigures the ring size for each layer.

VI. Experimental Setup

We implemented SCALE using a validated cycle-accurate C++ simulator to model the behavior of the hardware. We integrate our simulator with Ramulator [53] configured as High-bandwidth memory (HBM) with a bandwidth of 256 GB/s to model the off-chip storage. We used single-precision floating point datatype according to IEEE 754 for our evaluation. We evaluated the performance of SCALE using three citation graphs (Cora, CiteSeer, and PubMed) [54], one knowledge graph (Nell) [55], and one large post graph (Reddit) [28]. The detailed size and length of features in different layers of data sets are summarised in Table II. Additionally, we used the most popular representative GNN models: GCN [26], Gated-GCN (G-GCN) [27], GraphSage-Pool (GS-PL) [28], and GIN [29]. The programming model that we used is similar to Deep Graph Library (DGL) and PyTorch Geometric.

Table II: Structure of the Graph Datasets for the 2-layer GNN.

Datasets	Vertices	Edges	Average Degree	Feature Length
Cora	2,708	10,556	3.9	1,433 - 16 - 7
CiteSeer	3,327	9,104	2.7	3,703 - 16 - 6
PubMed	19,717	88,648	4.5	500 - 16 - 3
Nell	65,755	251,550	3.8	61,278 - 64 - 210
Reddit	232,965	114,615,892	492	602 - 64 - 41

CAD Tools and Methodology: To estimate the power, area, and timing characteristics of SCALE as an ASIC accelerator, we implemented it using Verilog. We synthesized it using Synopsys Design Compiler with the TSMC 32 nm standard library to learn its timing characteristics that we used in our simulator validating the accuracy of our model. We set the clock frequency at 1GHz. We performed RTL simulations to generate the waveform activity file to observe the switching activity of the logic gates. Next, we used Synopsys PrimeTime PX with the waveform activity file to measure the dynamic and static power consumption of SCALE. For the area and power of on-chip buffers, we employed Cacti 6.0 with 32 nm technology [56].

Baseline Platforms: To evaluate the efficiency and scalability of SCALE, we compare it with prior state-of-the-art GNN accelerators such as ReGNN [4], FlowGNN [3], AWB-GCN [2], and GCNAX [1]. We model the baseline architectures using our C++ simulator employing their respective optimizations. For a fair comparison, all the baseline accelerators are scaled to have the same clock frequency and the same number of single precision MAC units as SCALE. As FlowGNN utilizes a hybrid engine with node transform units performing the update phase and message passing units performing the aggregation phase, we use twice as many message passing units as node transform units. Although GCNAX and FlowGNN perform optimizations such as parallelization strategies and loop fusion, they suffer from imbalanced workloads in their processing units when scaling up the number of MAC units. Our reported latency of GCNAX and FlowGNN accounts for this workload imbalance. Additionally, we have scaled the bandwidth and on-chip memory to match SCALE.

VII. Evaluation Results

A. Performance Analysis

We configure SCALE as 32×16 PE array with a 4 MB global buffer. Each PE has 6 KB local buffers (4 KB update engine buffer and 2 KB aggregation engine buffer) and 2 MAC units, so the total MAC units of SCALE is 1024. We utilize the number of execution cycles as the performance metric. Fig. 10 shows the speedup compared to other state-of-the-art GCN and GNN accelerators. SCALE, on average, is $1.62\times$ and $2.01\times$ faster than AWB-GCN and GCNAX for the GCN model, respectively. For other models such as G-GCN, GS-PL, and GIN, SCALE, on average, is $1.57\times$ and $1.80\times$ faster than FlowGNN and ReGNN, respectively. The performance improvement primarily stems from the balanced workload with improved PE utilization and the coherent dataflow with simplified communication patterns. As shown in Fig. 11, the

