OPT-GCN: A Unified and Scalable Chiplet-based Accelerator for High-Performance and Energy-Efficient GCN Computation

Yingnan Zhao, *Student Member, IEEE,* Ke Wang, *Member, IEEE,* and Ahmed Louri, *Fellow, IEEE*

***Abstract*—As the size of real-world graphs continues to grow at an exponential rate, performing the Graph Convolutional Network (GCN) inference efficiently is becoming increasingly challenging. Prior works that employ a unified computing engine with a predefined computation order lack the necessary flexibility and scalability to handle diverse input graph datasets. In this paper, we introduce OPT-GCN, a chiplet-based accelerator design that performs GCN inference efficiently while providing flexibility and scalability through an architecture-algorithm co-design. On the architecture side, the proposed design integrates a unified computing engine in each chiplet and an active interposer, both of which are adaptable to efficiently perform the GCN inference and facilitate data communication. On the algorithm side, we propose dynamic scheduling and mapping algorithms to optimize memory access and on-chip computations for diverse GCN applications. Experimental results show that the proposed design provides a memory access reduction by a factor of 11.3×, 3.4×, 1.4× energy savings of 15.2×, 3.7×, 1.6× on average compared to HyGCN, AWB-GCN, and GCNAX, respectively.**

***Index Terms*—Graph Convolutional Network, Hardware Accelerator, Chiplet-based Design, Hardware-algorithm Co-design**

I. Introduction

DEEP learning has demonstrated remarkable success in a wide range of applications that rely on structured inputs, such as vectors and images [1]–[7]. However, there has been a growing number of applications that rely on relational and irregular inputs, such as graphs. Therefore, Graph Convolutional Network (GCN) [8]–[16] has been proposed to extend deep learning to graph-related applications with the aim of achieving improved performance [17]–[23].

Typically, the computation of inference in each GCN layer includes two primary phases: *Aggregation* and *Combination* [12], [24]–[27]. The aggregation phase depends on the input graph structure, which is commonly sparse, resulting in irregular memory access and computation patterns. The combination phase operates similarly to that of conventional convolutional neural networks (CNN), leading to regular memory access and computation patterns. Different communication and computation patterns of two distinct GCN phases impose new requirements for the underlying hardware architecture. Unfortunately, the current deep neural network accelerators [28]–[30] are optimized specifically for either alleviating irregularity or exploiting regularity in isolation, which are inefficient for GCN inference.

Prior works have proposed several tailored GCN accelerators to deliver substantial gains in both performance and energy efficiency [25], [31], [32]. HyGCN [31] has implemented a tandem-engine architecture that utilizes two computing engines to perform the aggregation and combination phases in a sequential manner. Two engines are connected through an on-chip buffer to facilitate the storage of intermediate data. Instead of using two separate computing engines, both AWB-GCN [32] and GCNAX [25] modify the computation order of GCN inference to perform the combination phase followed by the aggregation phase, with a unified computing engine. However, with the existence of diverse input datasets, prior works are limited in their flexibility to perform two distinct GCN phases in a dynamic sequence. This results in additional memory access and on-chip computations. Both data memory access and on-chip computations are crucial factors that have the potential to significantly improve overall performance and energy efficiency. Furthermore, the exponential expansion of real-world datasets has made it challenging to scale prior works, as the two distinct GCN phases demand different hardware requirements, resulting in significant costs.

To address the aforementioned challenges, we propose OPT-GCN in this paper, a chiplet-based hardware accelerator design that effectively executes GCN inference while providing flexibility and scalability through a collaborative architecture-algorithm co-design approach. At the architecture level, the proposed design implements a unified computing engine in each chiplet and an active interposer, both of which are adaptable and effectively meet the computational requirements of both GCN phases. At the algorithm level, the proposed design includes dynamic scheduling and mapping algorithms to improve performance and energy efficiency. The main contributions of this paper are detailed as follows:

- We propose a novel chiplet design, featuring a unified computing engine and employing heterogeneous on-chip dataflow to adaptively facilitate GCN inference. The unified computing engine can be configured to efficiently support the required computations of both the aggregation and combination phases. The heterogeneous on-chip dataflow is responsible for improving the data reuse during each GCN phase.
- We propose an active interposer fabric for efficient data communication between chiplets and the global buffer (GLB). Such an interposer design can be dynamically

Yingnan Zhao and Ahmed Louri are with George Washington University, Washington, DC, 20052. Email: {yzhao96, louri}@gwu.edu

Ke Wang is with University of North Carolina at Charlotte, Charlotte, NC 28223. E-mail: ke.wang@uncc.edu.

reconfigured to adapt to diverse GCN workload patterns.
- We also propose a dynamic workload scheduling algorithm (WSA), which is based on the input graph dataset and the applied GCN model. This algorithm is performed at the application level with the objective of determining the workload pattern that optimizes memory access and on-chip computations during both phases of GCN inference.
- We finally propose a workload mapping algorithm (WMA) based on the analysis of previously scheduled workload patterns and the hardware architecture configuration. This algorithm is executed at the tile level to effectively map scheduled workloads to the underlying hardware accelerator with the aim of improving performance and energy efficiency.

Experimental results with real-world graph datasets show that the proposed chiplet-based design provides a memory access reduction by a factor of 11.3×, 3.4×, 1.4× while achieving a speedup of 16.0×, 2.9×, 1.8× and energy savings of 15.2×, 3.7×, 1.6× on average compared to HyGCN, AWB-GCN, and GCNAX, respectively.

II. Background and Motivation

A. GCN Background

During the GCN inference, the feature vector of each vertex in the input graph dataset is updated by recursively aggregating and transforming the representation vectors of its neighboring vertices [24], [25], [32]–[34]. From the perspective of linear algebra, the layer-wise forward propagation of the GCN model can be formulated as Eq 1. Where A means the normalized graph adjacency matrix. $X^{(k)}$ is the feature matrix of the layer-k and $W^{(k)}$ is the weight matrix of layer-k. $\sigma(.)$ represents the non-linear activation functions [8], [12], [35].

$$X^{(k+1)} = \sigma(AX^{(k)}W^{(k)}) \tag{1}$$

Fig. 1 shows the main execution phases in GCN models, which are named aggregation and combination, respectively. In the aggregation phase, each vertex recursively collects features from its neighbors. However, as the neighbors of each vertex are not continuously stored in memory, this results in irregular memory access when performing the aggregation phase. During the combination phase, vertices use a pre-trained weight matrix to update their feature vectors. This process involves regular memory access for each vertex, as both the feature vector and the weight matrix are stored contiguously in memory. As a result, the two distinct GCN phases necessitate varying hardware requirements based on their communication patterns.

Additionally, concerning the input graph structure, some vertices display a larger number of neighbors than others, a unique characteristic known as the power-law distribution [36]–[42]. As shown in Eq. 2, the P(d) means the probability that a vertex has a degree d, and the exponent α indicates the skewness of the distribution. Given a predefined threshold, all vertices of the input graph can be divided into two categories: high-degree (HD) and low-degree (LD) vertices.

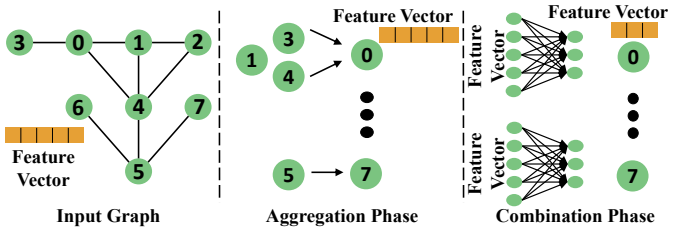


Fig. 1. Illustration of the GCN model with two main computation phases: Aggregation and Combination.

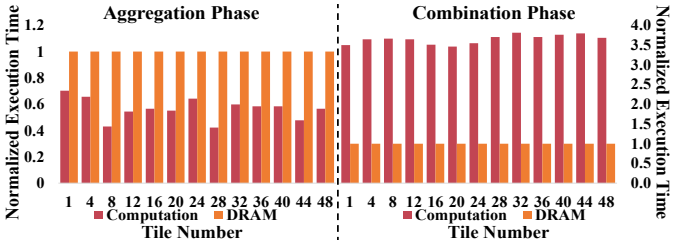


Fig. 2. Time distribution of each tile's memory access (DRAM) and on-chip computations for both the aggregation and combination phases. The time of computations is normalized to that of DRAM access.

While both types of vertices follow the same computational pattern during the combination phase, their computations differ during the aggregation phase due to variations in the number of neighbors they process.

$$P(d) = d^{\alpha} \tag{2}$$

The performance of each GCN phase is limited by different hardware resources owing to variations in memory access and computation patterns. For instance, as shown in Fig. 2, with several data chunks (tile) in the Cora [8], [43]–[45] (a real-world graph dataset), either data memory access or on-chip computations dominate the performance during the aggregation and combination phases, respectively. Therefore, using the entire architecture to perform either the aggregation or the combination phase induces hardware underutilization, thereby degrading performance and energy efficiency. To illustrate this issue, we evaluate three state-of-the-art solutions (HyGCN, AWB-GCN, and GCNAX) with the Cora dataset, where the hardware resource utilization averages 67%. As a result, it is crucial to design a GCN hardware accelerator that can effectively handle diverse workload patterns and hardware requirements to achieve high performance and energy efficiency during GCN inference.

B. Previous Works and Motivation

While several customized GCN accelerators have been proposed to enhance performance and reduce energy consumption in GCN inference [25], [31], [32], [46], with the existence of diverse input datasets and GCN models, current approaches are limited in their ability to provide flexible computation order for each GCN layer and dynamically allocate hardware resources based on specific requirements. Specifically, HyGCN [31] implements a tandem-engine architecture design, with two separate engines connected by an on-chip buffer, that can only perform the aggregation phase at first followed by the combination for all input vertices during each GCN layer. Both

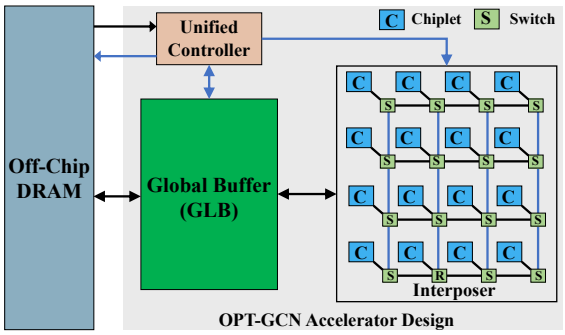


Fig. 3. Chiplet-based GCN accelerator design with N chiplets (N=16 in this example). The unified controller is used for implementing the proposed algorithms. The applied interposer facilitates data communication between chiplets, as well as between chiplets and the Global Buffer (GLB). Each chiplet features a unified computing engine that can perform computations for both GCN phases. Blue and black links represent the transmission of control signals and data, respectively.

