Morph-GCNX: A Universal Architecture for High-performance and Energy-efficient Graph Convolutional Network Acceleration

Ke Wang, *Member, IEEE,* Hao Zheng, *Member, IEEE,* Jiajun Li, *Member, IEEE,*
Ahmed Louri, *Fellow, IEEE*

Abstract—While current Graph Convolutional Networks (GCNs) accelerators have achieved notable success in a wide range of application domains, these GCN accelerators can not support various intra- and inter- GCN dataflows or adapt to diverse GCN applications. In this paper, we propose Morph-GCNX, a flexible GCN accelerator architecture for high-performance and energy-efficient GCN execution. The proposed design consists of a flexible Processing Element (PE) array that can be partitioned at runtime and adapt to the computational needs of different layers within a GCN or multiple concurrent GCNs. The proposed Morph-GCNX also consists of a morphable interconnection design to support a wide range of GCN dataflows with various parallelization and data reuse strategies for GCN execution. We also propose a hardware-application co-exploration technique that explores the GCN and hardware design spaces to identify the best PE partition, workload allocation, dataflow, and interconnection configurations, with the goal of improving overall performance and energy. Simulation results show that the proposed Morph-GCNX architecture achieves 18.8×, 2.9×, 1.9×, 1.8×, and 2.5× better performance, reduces DRAM accesses by a factor of 10.8×, 3.7×, 2.2×, 2.5×, and 1.3×, and improves energy consumption by 13.2×, 5.6×, 2.1×, 2.5×, and 1.3×, as compared to prior designs including HyGCN, AWB-GCN, LW-GCN, GCoD, and GCNAX, respectively.

Index Terms—Computer Architecture, Graph Convolutional Network (GCN), GCN Accelerator Design, Interconnection Network.

1 Introduction

Graph Neural Networks (GCNs) have achieved unparalleled success in a wide range of application domains, such as classification [1], [2], [3], prediction [4], [5], detection [6], [7], [8], and many others. Due to the data explosion in graph processing tasks, training and inference of GCNs involve substantial compute-and memory-intensive operations. As a result, domain-specific GCN accelerators [9], [10], [11], [12], [13], [14], [15] are developed to achieve significant performance and power improvements.

Existing GCN accelerators are optimized to accelerate two performance-dominating phases, namely aggregation and combination, by deploying uniquely designed computing units [9] or specifically tailored dataflows [10]. However, these rigid solutions neglect the fact that the computational characteristics of GCNs can vary significantly, where graph size and structure are different. Depending on the unique computational characteristics of GCNs, different GCN tasks (e.g., layers within a GCN, subgraphs of a distributed GCN application, or the size of GCN inputs) will benefit from diverse selections of dataflows (e.g., parallelism and data reuse strategies), which are not supported by current designs. Specifically, these dataflows require the communication fabrics capable of handling various data movement patterns (e.g. unicast and broadcast). The problem is further compounded by the concurrent execution of multiple GCN applications, all of which share the same computing units, on-chip buffers, and communication fabrics. This requires the accelerator to support dynamic partitioning to mitigate the interactions of multiple concurrent GCNs.In addition, an appropriate resource management policy is needed to allocate the appropriate amount of shared hardware components to each GCN application without inducing performance degradation.

In this paper, we explore the design of efficient accelerator architectures specifically tailored for GCNs. We propose Morph-GCNX, a universal architecture that can support the diverse computation and communication demands required by concurrently executing diverse GCN tasks, including GCN layers, subgraphs, inputs, and sparsity. In addition, we propose a hardware-application co-exploration algorithm to dynamically partition the morphable architectures to multiple sub-accelerators for multiple GCN task execution with the aim of finding the best dataflows and suitable sub-accelerator architecture for each GCN task. The major contributions of this paper are as follows:

- **Morphable Accelerator Design:** We propose a universal accelerator architecture to support high-performance and energy-efficient GCN execution. The proposed accelerator can be partitioned into multiple sub-accelerators with different sized PE-array,

- *Ke Wang is with the Department of Electrical and Computer Engineering, University of North Carolina at Charlotte, Charlotte, NC 28223. E-mail: ke.wang@uncc.edu.*
- *Hao Zheng is with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816. E-mail: Hao.Zheng@ucf.edu.*
- *Jiajun Li is with the School of Astronautics, Beihang University, Beijing, China. E-mail: jiajunli@buaa.edu.cn.*
- *Ahmed Louri is with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052. E-mail: louri@gwu.edu*

Manuscript received XX XX, 2022, revised XX XX, 2023.

global buffers, and interconnects. These components can be configured to support any desired inter-/intra-dataflows needed by the GCN tasks.