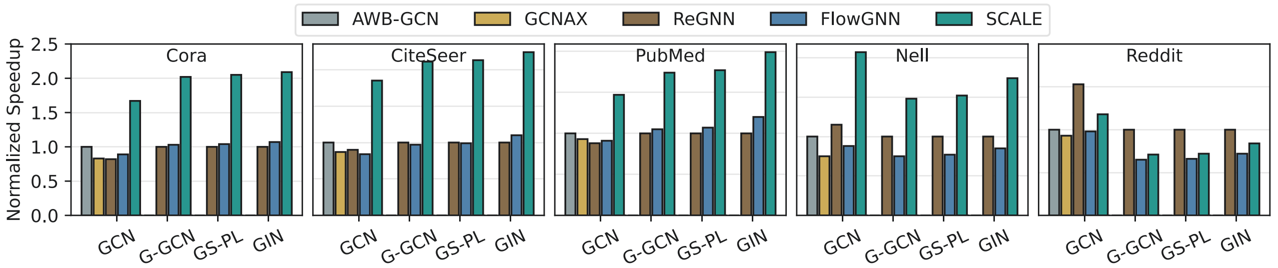


Figure 10: Normalized speedup comparison of AWB-GCN, GCNAX, ReGNN, FlowGNN, and SCALE for different datasets and GNN models.

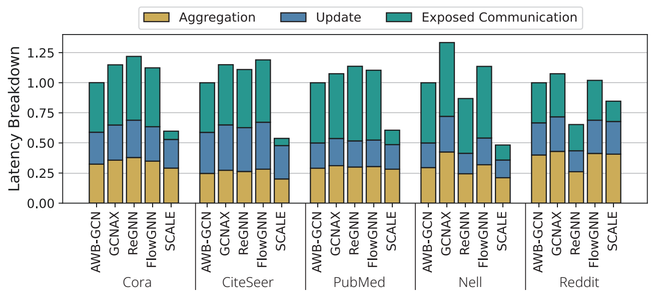


Figure 11: Latency Breakdown.

exposed communication latency is reduced by up to 87.56% as compared to baseline architectures. Next, we evenly distribute the workload that considers both edge and vertex variations, leading to better PE underutilization, which contributes up to 50.35% latency reduction in both aggregation and update phases. FlowGNN and GCNAX suffer from high workload imbalance in the aggregation phase due to their fixed workload assignment strategy, which limits their performance. On the other hand, although AWB-GCN mitigates the workload balance issue, they do not pipeline both phases of GNN computation. This comes with a considerable amount of redundant memory accesses. ReGNN eliminates redundant computation, but it employs disjoint engines for both phases and suffers from workload imbalance in the aggregation phase.

Although SCALE shows considerable improvement over the prior art for Cora, CiteSeer, PubMed, and Nell, its performance is slightly worse than ReGNN on Reddit. The reason is two-fold. First, Reddit graph dataset presents better regularity in graph connectivity, in which edge and vertex degrees show high similarity. Coupled with a low feature length, Reddit inherently exhibits a balanced workload on baseline accelerators. Additionally, Reddit has high graph-level data reuse, in which a pair of vertices is connected to a set of mutually shared vertices. Based on our dataset profiling, 75.5% of aggregation operations can be eliminated in Reddit. As such, ReGNN, on the other hand, outperforms SCALE as it primarily aims to minimize the large amount of redundant computation in Reddit. However, it must be noted that such redundancy removal techniques are orthogonal to SCALE. To further understand the performance benefits of SCALE, we implement the redundancy removal strategy to reprocess the graph dataset, and Table. III shows that SCALE with redundancy removal can outperform ReGNN by an average of 1.23× on the Reddit dataset across different models.

Table III: Normalized speedup of SCALE with redundancy removal compared to ReGNN.