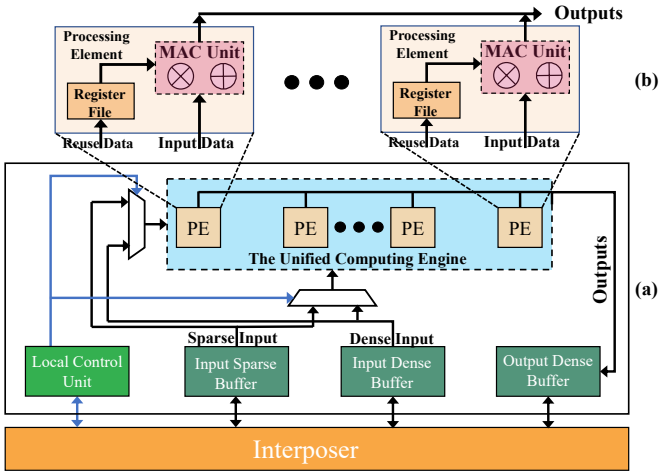


Fig. 4. (a) Chiplet GCN accelerator architecture design overview: The local control unit is used to configure the value of MUXes. MUXes are utilized to switch input buffers for a heterogeneous on-chip dataflow. The input sparse and dense buffers are used to store sparse and dense matrices respectively. The unified computing engine includes a 1×16 modified Processing Element (PE) array. (b) The modified Processing Element (PE) design with a register to store data locally and a MAC unit to perform required GCN computations. The blue and black links are used to transmit control signals and required data, respectively.

AWB-GCN [32] and GCNAX [25] deploy a unified computing engine to initially perform the combination phase followed by the aggregation phase during each GCN layer. While EnGN offers the flexibility to perform either the aggregation or the combination phase first [46], once the computation order is determined, all vertices experience the fixed computation order across all GCN layers. Consequently, none of the aforementioned works can optimize the computation order for each GCN layer, especially when considering the substantial variation in the degree of input vertices. Additionally, after determining the computation order, current approaches utilize the entire computing engine to sequentially perform one phase at a time. These approaches lack dynamic allocation of hardware resources based on the requirements of the input workload, consequently affecting overall hardware utilization. Furthermore, the exponential expansion of real-world datasets poses challenges to the scalability of prior works, as the two distinct phases demand different hardware requirements, resulting in significant complexity in architecture design.

III. Chiplet-based GCN Accelerator Design

In this paper, we propose OPT-GCN, a scalable chiplet-based GCN accelerator design that effectively addresses the aforementioned challenges through an architecture-algorithm co-design approach. OPT-GCN aims to accelerate GCN inference while ensuring accuracy is not compromised. The subsequent sections provide a comprehensive overview encompassing both architectural and algorithmic aspects in detail.

A. Architecture Design

1) Overall Architecture Design: First of all, we present the overall architecture design as shown in Fig. 3. The proposed architecture includes a unified controller, a global buffer (GLB), an active silicon interposer, and N chiplets (N=16 in this example) that work independently. The unified controller is connected to the main memory and the interposer layer. The dimensions of both the input dataset and the applied GCN model are directly loaded from the main memory to the unified controller, enabling the execution of the proposed algorithms. Additional details regarding the function of the proposed algorithms are provided in Sec. III-B. The details of the unified controller design are shown in Sec. III-A3. The global buffer (GLB) is made up of a buffer controller and several First-In-First-Out (FIFO) buffers that are separated into two chunks to store HD and LD vertices according to the power-law distribution introduced in Sec. II-A, respectively. The proposed design applies the active silicon interposer instead of the passive one to support the communication between chiplets, such as the weight matrix. The proposed design includes N (N=16 in this paper) separate chiplets. Each chiplet is deployed to independently perform a specific type of GCN computational task, either the aggregation or the combination phase. Additionally, due to the existence of sparsity, each chiplet processes a specific number of input vertices, without sharing intermediate data with others directly. Details of the proposed chiplet design are illustrated in Sec. III-A2.

2) Chiplet Design: Compared to previous works, the novel chiplet design presented in this paper offer flexibility and can be configured to efficiently perform computations of both aggregation and combination phases. As shown in Fig. 4, each chiplet includes a local control unit, input sparse and dense matrix buffers, an output dense buffer, a pair of MUXes, and a unified computing engine. The local control unit is connected to the interposer to receive the control signal from the unified controller. Based on the received signal, MUXes in each chiplet will be configured to switch the input sparse and dense buffers. This is because the chiplet applies a heterogeneous on-chip dataflow when performing computations of both GCN phases. The dataflow is implemented when mapping the data from the on-chip buffer to Processing Elements (PEs) for required computations in the spatial GCN accelerator domain. The choice of dataflow reveals the reuse type of data operands (input, weight, or output) over space and time. For the aggregation phase, the feature vector is pre-loaded from the input dense buffer to the register of each PE and stored locally.

The adjacency matrix (A) in Compressed Sparse Column (CSC) [47] format is streamed from the input sparse buffer to the PE array for computations. During the combination phase, the weight matrix is pre-loaded from the input dense buffer to registers of PEs, and the feature vector is streamed from the input sparse buffer to the computing engine. For the on-chip buffers, the Input Sparse Buffer (ISB) is used to store adjacency (A) and feature (X) matrices. The Input Dense Buffer (IDB) is used to store the feature (X), the pre-trained weight (W), and the intermediate (B) matrices. The Output Dense Buffer (ODB) is used to store the intermediate (B) and final (O) matrices. Here we assume that the result of a sparse-sparse or sparse-dense matrix-matrix multiplication (SpGEMM or SpMM) is a dense matrix [25], [48], which can be appropriately stored in the ODB. The unified computing engine consists of a 1×16 modified PE array. Fig. 4 (b) depicts the architectural details of the modified PE design, comprising a register that stores the pre-loaded data locally and a multiply-accumulate (MAC) unit to perform required GCN computations.

3) Unified Controller Design: The main functionality of the unified controller is to perform the proposed algorithms and inform chiplets about the type of upcoming workload. The unified controller includes multiple registers and arithmetic-logic units (ALUs), and it is connected to the main memory and interposer. Specifically, before performing the current GCN layer, the unified controller retrieves required parameters, such as sparsity, from the main memory to facilitate the execution of the proposed algorithms. These parameters are stored within local registers (32-bit). Subsequently, these parameters are forwarded to local ALUs to carry out the required computations according to the algorithm's specifications. The computation process involves two primary steps: In Step-1, the unified controller utilizes graph information, such as sparsity, to determine the computation order pattern, as introduced in Sec. III-B2. In Step-2, hardware information, such as the number of chiplets, is used to configure each chiplet based on the previously determined pattern, as depicted in Sec. III-B3. After performing the proposed algorithms through the local ALUs, the unified controller will send two types of control signals to each chiplet for further configuration. The function of the control signal (1-bit) is to notify each chiplet about the type of upcoming workload. More precisely, the value of **0** denotes a workload in the aggregation phase, whereas the value of **1** signifies a workload in the combination phase. Additionally, the unified controller tunes interposer to configure connections between chiplets, enabling efficient data communication.

4) Interposer Design: The silicon interposer serves the purpose of connecting chiplets and GLB, and it can be categorized into two types: passive and active interposers. Aside from limited wiring, most of the interposer area and wiring resources are underutilized for the passive interposer, though they have been paid. Therefore, instead of using the passive design, this paper applied the active interposer to provide better performance and economics. All switches in the interposer layer are connected to each other through a Network-on-Chip (NoC) design for data communication, as shown in Fig. 3. The blue links are used to transfer the control signal, and the black links are used to transfer the required data. After performing the proposed algorithms and finishing the required configuration of the interposer layer, each chiplet is assigned to independently and simultaneously perform computations for a specific GCN workload. Details of the data communication are illustrated in the following section.

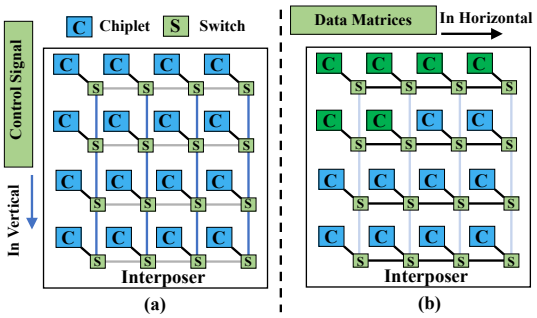


Fig. 5. With N (N=16 in this example) chiplets, (a) the configuration setup of each switch. (b) the workload distribution after the configuration setup. Those chiplets in blue are assigned to the aggregation workload and others are responsible for the combination workload. Blue and black links are used for transferring control signals and data, respectively.

5) Inter-chiplet Communication: In terms of data communication, every chiplet employs its respective switch to establish communication with the GLB and other chiplets for data loading and sharing. Before each chiplet engages in data loading and computations, the unified controller executes the proposed algorithms and transmits the configuration information of each chiplet to the switches within the same column for additional configuration, as illustrated in Fig. 3. The configuration information comprises a bit for every switch in the corresponding column, indicating whether it shares common data with neighboring switches in the same row. Rather than employing a mesh topology to connect all switches, switches within the same row or column are interconnected using a ring topology. Additionally, in the proposed design, vertical links are utilized for configuration setup, while horizontal links are designated for data communication. In cases where chiplets share common data with their neighboring counterparts, such as the combination phase, switches will transmit the data to the corresponding chiplet and the other adjacent switch. Following the completion of the necessary configuration setup, the loaded tiles are sequentially transmitted from the Global Buffer (GLB) to the respective chiplet through the switches in the interposer layer. In the interposer layer, only those horizontal links are active when there is shared data (weight matrix) between two chiplets in the same row. On the other hand, the chiplets assigned to the aggregation phase operate independently and concurrently. Applying this communication strategy offers several advantages. Firstly, it helps prevent data conflicts that can occur as the size of chiplets increases. Additionally, all switches can be easily configured through the unified controller before performing required computations, ensuring efficient communication and synchronization. Furthermore, this strategy enhances the reuse of the weight matrix, resulting in reduced memory access requirements.