- **Dynamic Optimization Algorithm Design:** We propose a hardware-application co-exploration algorithm for optimized GCN execution. The proposed algorithm models the performance and energy metrics for concurrent GCN tasks. According to the modeled metrics, the proposed algorithm searches the design space and dynamically selects the most suitable partitioning strategy, dataflow parameters (e.g., tile sizes, inter-phase dataflows, and intra-phase dataflows), and interconnection configurations.

We implement the Morph-GCNX microarchitecture in RTL. We also evaluate the proposed Morph-GCNX with a cycle-accurate simulator that can accurately capture the behavior of each hardware component of the GCN accelerator. Simulation shows that the proposed Morph-GCNX architecture achieves 18.8×, 2.9×, 1.9×, 1.8×, and 2.5× better performance, reduces DRAM accesses by a factor of 10.8×, 3.7×, 2.2×, 2.5×, and 1.3×, and improves energy consumption by 13.2×, 5.6×, 2.1×, 2.5×, and 1.3×, as compared to prior designs including HyGCN, AWB-GCN, LW-GCN, GCoD, and GCNAX, respectively.

2 Background and Motivation

2.1 GCN Basics

A Graph Convolutional Network (GCN) takes graph-structured data as input and learns a representation vector for each vertex in the graph. As shown in Fig 1, each GCN layer gathers the activations of the neighbor vertices from the previous GCN layer, and then updates the activation of the vertex, using convolution and matrix multiplication. The computation in a GCN layer can be formulated as:

$$a_v^{(k)} = Aggregate^{(k)}(h_u^{(k)} | u \in \mathcal{N}(v)),$$
$$h_v^{(k)} = Combine^{(k)}(a_v^{(k)}, h_v^{(k-1)}) \quad (1)$$

where $h_v^{(k)}$ is the representation feature vector of vertex v at the k-th layer. The *Aggregate* function aggregates multiple feature vectors from source neighbors to one single feature vector, and the *Combine* function transforms the feature vector of each vertex to another feature vector using a multi-layer perceptron neural network. For the entire GCN, the prevalent computation pattern is modeled as a chain matrix-multiplication (ChainMM) operation:

$$X^{(k+1)} = \sigma(\hat{A}X^{(k)}W^{(k)}) \quad (2)$$

Where $X^{(k)}$ and $W^{(k)}$ are the input feature and weight matrices of layer k. $\sigma(\cdot)$ is the non-linear activation function, and a typical activation function is ReLU. A is a normalized transformed matrix from the graph adjacency matrix.Table 1 summarizes all the notations to describe a GCN in this paper. The tile size ($T_{<Dimension>}$) is the number of elements that are mapped across PEs in parallel in a given dimension.

For GCN accelerators, the performance of GCN execution is correlated to the compute ordering of the ChainMM. For instance, the execution order of the ChainMM can be altered: $(A \times X) \times W$ or $A \times (X \times W)$. And these two