Datasets / Models	Cora	CiteSeer	PubMed	Nell	Reddit
GCN	2.15	2.31	1.98	2.07	1.13
G-GCN	2.22	2.36	1.92	1.85	1.34

B. *Scalability Analysis*

To compare the scalability, we normalize the accelerator's speedup over AWB-GCN configured with 512 MACs. The array dimensions of the PE array in our design are set as 16×16, 32×16, 32×32, and 64×32 for 512, 1K, 2K, and 4K MACs, respectively. We prefer to increase the row dimension rather than the column dimension, as increasing the column dimension will lead to a larger shift register array. As shown in Fig. 12, SCALE shows better scalability than prior accelerators showing an average speedup of 12.07× compared to speed up of 7.61×, 6.49×, 7.3×, and 6.68× for AWB-GCN, GCNAX, ReGNN, and FlowGNN respectively when configured for 4K MACs. SCALE shows better scalability primarily due to three reasons - a unified dataflow architecture, a balanced workload, and simplified interconnects. To be specific, even though AWB-GCN resolves the workload balance, it relies on an all-to-all network to redirect computations to underutilized computation resources. The disjoint aggregation and update engines of all the prior works require a complicated network to ensure data movement, which all suffer from bandwidth or latency issues. SCALE eliminates the need for complicated networks by having the proposed degree and vertex-aware scheduling and coherent dataflow. Given this, SCALE shows the best improvement for Nell, as it exhibits high irregularity in the graph structure and large feature length exacerbating workload imbalance in baseline accelerators. Even for such irregular graphs, SCALE shows good workload balance and a high degree of parallelism with a large accelerator size.

C. *Workload Balance Analysis*

To evaluate the proposed workload balancing techniques, we analyze the PE utilization of SCALE and compare it with the PE utilization of state-of-the-art GNN and GCN accelerators, FlowGNN (FG) and AWB-GCN (AWB). Note that all accelerators are configured with 1K MACs. We use performance measurement counters to measure the active cycles of the PEs during both the aggregation and update phase of all the models and report the average utilization.

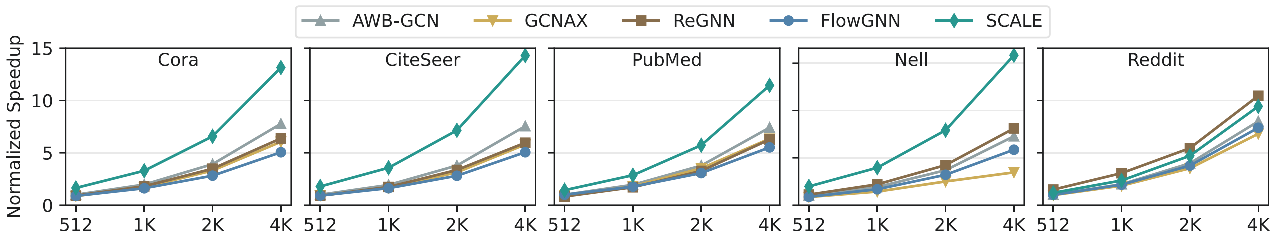


Figure 12: Scalability comparison: Normalized speedup of AWB-GCN, GCNAX, ReGNN, FlowGNN, and SCALE for different datasets with varying MAC units.

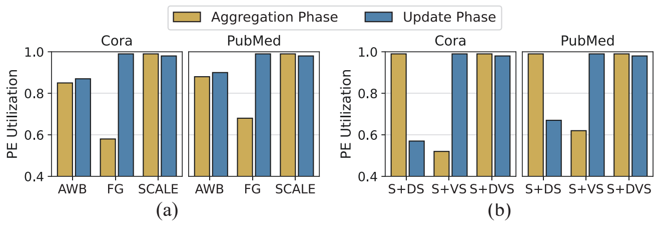


Figure 13: (a) Average PE utilization of two phases with different datasets and accelerators, and (b) ablation study of scheduling policies.

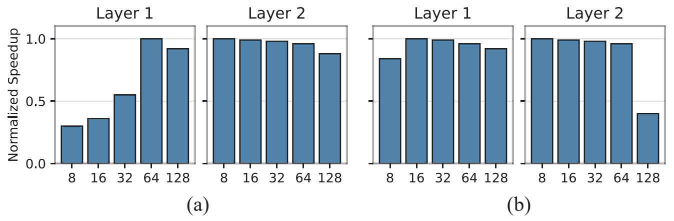


Figure 14: Performance comparisons with various ring sizes when running 2-layer GCN with (a) Cora and (b) PubMed.