Assume that the number of chiplets in the proposed design is 16, as depicted in Fig. 5. After performing the proposed

algorithms, the first six chiplets are designated for the combination workload, while the remaining chiplets are assigned to the aggregation workload. In this scenario, the switches in the first row will be interconnected to share the weight matrix, and the first two switches of the second row will also be configured to connect with each other. Furthermore, the proposed design exclusively facilitates data sharing among chiplets within the same row. In the case of chiplets located in different rows, even if both are assigned to execute a workload associated with the combination phase, they are prohibited from sharing data in order to prevent data blocking. Simultaneously, the switches of the chiplets designated for the aggregation workload will operate independently, without sharing data among them. After the configuration setup is completed, each chiplet will sequentially load data (such as adjacency, feature, and weight matrix) from the GLB for subsequent computations.

B. Algorithm Design

In this section, we begin by analyzing the flexible workload pattern of GCN inference proposed in this paper from an algorithmic perspective. Subsequently, we introduce a dynamic workload scheduling algorithm that determines the workload pattern based on the input dataset and the applied GCN model, which is operated at the graph level to optimize memory access and computations. Then, we present a mapping algorithm that works on the tile level, which effectively maps workload patterns to underlying hardware for improved performance and efficiency.

1) GCN Workload Pattern Analysis: The primary computational workload of GCN inference can be classified into two categories: firstly, vector-matrix multiplication that occurs during the aggregation phase, and secondly, matrix-matrix multiplication that occurs during the combination phase. Fig. 6 (a) and (b) illustrate the algorithm-level details of GCN matrix-matrix multiplication for two distinct workload patterns, where $< T_{M0}, T_{N0}, T_{C0}, T_{K0} >$ is the tile size tuple. Both T_{M0} and T_{N0} are equal to the number of vertices in the given graph. This paper utilizes two distinct parameters to represent the tile dimensions of the adjacency matrix (A) and feature matrix (X), respectively. The tile size determines the amount of data stored in the on-chip buffer during each epoch. Additionally, the tile size impacts the reuse time of data in each input matrix during matrix-matrix multiplication. Since different input graphs with distinct GCN models have preferable tile sizes, supporting various tile sizes increases the complexity of the hardware design [25], [27]. Therefore, this paper applies a pre-determined tile size $< 2048, 2048, 16, 16 >$ for all input graphs. The values of T_K and T_C are set to match the number of Processing Elements (PEs) within each chiplet so that the data elements of tiles X and W can be directly streamed from the local buffer to the corresponding PE without requiring additional partitioning. The value of T_N is configured to strike a balance between buffer utilization and execution time with limited buffer size.

During the computation of each phase, the loop tiling can be applied to leverage data locality. When performing each *for* loop with applying different unrolling techniques, distinct

Given three input matrices and one output matrix:
$A \in R^{T_{M0} \times T_{N0}}$, $X \in R^{T_{N0} \times T_{K0}}$, $W \in R^{T_{K0} \times T_{C0}}$, and $O \in R^{T_{M0} \times T_{C0}}$

```
// Aggregation phase
for (n = 0; n < TN0; n++) {
  for (k = 0; k < TK0; k++) {
    for (m = 0; m < TM0; m++) {
      B[m][k] += A[m][n] * X[n][k];
    }}}
// Combination phase
for (m = 0; m < TM0; m++) {
  for (k = 0; k < TK0; k++) {
    for (c = 0; c < TC0; c++) {
      O[m][c] += B[m][k] * W[k][c];
    }}}
```

(a) Aggregation->Combination

```
// Combination phase
for (n = 0; n < TN0; n++) {
  for (c = 0; c < TC0; c++) {
    for (k = 0; k < TK0; k++) {
      B[n][c] += X[n][k] * W[k][c];
    }}}
// Aggregation phase
for (m = 0; m < TM0; m++) {
  for (c = 0; c < TC0; c++) {
    for (n = 0; n < TN0; n++) {
      O[m][c] += A[m][n] * B[n][c];
    }}}
```

(b) Combination->Aggregation

Fig. 6. Computation order of GCN matrix multiplication from algorithm level: (a) Performing the aggregation and combination phases sequentially. (b) Performing the combination and aggregation phases sequentially. A, X, W, B, and O represent the Adjacency matrix, Feature matrix, Weight matrix, Intermediate matrix, and Final matrix, respectively.

sets of input/output data are reused during the entire GCN process. Take the Inner product as an example, as shown in Fig. 7 (a), there are two input matrices (A$\in R^{(M \times N)}$ and B$\in R^{(N \times K)}$) and one output matrix (O$\in R^{(K \times C)}$) for performing the matrix-matrix multiplication. Obviously, each element of the output matrix is reused and accumulated when performing the inner *for* loop, while the index of elements from both A and B matrices continues to increase. Despite the fact that the inner product loop unrolling technique utilizes output data locality, it can result in inefficiency and degrade overall performance. This is due to the varying number of inputs each Processing Element (PE) in the hardware receives as a result of sparsity, which causes workload imbalance. Concerning the row-based product, as illustrated in Fig. 7 (c), multiple rows of matrix B are necessary to access multiple times since they are shared for elements in the same column of matrix A. This results in additional memory access, particularly when considering the matrices A and B are sparse and dense, respectively, for performing Sparse-Dense Matrix-Matrix Multiplication. Therefore, the proposed design applies the outer product to reuse one of the input matrices during each GCN phase. Although this method sacrifices the reuse of the output matrix, it well supports the elimination of zero computations and avoids the workload imbalance problem because of the sparsity [25], [32]. All the input sparse matrices are stored in Compressed Sparse Column (CSC) format, while the input dense matrix is stored in a dense format in row-major order. As shown in Fig. 7 (b), since the A[i][j] is reused by all elements in the first row of the B matrix, these computations can be eliminated if A[i][j] is zero. Note that it is unnecessary to design and implement the "compare" logic in the hardware design to detect whether the current A[i][j] is zero, as the CSC format has already eliminated the zeros when compressing and storing the data.

2) Workload Scheduling Algorithm (WSA): Given an input graph with the applied GCN model, current approaches apply a fixed GCN computation order across all GCN layers with their customized hardware design, which is insufficient for various input graphs and applied GCN models. Specifically, given an input dataset (A$\in R^{(M \times N)}$ and X$\in R^{(N \times K)}$) and a weight matrix (W$\in R^{(K \times C)}$), existing designs perform either the aggregation phase or the combination phase

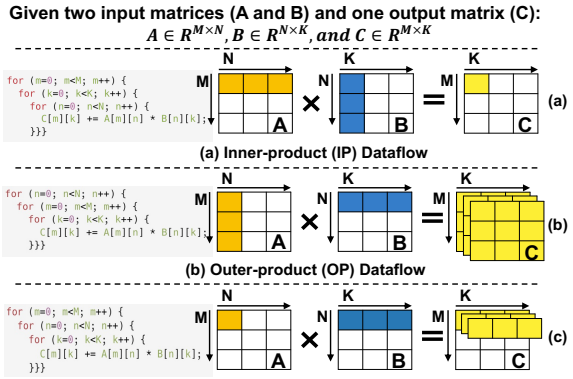


Fig. 7. Loop unrolling techniques: (a) Inner-product: The inner *for* loop produces an entire element of the output matrix, and the output matrix (O) is reused. (b) Outer-product: The inner *for* loop produces partial sums of one row of elements in the final output matrix, and the input matrix (A) is reused. (c) Row-product: The dense rows of matrix B are reused.

at first in each GCN layer. Under this circumstance, the length of each feature vector remains constant during the aggregation phase, while it changes from dimension **K** to dimension **C** during the combination phase. Hence, applying a fixed computation order for all GCN layers across various input graphs is inefficient. This efficiency arises due to the potential increase in feature-length during the combination phase, leading to additional memory access and computations for the subsequent phase within the same GCN layer. To address the aforementioned challenge, the proposed Workload Scheduling Algorithm (WSA) dynamically determines the computation order for each input vertex from the input graph during each GCN layer. This determination takes into account the parameters of both the input datasets and the applied GCN models, such as dimensions, sparsity, and power-law distribution. The primary objective of the proposed WSA is to minimize both data memory access and the overall execution time of GCN inference. In other words, a single vertex may experience a different computation order in distinct GCN layers, and vertices may undergo distinct GCN computation orders within the same GCN layer. Particularly, considering the parameters of the input graph and the applied GCN model as $< N_0, K_0, C_0 >$ with sparsity S_A, Eq. 3 and Eq. 4 illustrate the computational requirements for distinct computation orders, where the aggregation phase (case-1) and combination phase (case-2) are executed first, respectively. In this context, N_0, K_0, and C_0 represent the number of input vertices, the length of features, and the dimensions of the weight matrix, respectively.

$$\begin{aligned} Op_{case-1} = &\prod \mathrm{N}_0, K_0, N_0, S_A + \prod \mathrm{N}_0, K_0, N_0 S_A - 1 \\ &+ \prod \mathrm{N}_0, C_0, K_0 + \prod \mathrm{N}_0, C_0, K_0 - 1 \end{aligned} \tag{3}$$

$$\begin{aligned} Op_{case-2} = &\prod \mathrm{N}_0, C_0, K_0 + \prod \mathrm{N}_0, C_0, K_0 - 1 \\ &+ \prod \mathrm{N}_0, C_0, N_0, S_A + \prod \mathrm{N}_0, C_0, N_0 S_A - 1 \end{aligned} \tag{4}$$

$$\begin{aligned} Op_{diff} &= Op_{case-1} - Op_{case-2} \\ &= {N_0}^2 S_A (K_0 - C_0) + N_0 (N_0 S_A - 1)(K_0 - C_0) \\ &= N_0 (K_0 - C_0)(N_0 S_A + N_0^2 S_A - N_0) \end{aligned} \tag{5}$$

With Eq. 3 and Eq. 4, Eq. 5 demonstrates the discrepancy in the number of required computations between two different

Algorithm 1 Workload Scheduling Algorithm (WSA)