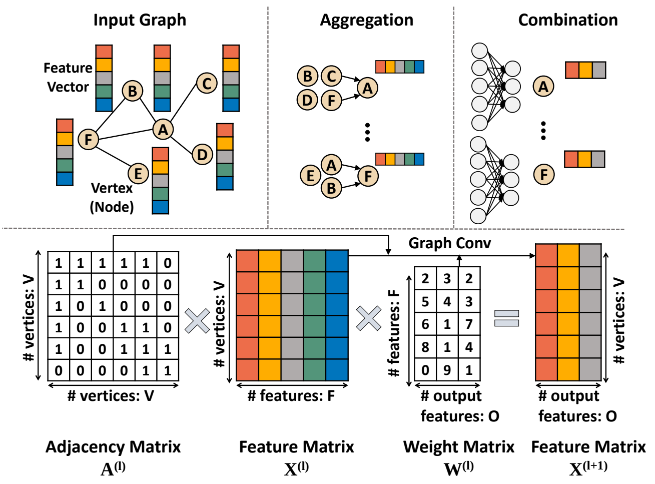


Fig. 1. GCN basics. The computation in a GCN layer consists of two phases, namely *Aggregation* and *Combination*. The figure shows an example graph with six nodes. The figure shows A: adjacency matrix,$X^{(l)}$: feature vectors of layer l, $W^{(l)}$: weight matrix of layer l, and $X^{(l+1)}$: feature vectors of layer $l+1$. Basic computation can be generalized as $X^{(l+1)} = A^{(l)}(X^{(l)}W^{(l)})$. AX is aggregation, XW is combination.

TABLE 1
GCN Notations.

Terminology	Description
G	graph $G = (V, E)$
V	vertices of G
E	edges of G
D_v	degree of vertex v
$e_{(i,j)}$	edge between vertex i and j
$A(A_{ij})$	adjacency matrix (elements)
a_v	aggregation feature vector
h_v	feature vector
X	feature matrix
F	number of input features
O	number of output features
$N_v(S_v)$	number of neighbors in the sampling set S_v
$T_{<Dimension>}$	tile size

execution orders are different in the following aspects. First, in GCN, the computation with A is closely related to the *Aggregation* phase, which can be generalized as sparse-sparse matrix multiplication (SpGEMM) or sparse-dense matrix multiplication (SpMM) operations, while computations with W are parts of the *Combination* phase, which are general matrix multiplication (GEMM) or dense matrix multiplication (DenseMM) operations. The difference in the sparsity of SpMM and GEMM operations can significantly affect the efficiency of the accelerator, in terms of memory accesses and PE utilization. Second, the order in which the product is parenthesized is closely correlated with the number of simple arithmetic operations needed to compute the final product, which leads to different computational latencies.

Additionally, the dataflows for *Aggregation* and *Combination* phases also impact the data reuse and the intensity of memory accesses, which can lead to diverse system-level performance and power metrics. For aggregation, the computation can be characterized using three layers of nested loops, namely the vertex dimension V, the input feature dimension F, and the neighbor dimension N. These three dimensions can be either in the inner loop or in

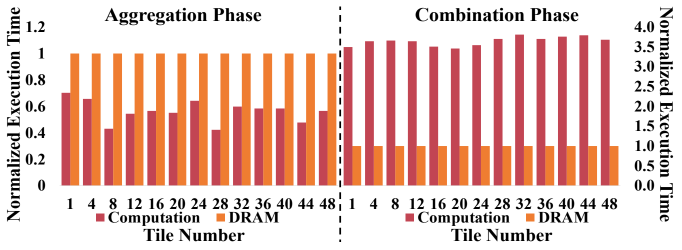


Fig. 2. Time distribution of each tile's memory access (DRAM) and on-chip computations for both the aggregation and combination phases. The time of computations is normalized to that of DRAM access.

the outer loop, and the vectors of each dimension can be either spatial or temporal. Different combinations of temporal and spatial loop orderings can play a critical role in aggregation performance. Moreover, the inter-phase execution order (aggregation to combination or combination to aggregation) also affects the data movement, data reuse, and buffer utilization of the accelerator, which are related to data transmission latency, DRAM accesses, and on-chip resource allocation efficiency. Therefore, as the selection of dataflows affects the performance of GCN inference, it is critical for an accelerator to support a wide range of dataflows and select the most suitable one for a given GCN task. Additionally, as different dataflows can lead to erratic workload mapping and diverse on-chip data transmission, it is also beneficial for an accelerator to be reconfigurable and adapt to the dynamically changing PE partitions, workload mapping, and PE communication patterns.