Fig. 13(a) shows SCALE efficiently balances the workload in both phases with an average PE utilization of 98.7% and 97.3% in aggregation and update, respectively. FlowGNN employs a scheduling similar to vertex-aware scheduling, which shows an average PE utilization of 99.1% in the update phase and 62.8% in the aggregation phase. AWB-GCN employs a runtime workload balancing scheme to distribute the workload evenly across the PEs, achieving a utilization close to SCALE with a PE utilization of 86.4% for the aggregation phase and 88.5% for the update phase on average. However, AWB-GCN's runtime balancing policy cannot be directly used for other GNN models that cannot be represented as SpMM. In contrast, SCALE, with the proposed scheduling, can balance the workload in both phases.

D. Ablation Study of Scheduling Policy

We include an ablation study of various scheduling policies, such as degree-aware scheduling (S+DS), vertex-aware scheduling (S+VS), and degree and vertex-aware scheduling (S+DVS). Fig. 13(b) shows that S+DVS achieves high PE utilization in both phases. On the other hand, S+DS has a high utilization of 99.1% in the aggregation phase but a lower utilization of 58.7% in the update phase. This is due to the even distribution of edges, which results in a balanced workload in the aggregation phase. S+VS has a high utilization of 99.2% in the update phase but a lower utilization of 54.7% in the aggregation phase due to the uneven distribution of edges. However, either scheduling is inefficient in balancing the workload in both phases.

E. Sensitivity Study of Ring Size

To study the effects of varying ring sizes on performance, we perform a sensitivity study in a 2-layer GCN model using Cora and PubMed Datasets, where the scheduling policy is consistent across all configurations. Fig. 14 shows that the performance varies across datasets. For example, the first layer of the GCN model would prefer a ring size of 64. This is because a small ring size may not be able to accommodate the entire weight matrix, resulting in excessive off-chip memory access. Conversely, a ring size that is too large may require a longer initial data load time and could result in a less balanced workload. For the second layer in Cora and PubMed datasets, the dimension of the weight matrix is 16×7 and 16×3, respectively. In such a case, the update phase would suffer from significant PE under-utilization because of the small weight size. To solve this, we configure SCALE with multiple small rings and duplicate the weight matrix, thereby increasing spatial parallelism.

F. Task Scheduling Overhead Analysis

Fig. 16(a) shows the ratio of task scheduling latency (t_{ts}) and task aggregation latency (t_{agg}) for various batch sizes. The task scheduling overhead is negligible when $t_{ts}/t_{agg} < 1$ (TS-Negligible) and the bottleneck when $t_{ts}/t_{agg} > 1$ (TS-Bound). We observe that the aggregation engines transition from TS-Bound to TS-Negligible as the batch size increases and graphs with a low feature length or low average degree require a larger batch size to cover the overhead. When batch size is above 500, t_{ts} is smaller than t_{agg} for all the datasets.

G. Energy and Area Analysis

Fig. 15 shows the energy breakdown of SCALE compared to baselines normalized to AWB-GCN. SCALE reduces the average DRAM and Global Buffer energy consumption by 36.8% and 53.2%, respectively.