Inputs: S_A: Sparsity of the input graph; N_0: Number of input vertices; K_0: Length of feature vector; C_0: Dimension of the pre-trained weight matrix; D_T: Pre-defined degree threshold; offset_array: The column offset array of the adjacency matrix that stored in CSC format;
Outputs: Workload pattern and related control signals
Begin:
Initialize: pStart =0, pEnd = 1, degree = 0, signal = 0;
Initialize: H = V×P, L = V×(1-P), variation = 0;
// dynamically decide workload pattern
variation = $\prod N_0, (K_0 - C_0), (N_0 S_A + N_0^2 S_A - N_0)$;
if variation ≤ 0 **then**
    // LD vertices perform the combination phase first
    signal = 1;
else
    // LD vertices perform the aggregation phase first
    signal = 0;
end if
//load vertex (n)
while $n \leq N_0$ **do**
    // do parallel
    // Calculate the degree of the current vertex (n)
    degree = offset_array[pEnd]-offset_array[pStart];
    if degree $\leq D_T$ and signal == 1 **then**
        workload pattern (n) = A(XW);
    else
        workload pattern (n) = (AX)W;
    end if
    pStart = pEnd, pEnd++;
end while

computation orders. According to the previous work [25], the density of matrix A typically does not exceed 0.21%, which means that the value of the third part $(N_0 S_A + N_0^2 S_A - N_0)$ in Eq. 5 is always positive. Consequently, the relationship between parameters K_0 and C_0 predominantly determines the computation order for the given input matrices (A, X, and W). If K_0 is bigger than C_0, performing the combination phase first can reduce the length of the feature from K_0 to C_0, and vice versa. With these formulations, the proposed WSA dynamically schedules the computation order for each input vertex during each GCN layer based on its degree. The rationale behind this is that using the entire architecture to perform each GCN phase at a time results in inefficiencies in either communication or computation, as discussed in Sec. II-A. Given a pre-defined threshold, the vertices of the input graph can be classified into two types: High-degree vertices (HD) and Low-degree vertices (LD). As the LD vertices occupy the majority of the input vertices, WSA schedules the workload from the perspective of LD vertices in order to optimize the entire GCN inference. Each LD vertex has two options to choose from: either perform the aggregation phase or the combination phase at first. Therefore, by using Eq. 5 along with the input dimensions, the workload of each input LD vertex can be determined easily using WSA. The HD vertices are scheduled with a different workload pattern compared to the LD vertices simultaneously. This strategy provides two advantages: firstly, it reduces memory access and computations by significantly reducing the feature vector length for the majority of vertices, as mentioned earlier; secondly, it improves the efficiency of the hardware architecture by performing different GCN phases

on different chiplets simultaneously, as previously introduced in Sec. II-A. The following pseudo-code illustrates details about how the proposed WSA schedules the workload for each input vertex dynamically. Specifically, WSA utilizes two pointers (pStart and pEnd) with an offset array derived from the CSC format to calculate the *degree* of the current input vertex. The parameter *variation* in line 5 is used to record the difference in the number of computations when initially performing the aggregation and the combination for the input LD vertex in the current GCN layer. The parameter *signal* is employed to indicate the computation order for the subsequent LD vertex in the current GCN layer. After determining the value of the signal (lines 6-14), the computation order of each vertex within the loaded tile is established and prepared for subsequent processing (lines 15-24).

3) Workload Mapping Algorithm (WMA): WMA operates at the tile level and is specifically designed to determine the number of chiplets assigned to each GCN phase and map the input tiles to the corresponding chiplets for computations. The following pseudo-code illustrates how the proposed WMA works in detail. Specifically, as described in Sec. III-B2, some of the loaded vertices perform the aggregation phase concurrently with others performing the combination phase. The WSA algorithm takes parameters such as the bandwidth (B), the size of the pre-defined tile ($< T_n, T_k, T_c >$), the generated workload pattern, and the structure of the input graph (S_A), as parameters to complete its functionality. Given $\boldsymbol{C}$ chiplets in total, each of them is responsible for performing computations of either the aggregation phase or the combination phase with the pre-defined tile size and limited bandwidth. During each epoch, there are (C$\times$ T_n) vertices' features loaded from the main memory for calculations. Due to sparsity, the number of LD vertices in the total loaded vertices varies. To illustrate the proposed WMA and facilitate understanding, we assume that during each epoch, there are **L** LD vertices and **H** HD vertices loaded, where the sum of H and L equals C $\times$ T_N. Given a tile size tuple ($< T_n, T_k, T_c >$), Eq. 6 and Eq. 7 illustrate the execution time (in cycles) of a single chip performing the aggregation phase or the combination phase, respectively. Particularly, the key difference between the two equations is that the aggregation phase includes a write-back time for storing the intermediate matrix B (B$\in R^{(N\times K)}$). This is because the computations dominate the performance of the combination phase as described in Sec. II-A.

$$T_{agg.} = \frac{{T_n}^2 S_A}{B} + \frac{T_n T_k}{B} + \frac{T_n T_k}{No.(PEs)} + \frac{T_n T_k}{B} \tag{6}$$

$$T_{comb.} = \frac{T_n T_k}{B} + \frac{T_k T_c}{B} + \frac{T_n T_c}{No.(PEs)} \tag{7}$$

$$\frac{No.(Chiplet_{agg.})}{No.(Chiplet_{comb.})} \approx \frac{T_{comb.}}{T_{agg.}} \tag{8}$$

To improve the efficiency and the overall performance, the execution time of chiplets that perform the combination phase must be equal to or greater than those performing the aggregation phase, as indicated in Eq. 8. Here, $Chiplet_{agg.}$ and $Chiplet_{comb.}$ represent the number of chiplets assigned to the aggregation and combination phases, respectively. Additionally, the sum of these two parameters equals the total number of chiplets the hardware has. After determining the number of chiplets assigned to the aggregation and combination phases, the system will sequentially map the workload to each chiplet to avoid data conflict during computation (lines 10-17). Before mapping, the unified controller sends the corresponding configuration information to each chiplet. However, in scenarios where the architecture has a limited number of chiplets, there may arise a corner case where no chiplet is accessible to carry out the combination phase after performing the Eq. 8. In this particular scenario, the architecture will choose one of the chiplets to perform the combination phase directly (lines 18-20).

Algorithm 2 Workload Mapping Algorithm (WMA)

1: **Inputs:** $< T_{n0}, T_{k0}, T_{c0} >$: Pre-defined tile size; S_A: Sparsity of loaded tile; N: Number of PEs; C: Number of Chiplets; B: Bandwidth between GLB and each chiplet.
2: **Outputs:** Each chiplet is assigned a specific input workload
3: **Begin:**
4: Initialize: $T_{agg.} = 0, T_{comb.} = 0, S_A = 0, flag = false$;
5: Initialize: pStart = 0, pEnd = N $\times$ $T_{n0}, tmp = 0$;
6: $S_A = (pEnd - pStart)/(T_{n0} \times T_{n0})$;
7: $T_{agg.}$= $\frac{{T_n}^2 S_A}{B} + \frac{T_n T_k}{B} + \frac{T_n T_k}{No.(PEs)} + \frac{T_n T_k}{B}$;
8: $T_{comb.}$ = $\frac{T_n T_k}{B} + \frac{T_k T_c}{B} + \frac{T_n T_c}{No.(PEs)}$;
9: tmp = $T_{comb.}/T_{agg.}$;
10: **for** each $i \in C$ **do**
11: **if** $i\%tmp \equiv 0$ **then**
12: current chiplet performs Aggregation;
13: **else**
14: current chiplet performs Combination;
15: flag = true;
16: **end if**
17: **end for**
18: **if** $flag \equiv false$ **then**
19: Chiplet-0 performs the Combination (Corner Case);
20: **end if**

C. Workflow Example

In order to clearly illustrate the proposed design, this paper presents a detailed five-step procedure with an example workflow in this subsection. For a given graph dataset and a GCN model with three input matrices: (A$\in R^{(N\times N)}$ and X$\in R^{(N\times K)}$) and a weight matrix (W$\in R^{(K\times C)}$), each step is illustrated as the following:

- **Step 1:** Dimensions of input matrices (A, X, and W), as well as the sparsity (S_A), are loaded from the main memory to the registers of the unified controller through dedicated data links, as shown in Fig. 3. Simultaneously, the feature matrix data will be loaded from the main memory to the GLB as both GCN phases require it. Additionally, the vertices of the input graph dataset loaded from the main memory to GLB can be classified into two categories based on a predefined threshold determined by the power-law distribution, as introduced in Sec. II-A.
- **Step 2:** Once the necessary data is received, the unified controller employs its local ALUs to perform the WSA algorithm and determine the dynamic workload pattern for each type (HD or LD) vertex, separately. When the calculation is finished, the unified controller sends a corresponding signal to the main memory controller, initiating the loading of necessary data, such as adjacency or weight matrices, for each type of vertex based on

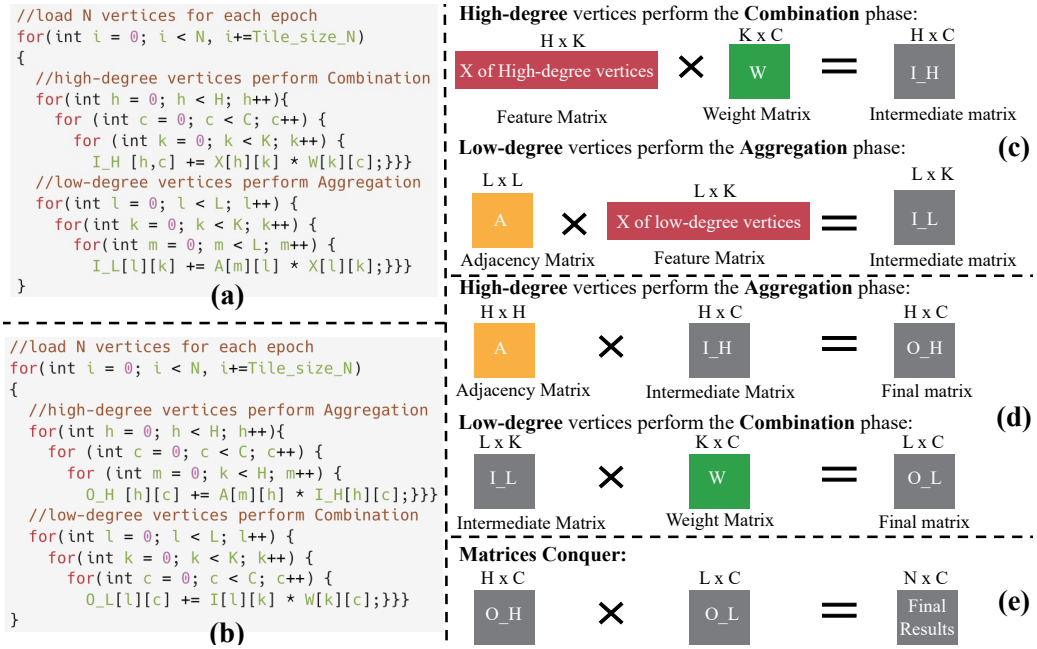


Fig. 8. The partition and conquer of matrices multiplication. Analysis from the algorithm level for (a) High-degree vertices perform Combination and low-degree vertices perform Aggregation. (b) High-degree vertices perform Aggregation and low-degree vertices perform Combination. (c) High-degree vertices perform the Combination and low-degree vertices perform Aggregation simultaneously. (d) High-degree vertices perform the Aggregation and low-degree vertices perform the Combination simultaneously. (e) Matrices of High/low-degree vertices are conquered to receive the final result.