2.2 GCN Accelerators

In GCN, the aggregation phase is dependent 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 deep neural networks, 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. Existing GCN accelerators implement hardware designs to accelerate the GCN aggregation and combination phases and improve overall performance and energy efficiency. For instance, HyGCN [9] exploits two dedicated compute engines to respectively accelerate aggregation and combination phases. GCoD [14] integrates two engines for SpMM and GEMM. AWB-GCN [10] and LW-GCN [15] address the workload imbalance issue in SpMM kernels. GCNAX [16] proposes a new GCN architecture that supports flexible dataflow to improve resource utilization. All these designs share a typical architecture that is comprised of a GCN accelerator chip and off-chip memory. Specifically, the GCN accelerator chip consists of a Processing Engine to support compute parallelism in matrix multiplications, a global buffer (GLB) that is constructed with SRAM scratchpad memory for data reuse, and a controller to map the GCNs onto the PUs and GLB. As compared to the communication across the memory hierarchy is intense within the accelerator chip, the off-chip memory (DRAM) accesses consumes the majority of the overall power and time [17].

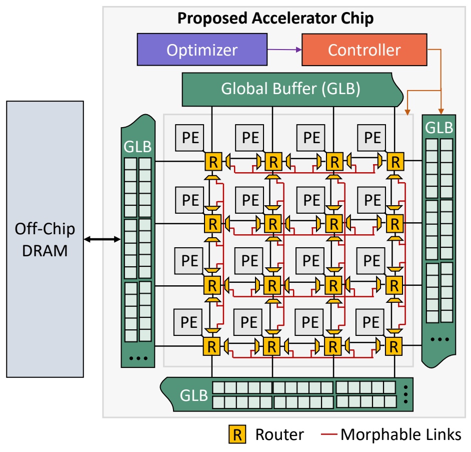
Although these GCN accelerators have delivered considerable performance improvements to some GCN applications, few of them have considered supporting a wide range of intra- and inter- GCN dataflows or adapting to diverse characteristics of GCN tasks. These designs with monolithic computation and communication patterns are limited in their flexibility to perform the two distinct GCN phases efficiently (due to restricted partitioning, fixed dataflow that limits data reuse, etc). Furthermore, as the GCN tasks have different data sizes and graph structures: some vertices display a larger number of neighbors than others. the vertices of the input graph can be divided into two categories: high-degree (HD) and low-degree (LD) vertices. 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. This can result in diverse global buffer requirements, memory access, and computation patterns, due to different intermediate result sizes. For instance, as shown in Fig. 2, with a number of data chunks (tile) in the Cora [18], [19], [20], [21] (a real-world graph dataset), either memory data access or on-chip computations dominate the performance during the aggregation and combination phases, respectively. As a result, a monolithic can be inefficient, and 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. Moreover, these existing GCN accelerators do not integrate an efficient strategy for executing multiple GCNs concurrently. To address these issues, we propose a morphable GCN accelerator design.

3 Morphable GCN Accelerator Design

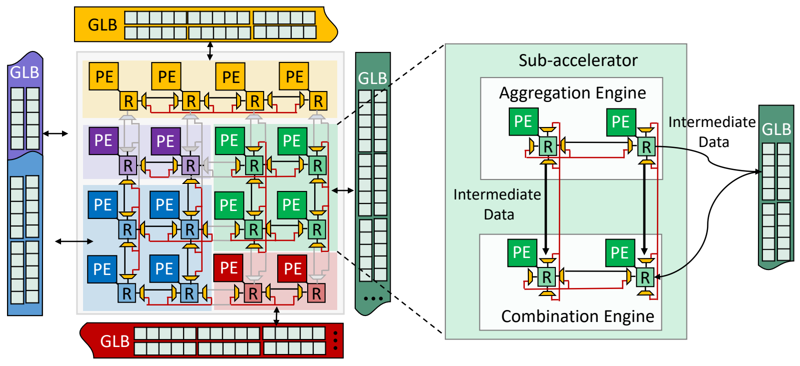
The proposed design has three major objectives: (1) to optimize the concurrent mapping of several GCN tasks (layers, sub-GCNs, or applications) at runtime with dynamic PE partitioning, (2) to select the most suitable intra- and inter-phase dataflows for each GCN task, and (3) to morph the interconnection of PEs adapting to the selected dataflow with the purpose of improving performance and energy efficiency. As such, each GCN task will be allocated with different numbers of PEs and buffers. Within each partitioned PE array, dataflow, together with the interconnect, would be optimized according to the needs of the running application. The details of the proposed design are described as follows. In this section, we present the flexible architecture design of the proposed accelerator. The algorithm used for selecting the accelerator's configuration parameters is described in Sec. 4.