The energy reduction results are from higher intermediate data reuse at the register level, thereby reducing read/write operations to Global Buffer and DRAM. This allows for efficient usage of on-chip storage leading to higher input data reuse. The intermediate data reuse will increase the number of read/write at the register level in SCALE, as such it exhibits a 5.72× Local Buffer energy consumption on average as compared to prior works. Meanwhile, Reddit presents a

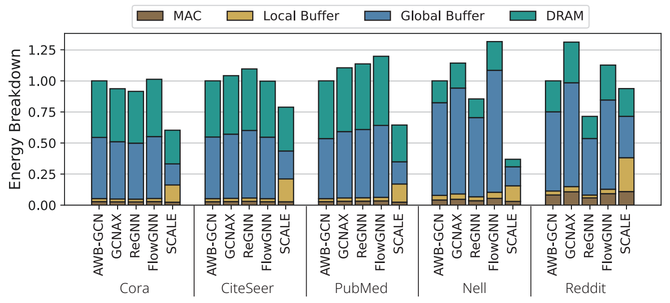
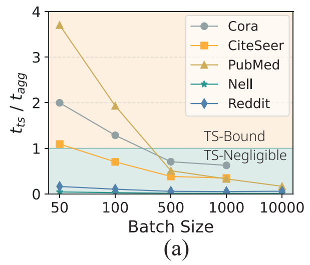


Figure 15: Energy Breakdown.



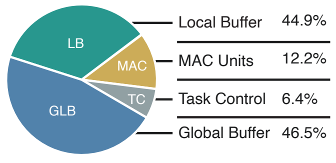


Figure 16: (a) Task scheduling overhead with different batch sizes and (b) area breakdown.

large portion of mutually shared vertices, which translates to a higher reduction in DRAM and Global Buffer access in ReGNN. For Nell, with a large feature length, the benefits of intermediate data reuse become more significant. Fig. 16(b) shows the breakdown of the total die area of SCALE. Storage components like Global Buffer and Local Buffer account for 81.4% of the total area, whereas the MACs and Task Control occupy 12.2% and 6.4% of the total die area, respectively.

VIII. Related Work

A. GCN Accelerators

GCN is one of the most prevalent GNN variants which allows the application of convolutional layers directly on graph-structured data. A variety of accelerators have been developed to enhance GCN performance. For example, AWB-GCN [2] employs a runtime workload rebalancing scheme to redistribute uneven workload through an all-to-all network. GCNAX [1] exploits loop fusion and reordering to unify the computation characteristics of both GCN phases while reducing off-chip memory access. I-GCN [22] employs a breadth-first search algorithm to extract dense matrix regions, thus improving computation efficiency and data locality. GCoD [57] decouples dense and sparse regions of the adjacency matrix and accelerates them in different computing engines. REFLIP [58] and PIM-GCN [59] accelerate GCN in the form of matrix-vector multiplication in crossbar-based processing-in-memory (PIM) architecture. However, prior research mostly focuses on analog computing, and their proposed design is unable to support complex GNN models with search and comparison functions. Furthermore, these GCN accelerators employ spatial or SIMD architectures adapting to graph irregularity, so intermediate and partial results are hardly reused at the register level.

B. Graph Quantization, Sampling, and Pruning

To further enhance GNN model performance, various optimization techniques have been proposed. DBQ [60] introduces a degree-based quantization, in which only insensitive nodes are quantized with low precision. MEGA [61] proposes a mixed-precision quantization method depending on vertex's in-degree. GNNSampler [62] exploits the data locality among nodes and their neighbors, eliminating irregular memory access. DyGNN [63] proposes EdgePrune and VertexPrune to eliminate redundant computations caused by message aggregation. PruneGNN [64] develops a dimension-pruning-aware sparse training method coupled with a SIMD-aware SpMM kernel to exploit matrix-operator-level parallelism. It is evident that all the quantization, sampling, and pruning optimizations only optimize the graph data without changing the GNN operations. As such, these optimization techniques are orthogonal to SCALE.