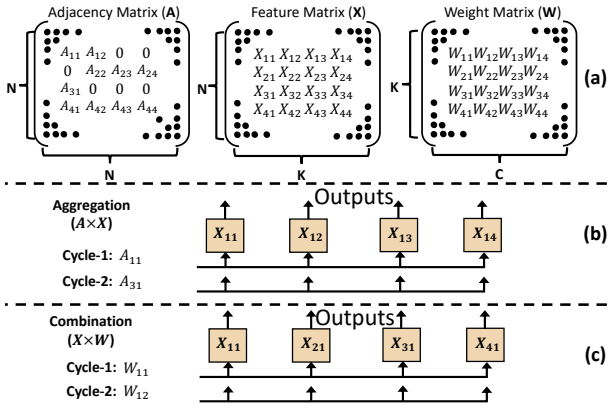


Fig. 9. Data workflow of the proposed chiplet design. (a) Three input matrices (A∈ $R^{(N\times N)}$, X∈ $R^{(N\times K)}$, and W∈ $R^{(K\times C)}$) of input graphs. (b) For the aggregation phase, values of the input dense buffer (Feature vector-X) are pre-loaded, while values of the input sparse buffer (Adjacency list-A) are broadcast in CSC format. (c) For the combination phase, values of the input sparse buffer (Feature vector-X) are pre-loaded, while values of the input dense buffer (Weight matrix-W) are broadcast.

the dynamic workload pattern. Due to the large data chunk of the feature matrix, loading it from the main memory to the GLB takes longer than performing the WSA algorithm. Since loading the feature matrix and performing the algorithm are processed concurrently, the time required for performing the WSA and transmitting control signals is negligible and can be disregarded.

- **Step 3:** Following the determination of the workload pattern at the application level, the unified controller performs WMA to ascertain the workload of each chiplet. Subsequently, the controller transmits the relevant control signal to each chiplet to facilitate further reconfiguration. Based on the received signal, the local control unit in each chiplet is responsible for configuring the local MUXes to switch the input sparse/dense buffer. This is because a heterogeneous dataflow is applied when performing the two distinct GCN phases with the aim of maximizing data reuse. After the configuration process, each chiplet initiates the loading of data and uses the local unified computing engine to perform the required vector-matrix (aggregation) or matrix-matrix (combination) multiplications. Fig 8 (a) and (b) illustrate the execution of the algorithm at the vertex level, where high-degree (HD) vertices perform the combination phase, and low-degree (LD) vertices perform the aggregation phase simultaneously. Fig 8 (c) and (d) show how the dimensions of each input matrix changed during the entire process.
- **Step 4:** During the computation process, chiplets utilize distinct preloaded dataflow strategies for each of the current GCN phases. In the aggregation phase, the values of the input dense buffer (X) are loaded into the register of each PE and stored locally. In the combination phase, the values of the input sparse buffer (X) are loaded and stored locally within each PE, as shown in Fig 9 (b) and (c), respectively. The other matrix, which is used for required matrix multiplication, is broadcast to all PEs in the computing engine on a cycling basis during both GCN phases. During the aggregation phase, the entire PE array is used to update one vertex's features. During the combination phase, each PE is responsible for updating one specific vertex's features.
- **Step 5:** After completing the required computations of both GCN phases within each chiplet, the resultant matrices are subsequently stored in the main memory,

TABLE I
DETAILS OF THE 2-LAYER GCN DATASET USED FOR EVALUATION

Datasets	Cora	Citeseer	Pubmed	Nell	Reddit
Input Graph G=(V,E)	G=(2708,10556)	G=(3327,9104)	G=(19717,88648)	G=(65755,266144)	G=(232965,114615892)
Feature Length	1,433-16-7	3,703-16-6	500-16-3	61,278-64-186	602-64-41
Density of Adjacency Matrix (A)	0.18%	0.11%	0.028%	0.0073%	0.21%
Density of Feature Matrix (X)	L1:1.27%, L2:78%	L1:0.85%, L2:89.1%	L1:10.0%, L2:77.6%	L1:0.01%, L2:86.4%	L1:51.6%, L2:60.0%
Density of Weight Matrix (W)	L1:100%, L2:100%	L1:100%, L2:100%	L1:100%, L2:100%	L1:100%, L2:100%	L1:100%, L2:100%

as shown in Fig 8 (e). As explained in Sec. III-B1, the proposed design utilizes the outer product instead of the inner product when performing the loop unrolling from the algorithm level for matrix-matrix multiplications. Therefore, this strategy easily eliminates the need for designing additional hardware components to combine the final results of both vertex types (HD and LD).

IV. EXPERIMENTAL METHODOLOGY

A. Experimental Setup

We evaluate the performance (execution time) of the proposed chiplet-based GCN accelerator design through a cycle-accurate simulator using C++ to model the behavior of the hardware with the analysis mentioned in previous sections. The simulator counts the demand amount of on-chip and off-chip memory access, which is used to estimate the related energy consumption according to [49]. To measure the area and power consumption, we model all the required hardware inside the proposed design. We use the Synopsys Design Compiler with the TSMC 40*nm* library to synthesize and estimate the power using Synopsys PrimeTime PX. We set the clock frequency at 500 MHz. Furthermore, we use Cacit [50] and Dsent [51] to estimate the area, power, and access latency of the switch, on-chip FIFO buffers, MUX-DEMUXes, and control logic. We compare the proposed chiplet-based accelerator design (OPT-GCN) with three previous GCN accelerators: HyGCN [31], AWB-GCN [32], and GCNAX [25]. To simulate the power-law distribution for all GCN datasets simply, we assume that 20% of the total graph vertices are high-degree vertices and occupy around 80% of the entire input edges. Additionally, we conduct a hardware parameter-sensitive analysis, including the power-law distribution, number of PEs, and limited bandwidth, to show how each parameter impacts the overall performance.

The baseline accelerators are scaled to be equipped with the same number of computing units and memory bandwidth as the proposed design. Since most of the previous accelerators use single-precision floating point numbers (32-bit), on-chip links used for data communication are 32-bit in width for all accelerators. Since the HyGCN deploys a tandem-engine architecture with two computing engines for the Aggregation and Combination phases, MAC units are divided into two groups equally to simulate the two separate engines for required computations. For AWB-GCN and GCNAX, they both use one 1×16 MAC (PE) array as a unified computing engine for both aggregation and combination phases. For the proposed chiplet-based design, each chiplet applies the 1×16 MAC (PE) array as a unified computing engine to support 32-bit data computations. Furthermore, all baseline accelerators are equipped with the same-sized on-chip buffer storage. For a fair comparison, the DRAM bandwidth for all accelerators is scaled to 128GB/s.

B. Evaluation Datasets

In this paper, we leverage commonly used datasets from previous literature [17], [52]–[57] to conduct the further experimental evaluation. These datasets include Cora, Citeseer, Pubmed, Nell, and Reddit. Cora, Citeseer, and Pubmed are well-known datasets for paper citation networks, node classification, and text summarization [8], [54], [58], [59]. The Reddit dataset represents an undirected graph of social networks, comprising posts gathered from the Reddit discussion forum. The Nell dataset, on the other hand, is a knowledge graph obtained from the Never-Ending Language Learning project. Table I provides detailed information about each dataset used in this study, including its structure and data density. It is important to note that, except for the NELL dataset, the length of features in most datasets decreases with each GCN layer. The behavior of the NELL dataset differs in this aspect.

C. Architecture Modeling

In this subsection, we present a comprehensive analytical model of the performance (cycles of execution time) and data memory access required during our evaluation section. To capture the polyhedral nature of the design space, we demonstrate our analytical model using the design choices depicted in Fig. 6 as a prime example.