3.1 Overview of the Proposed Morph-GCNX Accelerator

We propose an accelerator architecture for efficient GCN execution, which is depicted in Fig. 3 (a). The proposed accelerator consists of an accelerator chip and an off-chip DRAM. The accelerator chip is comprised of distributed global buffers (SRAMs), a processing element (PE) array, and a morphable interconnect design. The global buffers



(a)



(b)

Fig. 3. The proposed Morph-GCNX accelerator design. (a) The overall architecture of the proposed accelerator. (b) An example showing that the universal accelerator is partitioned into five disjoint sub-accelerators to execute multiple GCN tasks concurrently. Each sub-accelerator consists of a sub-PE-array, GLB banks, and a disjoint portion of morphable interconnects. The sub-accelerator is further partitioned into aggregation and combination engines.

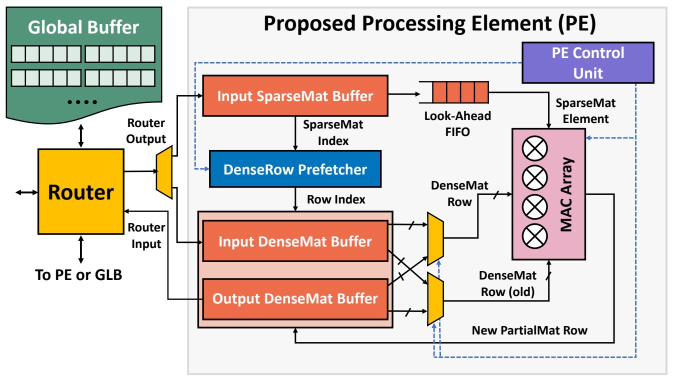


Fig. 4. The proposed processing element (PE) architecture.

(GLBs) are implemented as a multi-bank scratchpad memory, where each bank can be shared by different PEs. The proposed accelerator also integrates an optimizer and a controller. The optimizer models the computation and communication requirements of the GCN tasks and decides the optimized GCN configurations (e.g., tile size, loop order, etc.) for different GCN tasks. The optimizer generates corresponding commands to the Control Unit. Within the proposed accelerator, GLBs and PE array can be partitioned into multiple sub-accelerators as shown in Fig. 3 (b). For each partitioned sub-accelerator, the PEs can be further partitioned to aggregation and combination engines of any size, and the interconnect can also be morphed to support any connectivity required by the dataflows.

3.2 PE Architecture Design

Fig. 4 depicts the proposed PE architecture, which is a uniformed engine that can efficiently process sparse operations (SpMM and SpGEMM) and dense operations (DenseMM and GEMM). To support uniformed operations, the PE architecture is featured with a SparseMat Buffer (SMB), Input/Output DenseMat Buffers (IDMB/ODMB), a Look-Ahead FIFO, a DenseRow Prefetcher (DRP), a multiply-and-accumulate (MAC) Array, and a PE Control Unit.

Specifically, to process SpMM and SpGEMM, the tiled sparse matrix is delivered from GLB banks into SMB in CSC format via a morphable interconnect. Concurrently, part of the input/output matrix is fetched into IDMB and ODMB with a dense format. To perform SpMM, an element from SMB is pre-fetched to the FIFO, and its row index is sent to DRP. Afterward, the DRP fetches the corresponding input DenseMat row from IDMB using the coordinate information of the PartialMat row. Given the latency for matching the SparseMat element and the input DenseMat, a look-ahead FIFO is developed to hide this latency. The FIFO handles and forwards the SparseMat element to MAC Array. Upon receiving the operands, the MAC array will perform the outer product between the SparseMat and DenseMat elements, and its output will be accumulated. The computation results are stored in the output DenseMat Buffer and written back to the global buffer using the corresponding router. To process DenseMM and GEMM, the router directly loads data to the input DenseMat Buffer instead of using the input SparseMat Buffer.

3.3 Morphable Interconnection Network Design

In GCN, most of the existing dataflows rely on broadcast communication to pursue spatial data reuse or on unicast communication for partialMat accumulation. Existing architectures deploy rigid interconnect designs that only support either broadcast or unicast. Thus the performance is limited. This motivates us to design a morphable interconnect to support both broadcast and unicast functions at each row or column of the partitioned PE array. The morphable interconnect consists of two key designs, namely morphable router and morphable links. As shown in Fig. 5 (a), in a 4×4 morphable interconnect design, sixteen morphable routers are connected in a mesh-like topology. Bi-directional morphable links are connected to morphable routers at each row and column.