C. Sparse Tensor Algebra Accelerators

Given that the aggregation phase of several GNN models can be abstracted as SpMM, several accelerator architectures have been proposed to facilitate SpMM computations. OuterSpace [65] proposes an outer product-based SpMM to achieve higher input reuse compared to an inner product-based method. ExTensor [66] proposes a novel approach to eliminate all the unnecessary computations associated with zero elements. SIGMA [67] adopts a Benes network to dynamically pair non-zero elements and distribute them to all the multipliers. Flexagon [68] exploits a flexible dataflow design for Sparse-Sparse Matrix Multiplication (SpMSpM) adapting to various matrix dimensions. However, prior works typically are inefficient in handling GNN models due to the following issues. First, the mentioned sparse tensor accelerators typically target SpMM or SpMSpM with lower sparsity, whereas graph data exhibits a much higher sparsity ratio. In addition, the intermediate data reuse is not considered in the sparse tensor accelerators, as they are optimized only for SpMM, not for chained reduce and matrix multiplications.

IX. Conclusion

In this paper, we propose an elastic accelerator, SCALE, that can support a wide range of GNN models and graph irregularities with much-improved performance and energy efficiency. Specifically, SCALE consists of three designs, a novel systolic array-like architecture, a degree and vertex-aware scheduling, and a new dataflow tailored for fused graph and neural operations. The proposed systolic array-like architecture is robust to edge and vertex variations and can accommodate edge, vertex, feature, and operator parallelism simultaneously. The degree and vertex-aware scheduling can alleviate the workload imbalance issues in the message aggregation and vertex update phases. Additionally, the proposed dataflow can unify the data movement of both graph and neural operators without extra communication overheads. Our simulation results show that SCALE achieves 1.82× speedup with 38.9% energy reduction on average over the state-of-the-art GNN accelerators.

References

[1] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788. IEEE, 2021.

[2] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[3] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1099–1112. IEEE, 2023.

[4] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. ReGNN: A redundancy-eliminated graph neural networks accelerator. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 429–443. IEEE, 2022.

[5] Zhiwei Guo and Heng Wang. A deep graph neural network-based mechanism for social recommendations. In *IEEE Transactions on Industrial Informatics*, pages 2776–2783, 2021.

[6] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. DeepWalk: online learning of social representations. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 701–710. ACM, 2014.

[7] Xiao Li, Li Sun, Mengjie Ling, and Yan Peng. A survey of graph neural network based recommendation in social networks. In *Neurocomputing*, page 126441. Elsevier, 2023.

[8] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 974–983. ACM, 2018.

[9] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. Graph neural networks in recommender systems: A survey. In *ACM Computing Surveys*, pages 1–37. ACM, 2020.

[10] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. AliGraph: A comprehensive graph neural network platform. In *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 2094–2105. ACM, 2019.

[11] Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. Graph neural networks and their current applications in bioinformatics. In *Frontiers in Genetics*, 2021.

[12] John M. Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Zídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andy Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David A. Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. Highly accurate protein structure prediction with alphafold. In *Nature*, pages 583 – 589, 2021.

[13] Chuanze Kang, Han Zhang, Zhuo Liu, Shenwei Huang, and Yanbin Yin. Lr-gnn: A graph neural network based on link representation for predicting molecular associations. In *Briefings in Bioinformatics*, page bbab513. Oxford University Press, 2022.

[14] Shilin Tian, Chase Szafranski, Ce Zheng, Fan Yao, Ahmed Louri, Chen Chen, and Hao Zheng. Vita: Vit acceleration for efficient 3d human mesh recovery via hardware-algorithm co-design. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2024.

[15] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks. In *IEEE Transactions on Computers*, pages 1511–1525. IEEE, 2020.

[16] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. NeuGraph: Parallel deep neural network computation on large graphs. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, pages 443–458. ACM, 2019.

[17] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware acceleration of graph neural networks. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[18] Feng Shi, Ahren Yiqiao Jin, and Song-Chun Zhu. VersaGNN: a versatile accelerator for graph neural networks. In *arXiv preprint arXiv:2105.01280*, 2021.

[19] Zhuofu Tao, Chen Wu, Yuan Liang, Kun Wang, and Lei He. LW-GCN: A lightweight fpga-based graph convolutional network accelerator. In *ACM Transactions on Reconfigurable Technology and Systems*, pages 1–19. ACM, 2021.