1) Performance Modeling: Because of the limited capacity of GLB and the on-chip buffer, all evaluation parameters are modeled based on pre-defined tile size. Given a specific tile size ($< T_{n0}, T_{k0}, T_{c0} >$) with sparsity ($S_A$) and a power-law distribution ($P\%$), the total number of execution cycles for one specific chiplet to perform the required computations of either the HD vertices or the LD vertices are shown as Eq. 9 and Eq. 10, respectively. Since the computation order impacts the execution cycles, we assume that the computation order for HD and LD vertices are (AX)W and A(XW) respectively. As mentioned previously, all chiplets work on different workloads concurrently and independently. Therefore, the total number of execution cycles to perform the required computations of the entire GCN model is determined by the chiplet that completes its workload last.

$$\begin{cases} N_{HD\_cycles} = N_{agg.} + N_{comb.} \\ N_{agg.} = S_A \times \lceil \frac{NP}{T_{n0}} \rceil \times \lceil \frac{NP}{T_{n0}} \rceil \times \lceil \frac{K}{T_{k0}} \rceil \times (\mathrm{T}_{n0} \times \mathrm{T}_{k0}) \\ N_{comb.} = \lceil \frac{NP}{T_{n0}} \rceil \times \lceil \frac{K}{T_{k0}} \rceil \times \lceil \frac{C}{T_{c0}} \rceil \times (\mathrm{T}_{n0} \times \mathrm{T}_{c0}) \end{cases} \tag{9}$$

$$\begin{cases} N_{LD\_cycles} = N_{agg.} + N_{comb.} \\ N_{agg.} = S_A \times \lceil \frac{N(1-P)}{T_{n0}} \rceil \times \lceil \frac{N(1-P)}{T_{n0}} \rceil \times \lceil \frac{C}{T_{c0}} \rceil \times (\mathrm{T}_{n0} \times \mathrm{T}_{c0}) \\ N_{comb.} = \lceil \frac{N(1-P)}{T_{n0}} \rceil \times \lceil \frac{K}{T_{k0}} \rceil \times \lceil \frac{C}{T_{c0}} \rceil \times (\mathrm{T}_{n0} \times \mathrm{T}_{c0}) \end{cases} \tag{10}$$

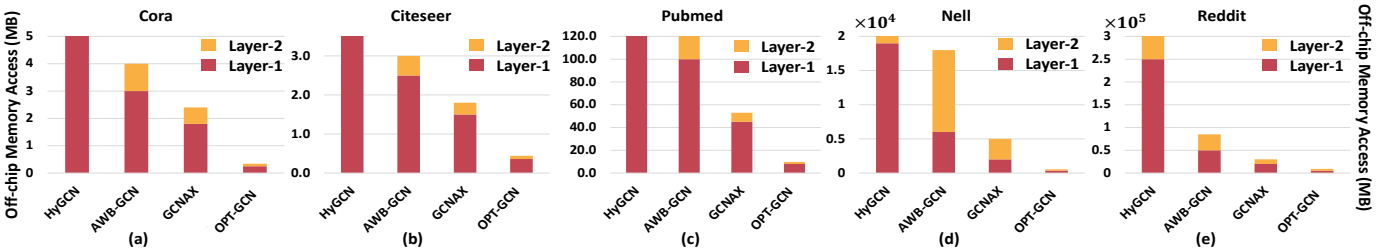


Fig. 10. Total off-chip memory access (MB) of the chiplet-based design (OPT-GCN) compared to prior accelerators on real-world graph datasets. (a) Cora, (b) Citesser, (c) Pubmed, (d) Nell, and (e) Reddit (lower is better).

2) Data Access Modeling: Since the main memory data accesses typically dominate the energy consumption of accelerators [25], [29], [31], reducing the number of data accesses can significantly improve the overall energy efficiency. As the design space is polyhedral, we will use an example to illustrate how the analytical models for the main memory data accesses are built. Assume the predefined tile size for each chiplet is $(< T_{n0}, T_{k0}, T_{c0} >)$ with **an average sparsity** (S_A) and loop fusion is not enabled during the two GCN phases. Given N Chiplets, there are (N$\times T_{n0}$) vertices loaded from the main memory to GLB during each interaction. Same with Sec. IV-C1, we assume that the aggregation phase is performed first by the HD vertices, while the combination phase is performed first by the LD vertices. Therefore, the total number of the main memory accesses is shown in Eq. 11. Here $\alpha_A, \alpha_X, \alpha_B, \alpha_O$ and M_A, M_X, M_B, M_W, M_O denote the trip counts and buffer sizes of memory accesses to matrix A/X/B/W/O respectively.

$$\begin{cases} N_{Total} = N_{HD\_Access} + N_{LD\_Access} \\ N = \alpha_A M_A + \alpha_X M_X + \alpha_{I1} M_{I1} \\ +\alpha_{I2} M_{I2} + \alpha_W M_W + \alpha_O M_O \end{cases} \tag{11}$$

Take the **HD** vertices with (AX)W computation order as an example:

$$\begin{cases} M_A = S_A \cdot T_{n0} \cdot T_{n0} \\ M_X = T_{n0} \cdot T_{k0} \\ M_{I1} = T_{n0} \cdot T_{k0} \\ M_{I2} = T_{n1} \cdot T_{k1} \\ M_W = T_{k1} \cdot T_{c1} \\ M_O = T_{n1} \cdot T_{c1} \end{cases} \tag{12}$$

$$\begin{cases} \alpha_A = \alpha_X = \frac{NP}{T_{n0}} \cdot \frac{N}{T_{n0}} \cdot \frac{K}{T_{k0}} \\ \alpha_{I1} = \frac{N}{t_{n0}} \cdot \frac{K}{T_{k0}} \\ \alpha_{I2} = \alpha_W = \frac{N}{T_{n0}} \cdot \frac{K}{T_{k0}} \cdot \frac{C}{T_{c0}} \\ \alpha_O = \frac{N}{T_{n0}} \cdot \frac{C}{T_{c0}} \end{cases} \tag{13}$$

Note that $\alpha_{I1}, \alpha_{I2}, M_{I1}, M_{I2}$ are used to differentiate the accesses in two different GCN phases respectively. In this model, we assume that the zeros in matrix A are evenly distributed so we can use the overall density (S_A) of A to represent the density of each loaded chunk when estimating the memory access of A. Although it does not reflect the actual distribution, we make this assumption for simplicity since considering the sparsity distribution would lead to a significant increase in model complexity. Additionally, we have observed that the estimated values generated by the model show minimal deviation when compared to the actual values obtained from a cycle-accurate simulation.

V. Evaluation and Analysis

A. Data Memory Access

Fig. 10 shows the normalized data memory access of the proposed design compared to previous works. Firstly, OPT-GCN partitions the input matrix into sub-matrices based on the power-law distribution for further parallel processing. Then, OPT-GCN implements a computation order decision algorithm (WSA) to determine the optimal computation orders for both high-degree and low-degree vertices, which reduces the DRAM access. Compared to previous works, only PEs in chiplets that perform the combination phase share the weight matrix. Other chiplets will independently load the required data from GLB to perform the aggregation phase. Compared to the scale-up HyGCN, the new chiplet design proposed in this paper uses a unified computing engine to support both phases efficiently instead of two separate engines. Compared to both scale-up AWB-GCN and GCNAX, the proposed design utilizes the power-law distribution to reduce off-chip memory access. Although the proposed design utilizes a predefined tile size for all GCN datasets inducing extra DRAM access, it is acceptable compared to hardware complexity when designing an architecture to support adaptive tile sizes for variable datasets. Compared to HyGCN, AWB-GCN, and GCNAX, the proposed design reduces the DRAM access by a factor of 11.3×, 3.4×, and 1.4×, respectively.

B. Performance

Fig. 11 shows the performance of the proposed design (OPT-GCN) compared to the other scale-up baselines, measured by the total number of execution cycles. The execution time includes the time of configuration calculation, sending the control signals, and setting up. The proposed design achieves a reduced execution time by 16.0×, 2.9×, 1.8× on average compared to prior works. This benefit arises from the unified computing engine implemented in each chiplet, which avoids workload imbalance induced by tandem engines when processing variable datasets. Moreover, OPT-GCN meticulously designs dataflows among chiplets and within chiplets, leading to additional reductions in data memory access. Additionally, OPT-GCN introduces and employs two algorithms to dynamically configure the architecture, optimizing computation patterns for the input dataset and the applied GCN model, and efficiently utilizing hardware resources. This paper also includes a performance breakdown analysis, aiming to elucidate the effectiveness of various methodologies proposed in the design for enhanced understanding, as depicted in Fig. 13. OPT-GCN-1 and OPT-GCN-2 denote the proposed architecture without and with the degree-aware scheduling algorithm, respectively. The performance breakdown analysis utilizes the NELL dataset, selected for its significant variation in feature length during each GCN layer.

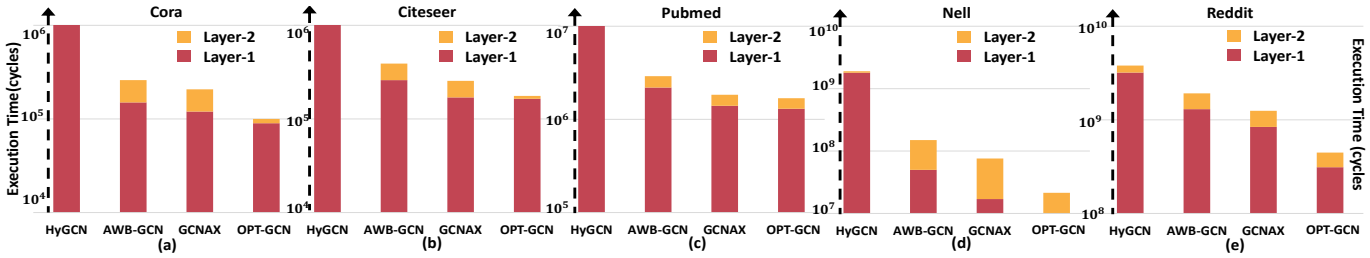


Fig. 11. Total execution time (cycles) of the chiplet-based design (OPT-GCN) compared to prior accelerators on real-world graph datasets. (a) Cora, (b) Citesser, (c) Pubmed, (d) Nell, and (e) Reddit (lower is better).

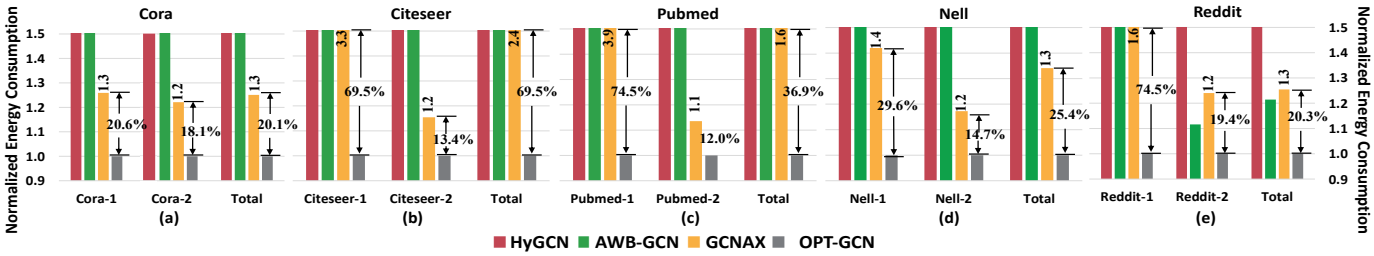


Fig. 12. Normalized energy consumption of the chiplet-based design (OPT-GCN) compared to prior accelerators on real-world graph datasets. (a) Cora, (b) Citesser, (c) Pubmed, (d) Nell, and (e) Reddit. All values are normalized to the proposed design. Some values of HyGCN and AWB-GCN are beyond the y-axis range for all datasets, which are not shown in the figure (lower is better).