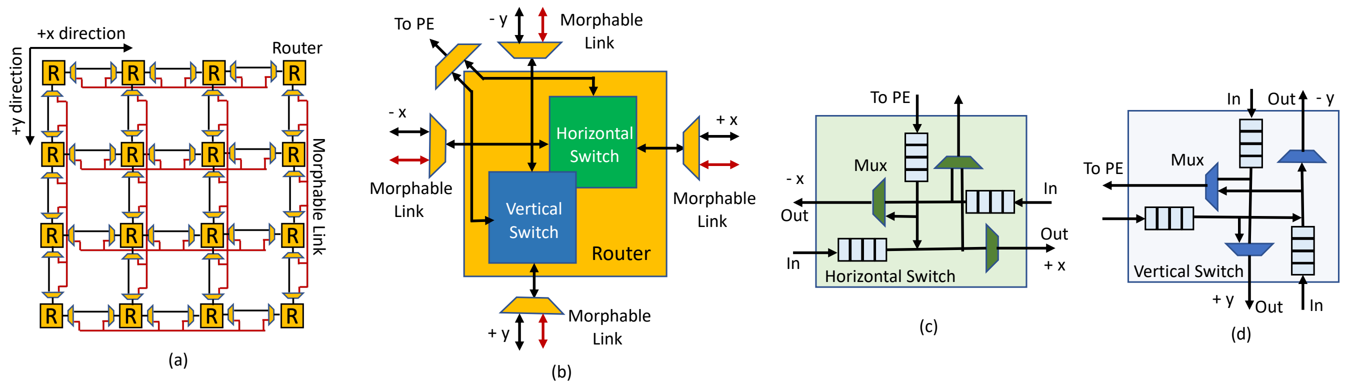


Fig. 5. The proposed morphable interconnect design: (a) An example of 4 × 4 morphable interconnect, (b) Morpahble router architecture consists of horizontal and vertical switches, (c) Horizontal switch architecture, and (d) Vertical switch architecture.

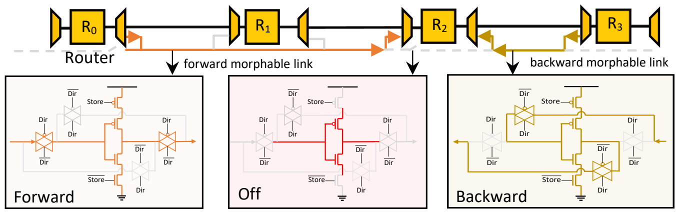


Fig. 6. The proposed morphable link design.

3.3.1 *Morphable Router*

While the PEs are connected in a mesh-like topology, the proposed morphable router can only support row/column-wise communications. This is because the latency overhead of conventional mesh routers would be prohibitive for GCN accelerators. Specifically, the proposed morphable architecture consists of a vertical switch and a horizontal switch as shown in Figure5 (c) and (d). The vertical switch processes column-wise communication, and the horizontal switch processes row-wise communication. Vertical and horizontal switches are further connected to the morphable links which will be discussed in the next section.

3.3.2 *Morphable Link*

The unicast and broadcast communications should be enabled for every row and column of the PE array, but neither bus nor mesh can provide both functions. To solve this issue, we propose a morphable link design to facilitate broadcast communication in the mesh-like topology. The morphable link is equipped with quad-state repeaters [22] as shown in Fig. 6. The quad-state repeaters can disable signal propagation (link segmentation) and switch the signal propagation directions (link reversal). For example, when the 'Dir' signal turns on, the link will be used for forward signal propagation, whereas the direction reverses when the 'Dir' turns off. When the 'store' is enabled, the link will be segmented into two parts. Therefore, each bi-directional link can be utilized as two uni-directional links with improved injection bandwidth, thereby supporting broadcast communication. The disabled quad-state repeaters can segment a morphable link into several short links, each supporting one partitioned sub-PE array.