[20] Lingxiang Yin, Jun Wang, and Hao Zheng. Exploring architecture, dataflow, and sparsity for gcn accelerators: A holistic framework. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, pages 489–495. ACM, 2023.

[21] Sanjay Gandham, Lingxiang Yin, Hao Zheng, and Mingjie Lin. SAGA: Sparsity-agnostic graph convolutional network acceleration with near-optimal workload balance. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9. IEEE, 2023.

[22] Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1051–1063. IEEE, 2021.

[23] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. In *arXiv preprint arXiv:1710.10903*, 2017.

[24] Md Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. Fusedmm: A unified sddmm-spmm kernel for graph embedding and graph neural networks. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 256–266. IEEE, 2021.

[25] Sami Abu-El-Haija, Bryan Perozzi, Amol Kapoor, Nazanin Alipourfard, Kristina Lerman, Hrayr Harutyunyan, Greg Ver Steeg, and Aram Galstyan. Mixhop: Higher-order graph convolutional architectures via sparsified neighborhood mixing. In *Proceedings of International Conference on Machine Learning (ICML)*, pages 21–29. PMLR, 2019.

[26] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *arXiv preprint arXiv:1609.02907*, 2016.

[27] Xavier Bresson and Thomas Laurent. Residual gated graph convnets. In *arXiv preprint arXiv:1711.07553*, 2017.

[28] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of International Conference on Neural Information Processing Systems (NIPS)*, pages 1025–1035. ACM, 2017.

[29] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *arXiv preprint arXiv:1810.00826*, 2018.

[30] Petar Veličković. Message passing all the way up. In *arXiv preprint arXiv:2202.11097*, 2022.

[31] Kevin Kiningham, Philip Levis, and Christopher Ré. Grip: A graph neural network accelerator architecture. In *IEEE Transactions on Computers*, pages 914–925. IEEE, 2022.

[32] Xin Zheng, Yixin Liu, Shirui Pan, Miao Zhang, Di Jin, and Philip S Yu. Graph neural networks for graphs with heterophily: A survey. In *arXiv preprint arXiv:2202.07082*, 2022.

[33] Maciej Besta and Torsten Hoefler. Parallel and distributed graph neural networks: An in-depth concurrency analysis. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 2584–2606. IEEE, 2024.

[34] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of International Conference on Machine Learning-Volume 70 (ICML)*, pages 1263–1272. JMLR.org, 2017.

[35] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *arXiv preprint arXiv:1609.02907*, 2017.

[36] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 17–30. ACM, 2012.

[37] Fangzhou Ye, Lingxiang Yin, Amir Ahsaei Ghazizadeh, and Hao Zheng. Egma: Enhancing data reuse and workload balancing in message passing gnn acceleration via gram matrix optimization. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2024.

[38] Duane Merrill and Michael Garland. Merge-based parallel sparse matrix-vector multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 678–689. IEEE, 2016.

[39] Abbas Karimi, Kiarash Aghakhani, Seyed Ehsan Manavi, Faraneh Zarafshan, and SAR Al-Haddad. Introduction and analysis of optimal routing algorithm in benes networks. In *Procedia Computer Science*, pages 313–319. Elsevier, 2014.

[40] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. HyGCN: A GCN accelerator with hybrid architecture. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[41] Hao Zheng, Ke Wang, and Ahmed Louri. Adapt-noc: A flexible network-on-chip design for heterogeneous manycore architectures. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 723–735. IEEE, 2021.

[42] Lingxiang Yin, Amir Ghazizadeh Ahsaei, Ahmed Louri, and Hao Zheng. ARIES: Accelerating distributed training in chiplet-based systems via flexible interconnects. In *Proceedings of IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9. IEEE, 2023.

[43] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[44] Pengcheng Yao, Long Zheng, Yu Huang, Qinggang Wang, Chuangyi Gui, Zhen Zeng, Xiaofei Liao, Hai Jin, and Jingling Xue. ScalaGraph: A scalable accelerator for massively parallel graph processing. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 199–212. IEEE, 2022.