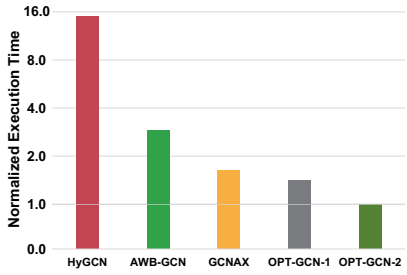


Fig. 13. Normalized execution time of the proposed design compared to prior accelerators with NELL dataset (lower is better).

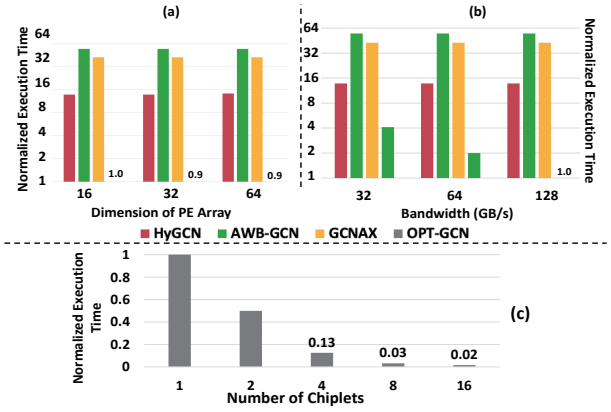


Fig. 14. Scalability analysis: (a) Scaling the number of Processing Elements (PEs). (b) Scaling the bandwidth. (c) Scaling the number of chiplets.

C. Energy Consumption

All accelerators estimate the related energy consumption according to [50], and are normalized based on the proposed chiplet-based design. Compared to HyGCN, the proposed design utilizes the compressed A matrix in CRC format to directly perform computations of the aggregation phase, which eliminates redundant GCN computations during the aggregation phase, which are induced by "0s" in the uncompressed A matrix. Additionally, the proposed design implements optimization from the algorithm level for reduced energy consumption. Specifically, the computation order decision algorithm (WSA) is used to reduce the number of main memory accesses and the number of computations. The proposed mapping algorithm is used to improve the efficiency of the hardware platform with a hybrid workload pattern, which decreases the total execution time. As shown in Fig. 12, the proposed design achieves 15.2×, 3.7× 1.6× energy savings on average compared to the scale-up HyGCN, AWB-GCN, and GCNAX, respectively.

D. Scalability Analysis

As stated in Sec. V, the baseline accelerators are scaled to match the hardware resources of the proposed design and a predefined threshold of the power-law distribution for evaluation, which could potentially compromise their efficiency. Therefore, we conducted a scalability analysis in this section to evaluate how variations in parameters affect the performance of the system. All the analysis uses NELL [8] as the input graph dataset. This is because the NELL dataset exhibits a distinct characteristic in that the length of its features is augmented in each GCN layer, setting it apart from the other datasets used in this paper. To conduct a thorough scalability analysis, we model the execution time of our proposed design and compare it to prior works as the bandwidth, and the number of PEs is scaled up. All values have been normalized to the proposed design that features 16 PEs and a bandwidth of 128 GB/s. As shown in Fig. 14 (a), we scale the number of PEs per chiplet from 16 PEs to 64 PEs while maintaining a predefined DRAM bandwidth of 128 GB/s. The proposed accelerator demonstrates superior performance in comparison to the prior accelerators. To explore the scalability of bandwidth, we sweep

the DRAM bandwidth provisioning varied from 32 GB/s up to 128 GB/s while keeping the other hardware parameters constant (each baseline configuration comprises 16 PEs in their computing engine), as shown in Fig. 14 (b). Additionally, in Fig. 14 (c), we present the performance scalability of running NELL across various numbers of chiplets, each consisting of 16 PEs. All execution times are normalized to a single chiplet performing GCN inference with the given input graph.

E. Sensitivity Analysis

In order to investigate the impact of the threshold sensitivity in power-law distribution, as shown in Fig. 15 (a) and (b), we conducted a sweeping analysis by varying the percentage of the high-degree (HD) vertices from 10% to 30% while occupying a constant number (80%) of the total edges. Subsequently, the percentage of edges occupied by HD vertices (20%) varied from 90% to 70%. It is evident that a graph with a high number of HD vertices and edges exhibits superior performance in relation to DRAM access. It should be noted that the data presented in this context has been normalized to the case that 20% HD vertices occupy 80% of total edges. To provide a clear representation of the trends in performance, energy consumption, and area cost as the dimension of the PE array scales up in each chiplet, we have prioritized each parameter. The final cost of each PE array is determined by summing up the values of each evaluation parameter and its corresponding priority. As a result, the cost value accurately reflects the relationship between the evaluation parameters and the dimension of the PE array, as depicted in Fig. 15 (c). The X-axis (N) denotes the dimension of the N×1 PE array within the unified computing engine. The experimental results suggest that employing 16 PEs in the proposed design achieves an optimal equilibrium among performance, energy consumption, and area consumption.

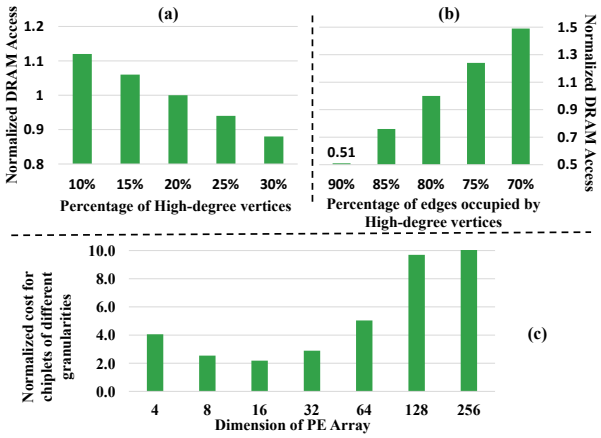


Fig. 15. Parameter sensitive analysis: (a) A fixed number of edges are occupied by different percentages of high-degree (HD) vertices. (b) A fixed number of HD vertices are occupied by different percentages of edges. (c) Chiplet Granularity: The X-axis (N) represents the dimension of the N×1 PE array within the unified computing engine, ranging from 4 PEs to 128 PEs.

F. Area and Power Consumption

Table II summarizes details of the area and power consumption of the proposed design, which includes a 2×2 chiplet GCN accelerator array in total. Extra Hardware includes the additional links used to send control signals for configuration. Because the proposed design is a chiplet-based design with three layers, the largest of the three layers, which is the chiplet array, determines the overall area consumption. Power consumption can be divided into two types: static power consumption and dynamic power consumption. Table II only lists the static power consumption of each hardware component in the proposed design. Obviously, the GLB is the most power-consumed hardware component, which is around 50% of the entire design. For the dynamic power consumption, the computing engine of each chiplet processes different GCN phases with a variable number of vertices, which impacts the specific value of dynamic power consumption. Additionally, the dynamic workload pattern also determines the memory access that each chiplet has. As a result, the dynamic power consumption of the overall architecture design is determined by the sparsity, the size of the input graph, and the applied GCN model.

TABLE II
AREA AND POWER CONSUMPTION

Hardware parameters	**Area** (mm^2)	**Power (mW)**
The unified controller	0.3	15.5
Global Buffer	10.8	297.2
A Specific Chiplet design	3.7	92.15
Extra Hardware	0.4	20.8
Total	14.8	702.1

VI. CONCLUSION

In this paper, we present a chiplet-based GCN accelerator design that offers high performance and energy efficiency for graph-related applications through an architecture-algorithm co-design. The proposed accelerator can be configured at the architecture level to provide GCN inference with flexibility and scalability. Additionally, from the algorithm side, we propose workload scheduling and mapping algorithms that leverage the input graph dataset, the applied GCN model, and the hardware configuration to achieve maximum efficiency. Experimental results with real-world graphs show that the proposed design outperforms prior works in terms of performance and energy efficiency.

REFERENCES

[1] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the 29th IEEE conference on computer vision and pattern recognition (CVPR)*, pages 779–788. Las Vegas, NY, USA, June. 26 - July. 1, 2016.

[2] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proceedings of the 29th advances in neural information processing systems (NIPS)*, pages 1–9. Montreal, Quebec, Canda, December 7-12, 2015.

[3] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[4] Weijing Shi and Raj Rajkumar. Point-gnn: Graph neural network for 3d object detection in a point cloud. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition (CVPR)*, pages 1711–1719, June 14-19, 2020.

[5] Xiaoyang Wang, Yao Ma, Yiqi Wang, Wei Jin, Xin Wang, Jiliang Tang, Caiyan Jia, and Jian Yu. Traffic flow prediction via spatial temporal graph neural network. In *Proceedings of the web conference 2020*, pages 1082–1092. Taipei, China, April 20-24, 2020.

[6] Shutao Li, Weiwei Song, Leyuan Fang, Yushi Chen, Pedram Ghamisi, and Jon Atli Benediktsson. Deep learning for hyperspectral image classification: An overview. *in IEEE Transactions on Geoscience and Remote Sensing*, vol. 57, no. 9, pp. 6690–6709, April, 2019.

[7] Tsung-Han Chan, Kui Jia, Shenghua Gao, Jiwen Lu, Zinan Zeng, and Yi Ma. Pcanet: A simple deep learning baseline for image classification? *in IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 5017–5032, September, 2015.

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

[9] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

[10] Ruoyu Li, Sheng Wang, Feiyun Zhu, and Junzhou Huang. Adaptive graph convolutional neural networks. In *Proceedings of the 32nd Association for the Advancement of Artificial Intelligence (AAAI)*. New Orleans, Louisiana, USA, February 2-7, 2018.

[11] Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *Proceedings of the 36th International Conference on Machine Learning (ICML)*, pages 6861–6871. Long Beach, California, USA, June 9-15, 2019.

[12] 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 the 54th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1051–1063. Athens, Greece, October 18-22, 2021.

[13] Bingyi Zhang, Rajgopal Kannan, and Viktor Prasanna. Boostgcn: A framework for optimizing gcn inference on fpga. In *Proceedings of the 29th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 29–39, May 9-12, 2021.

[14] Tao Yang, Dongyue Li, Fei Ma, Zhuoran Song, Yilong Zhao, Jiaxi Zhang, Fangxin Liu, and Li Jiang. Pasgcn: An reram-based pim design for gcn with adaptively sparsified graphs. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 42, no. 1, pp. 150–163, May, 2022.

[15] Sudipta Mondal, Susmita Dey Manasi, Kishor Kunal, S Ramprasath, Ziqing Zeng, and Sachin S Sapatnekar. A unified engine for accelerating gnn weighting/aggregation operations, with efficient load balancing and graph-specific caching. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, December, 2022.

[16] Kishor Kunal, Tonmoy Dhar, Meghna Madhusudan, Jitesh Poojary, Arvind K Sharma, Wenbin Xu, Steven M Burns, Jiang Hu, Ramesh Harjani, and Sachin S Sapatnekar. Gnn-based hierarchical annotation for analog circuits. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, December, 2022.

[17] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *in IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, March, 2020.

[18] Ziwei Zhang, Peng Cui, and Wenwu Zhu. Deep learning on graphs: A survey. *in IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 1, pp. 249–270, March, 2020.

[19] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *arXiv preprint arXiv:1902.08730*, 2019.

[20] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *in AI Open*, vol. 1, pp. 57–81, January, 2020.

[21] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *in IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, December, 2008.

[22] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pages 257–266. Anchorage, Alaska, USA, August 4-8, 2019.

[23] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *in Computational Social Networks*, vol. 6, no. 1, pp. 1–23, March, 2019.

[24] Xuefeng Xian, Ligang Fang, and Shiming Sun. Regnn: A repeat aware graph neural network for session-based recommendations. *in IEEE Access*, vol. 8, no. 1, pp. 98518–98525, May, 2020.

[25] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *Proceedings of the 27th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788. Seoul, South Korea, February 27 - March 3, 2021.

[26] 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 the 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474. Seoul, South Korea, April 2-6, 2022.

[27] Jiajun Li, Hao Zheng, Ke Wang, and Ahmed Louri. Sgcnax: A scalable graph convolutional neural network accelerator with workload balancing. *in IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 33, no. 11, pp. 2834–2845, December, 2021.

[28] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W Keckler, and William J Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. Toronto, ON, Canada, Jun 24-28, 2017.

[29] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *in ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, February, 2014.

[30] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *in IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138(1):127–138, November, 2016.

[31] 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 the 26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. San Diego, CA, USA, February 22-26, 2020.

[32] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. Virtual, October 17-21, 2020.

[33] Shun Li, Yuxuan Tao, Enhao Tang, Ting Xie, and Ruiqi Chen. A survey of field programmable gate array (fpga)-based graph convolutional neural network accelerators: challenges and opportunities. *in PeerJ Computer Science*, vol. 8, pp. e1116, November, 2022.

[34] Chukwufumnanya Ogbogu, Aqeeb Iqbal Arka, Lukas Pfromm, Biresh Kumar Joardar, Janardhan Rao Doppa, Krishnendu Chakrabarty, and Partha Pratim Pande. Accelerating graph neural network training on reram-based pim architectures via graph and model pruning. *in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, December, 2022.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *in Communications of the ACM*, vol. 60, no. 6, pp. 84–90, May, 2017.

[36] Jilan Lin, Shuangchen Li, Yufei Ding, and Yuan Xie. Overcoming the memory hierarchy inefficiencies in graph processing applications. In *Proceedings of the 40th IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9. Munich, Germany, November 1-6, 2021.

[37] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30. Hollywood, CA, USA, October 8-10, 2012.

[38] Lada A Adamic and Bernardo A Huberman. Power-law distribution of the world wide web. *in American Association for the Advancement of Science*, vol. 287, no. 5461, pp. 2115–2115, March, 2000.

[39] Jason Mohoney, Roger Waleffe, Henry Xu, Theodoros Rekatsinas, and Shivaram Venkataraman. Marius: Learning massive graph embeddings on a single machine. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 533–549. virtual, July 14-16, 2021.

[40] Muhammad Adnan, Yassaman Ebrahimzadeh Maboud, Divya Mahajan, and Prashant J Nair. Accelerating recommendation system training by leveraging popular choices. *arXiv preprint arXiv:2103.00686*, 2021.

[41] Stefano Boccaletti, Vito Latora, Yamir Moreno, Martin Chavez, and D-U Hwang. Complex networks: Structure and dynamics. *in Physics reports*, vol. 424, no. 4-5, pp. 175–308, February, 2006.

[42] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. Power-law distributions in empirical data. *in SIAM review*, vol. 51, no. 4, pp. 661–703, 2009.

[43] Mohammad Rasool Izadi, Yihao Fang, Robert Stevenson, and Lizhen Lin. Optimization of graph neural networks with natural gradient descent. In *Proceedings of the 2020 IEEE international conference on big data (IEEE BigData 2020)*, pages 171–179. Virtual, December 10-13, 2020.

[44] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Heinrich Müller. Splinecnn: Fast geometric deep learning with continuous b-spline kernels. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 869–877. Salt Lake City, UT, USA, June 18-22, 2018.

[45] Sitao Luan, Chenqing Hua, Qincheng Lu, Jiaqi Zhu, Mingde Zhao, Shuyuan Zhang, Xiao-Wen Chang, and Doina Precup. Is heterophily a real nightmare for graph neural networks to do node classification? *arXiv preprint arXiv:2109.05641*, 2021.

[46] 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*, vol. 70, no. 9, pp. 1511–1525, August, 2020.

[47] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 233–244. Calgary, AB, Canada, August 11-13, 2009.

[48] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. Virtual, October 17-21, 2020.

[49] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pages 243–254. Seoul, South Korea, June 18-22, 2016.

[50] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *in HP laboratories*, vol. 27, pp. 28, April, 2009.

[51] Chen Sun, Chia-Hsin Owen Chen, George Kurian, Lan Wei, Jason Miller, Anant Agarwal, Li-Shiuan Peh, and Vladimir Stojanovic. Dsent-a tool connecting emerging photonics with electronics for opto-electronic networks-on-chip modeling. In *Proceedings of the 6th IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 201–210. Copenhagen, UT, USA, May 9-11, 2012.

[52] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and S Yu Philip. A survey on knowledge graphs: Representation, acquisition, and applications. *in IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 2, pp. 494–514, April, 2021.

[53] Andrew Kachites McCallum, Kamal Nigam, Jason Rennie, and Kristie Seymore. Automating the construction of internet portals with machine learning. *in Information Retrieval*, vol. 3, pp. 127–163, July, 2000.

[54] C Lee Giles, Kurt D Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the 3rd ACM conference on Digital libraries*, pages 89–98. Pittsburgh, PA, USA, June 23-26, 1998.

[55] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *in AI magazine*, vol. 29, no. 3, pp. 93–93, September, 2008.

[56] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st Annual Conference in Neural Information Processing Systems (NeurIPS)*. Long Beach, CA, USA, December 5-6, 2017.

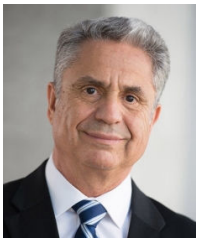
[57] Kun Zhan and Chaoxi Niu. Mutual teaching for graph convolutional networks. *in Future Generation Computer Systems*, vol. 115, pp. 837–843, October, 2020.

[58] Lu Yu, Shichao Pei, Lizhong Ding, Jun Zhou, Longfei Li, Chuxu Zhang, and Xiangliang Zhang. Sail: Self-augmented graph contrastive learning. In *Proceedings of the 36th AAAI Conference on Artificial Intelligence*, pages 8927–8935. Virtual, February 22 - March 1, 2022.

[59] Muhammet Balcilar, Guillaume Renton, Pierre Héroux, Benoit Gauzere, Sebastien Adam, and Paul Honeine. Bridging the gap between spectral and spatial domains in graph neural networks. *arXiv preprint arXiv:2003.11702*, 2020.

Yingnan Zhao received the BS degree in computer science from the Zhejiang University of Technology, China, in 2018, and the MS degree in electrical engineering from the George Washington University, US, in 2020. He is currently working toward a Ph.D. degree in computer engineering at George Washington University, Washington, DC. His research interests include graph neural networks, AI accelerator design, and interconnection networks.

Ke Wang received the Ph.D. degree in Computer Engineering from the George Washington University in 2022. He received the M.S. degree in Electrical Engineering from Worcester Polytechnic Institute in 2015, and the B.S. degree in Electrical Engineering from Peking University in 2013. He is currently an Assistant Professor of Electrical and Computer Engineering at the University of North Carolina at Charlotte. His research work focuses on parallel computing, computer architecture, interconnection networks, and machine learning.



Ahmed Louri (Fellow, IEEE) received the PhD degree in computer engineering from the University of Southern California, Los Angeles, California, in 1988. He is the David and Marilyn Karlgaard Endowed chair professor of electrical and computer engineering at the George Washington University, Washington, DC., which he joined in August 2015. He is also the director of High Performance Computing Architectures and Technologies Laboratory. From 1988 to 2015, he was a professor of electrical and computer engineering at the University of Arizona, Tucson, Arizona, and during that time, he served six years (2000 to 2006) as the chair of the Computer Engineering Program. From 2010 to 2013, he served as a program director in the National Science Foundation's (NSF) Directorate for Computer and Information Science and Engineering. He directed the core computer architecture program and was on the management team of several cross-cutting programs. He conducts research in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks, optical interconnects for parallel computing systems, reconfigurable computing systems, and power-efficient and reliable Network-on-Chips (NoCs) for multicore architectures. Recently he has been concentrating on energy-efficient, reliable, and high-performance many-core architectures, accelerator-rich reconfigurable heterogeneous architectures, machine learning techniques for efficient computing, memory, and interconnect systems, emerging interconnect technologies (photonic, wireless, RF, hybrid) for NoCs, future parallel computing models and architectures (including convolutional neural networks, deep neural networks, and approximate computing), and cloud-computing and data centers. He is the recipient of 2020 IEEE Computer Society Edward J. McCluskey Technical Achievement Award for pioneering contributions to the solution of on-chip and off-chip communication problems for parallel computing and many-core architectures. For more information, please visit https://hpcat.seas. gwu.edu/Director.html.