[45] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *Proceedings of USENIX Conference on Operating Systems Design and Implementation (OSDI)*, pages 301–316. ACM, 2016.

[46] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117. IEEE, 2015.

[47] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, pages 472–488. IEEE, 2013.

[48] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. Graphh: A processing-in-memory architecture for large-scale graph processing. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 640–653. IEEE, 2019.

[49] Andrea Lodi, Silvano Martello, and Daniele Vigo. Recent advances on two-dimensional bin packing problems. In *Discrete Applied Mathematics*, pages 379–396, 2002.

[50] Tayo Oguntebi and Kunle Olukotun. GraphOps: A dataflow library for graph analytics acceleration. In *Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. ACM, 2016.

[51] Hyoukjun Kwon, Prasanth Chatarasi, Michael Pellauer, Angshuman Parashar, Vivek Sarkar, and Tushar Krishna. Understanding reuse, performance, and hardware cost of DNN dataflow: A data-centric approach. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 754–768. IEEE, 2018.

[52] Raveesh Garg, Eric Qin, Francisco Munoz-Mart'inez, Robert Guirado, Akshay Jain, S. Abadal, Jos'e L. Abell'an, Manuel E. Acacio, Eduard Alarc'on, Sivasankaran Rajamanickam, and Tushar Krishna. Understanding the design space of sparse/dense multiphase dataflows for mapping graph neural networks on spatial accelerators. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 571–582. IEEE, 2021.

[53] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. In *IEEE Computer Architecture Letters*, pages 45–49, 2016.

[54] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. In *AI magazine*, pages 93–93, 2008.

[55] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam Hruschka, and Tom Mitchell. Toward an architecture for never-ending language learning. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, pages 1306–1313. AAAI Press, 2010.

[56] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. CACTI 6.0: A tool to understand large caches. In *University of Utah and Hewlett Packard Laboratories, Technical Report*, 2009.

[57] Haoran You, Tong Geng, Yongan Zhang, Ang Li, and Yingyan Lin. Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474. IEEE, 2022.

[58] Yu Huang, Long Zheng, Pengcheng Yao, Qinggang Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1029–1042. IEEE, 2022.

[59] Nagadastagiri Challapalle, Karthik Swaminathan, Nandhini Chandramoorthy, and Vijaykrishnan Narayanan. Crossbar based processing in memory accelerator architecture for graph convolutional networks. In *Proceedings of IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. IEEE, 2021.

[60] Yilong Guo, Yuxuan Chen, Xiaofeng Zou, Xulei Yang, and Yuandong Gu. Algorithms and architecture support of degree-based quantization for graph neural networks. In *Journal of Systems Architecture*, page 102578. Elsevier, 2022.

[61] Zeyu Zhu, Fanrong Li, Gang Li, Zejian Liu, Zitao Mo, Qinghao Hu, Xiaoyao Liang, and Jian Cheng. Mega: A memory-efficient gnn accelerator exploiting degree-aware mixed-precision quantization. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 124–138. IEEE, 2024.

[62] Xin Liu, Mingyu Yan, Shuhan Song, Zhengyang Lv, Wenming Li, Guangyu Sun, Xiaochun Ye, and Dongrui Fan. Gnnsampler: Bridging the gap between sampling algorithms of gnn and hardware. In *Proceedings of Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 498–514. Springer, 2022.

[63] Cen Chen, Kenli Li, Xiaofeng Zou, and Yangfan Li. Dygnn: Algorithm and architecture support of dynamic pruning for graph neural networks. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 1201–1206. IEEE, 2021.

[64] Deniz Gurevin, Mohsin Shan, Shaoyi Huang, MD Amit Hasan, Caiwen Ding, and Omer Khan. Prunegnn: Algorithm-architecture pruning framework for graph neural network acceleration. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 108–123. IEEE, 2024.

[65] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[66] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 319–333. IEEE, 2019.

[67] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[68] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 252–265. ACM, 2023.