4 Hardware-Application Co-exploration Algorithm for Multi-GCN Optimization

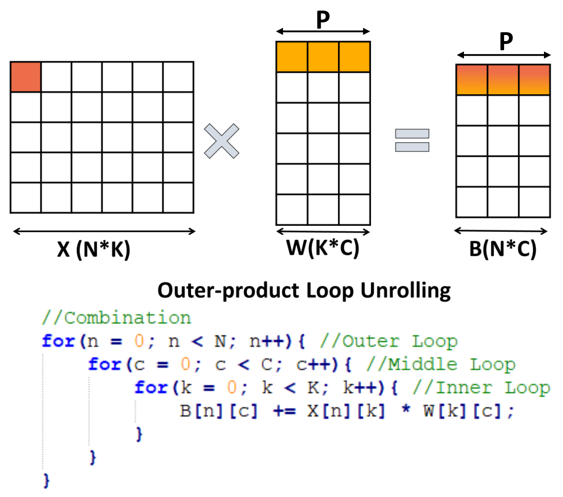
In this section, we present a hardware-application co-exploration algorithm for optimized GCN execution using the optimizer. Specifically, the GCN optimizer samples and models the performance and energy metrics for concurrent GCNs. The proposed GCN optimizer searches the design space and dynamically selects the most suitable PE partitions and dataflow parameters (e.g., tile sizes, inter-phase dataflows, and intra-phase dataflows). The algorithm follows three steps. First, the PE array is partitioned into disjoint sub-PE arrays, and each sub-PE array is responsible for executing one GCN task. Second, the sub-PE-arrays each are further partitioned into dynamically sized aggregation and combination engines. Third, the dataflow parameters for the aggregation and combination engines are decided. Details of each step are discussed as follows.

4.1 PE Array Partition for Concurrent GCNs

In the proposed accelerator design, we iteratively assign a number of PEs and a portion of GLBs to independent GCN tasks that are running concurrently. For concurrent GCNs, the optimizer uses the neighbor sampling technique [23] to sample the GCNs and calculates the number of vertices, edges, the sizes of matrices, and MAC operations for each GCN task. The hardware resource requirements are estimated using the sampling results, and the PE array, interconnects, and GLB banks are partitioned into sub-accelerators accordingly. At runtime, when a new GCN task joins, it will be first sampled by the optimizer, and the hardware resources are reallocated to all concurrent GCN tasks. Similarly, after the execution of any GCN task, the corresponding hardware resources are released to be allocated to other GCN tasks. Note that in this paper, we assume concurrent GCN tasks have the same priority, meaning that we treat each GCN task only relying on their sampled sizes. The PE partition is fair as long as the average waiting time for each GCN task is minimized.

4.2 Sub-accelerator Partition for Aggregation and Combination Engines

After the sub-accelerators are formed, the proposed optimizer decides the size of the aggregation and combination



```
//Combination
for(n = 0; n < N; n++){ //Outer Loop
    for(c = 0; c < C; c++){ //Middle Loop
        for(k = 0; k < K; k++){ //Inner Loop
            B[n][c] += X[n][k] * W[k][c];
        }
    }
}
```

Fig. 7. An example of outer-product loop unrolling.

engines, respectively. The major challenge faced by this phase is to allocate hardware resources without inducing workload imbalance. We propose a loop-unrolling-based technique to decide the partitions for aggregation and combination engines. We also propose a runtime group-and-shuffle mechanism to handle the issue of inter-PE workload imbalance.

4.2.1 *Sub-accelerator Partitioning using Loop-unrolling*

As the aggregation phase can be generalized as SpGEMM or SpMM operations while combination operations are DenseGEMM, it would result in imbalanced intra-PE workload since the MAC units are underutilized if they use zeros for the calculation. In this paper, we use outer-product loop unrolling to avoid intra-PE workload imbalance and determine the PE array organization and scale, loop parallelization strategy, and the required sizes of on-chip buffers. Fig. 7 shows an example of an outer-product loop unrolling for the combination phase. At each cycle, every row element of W is multiplied by an element of X. The accumulation of these elements will ultimately formulate matrix B. $(2\times P+1)$ SRAM reads and P SRAM writes will be required at each cycle, If data reuse is not enabled at local registers. The unrolling factor$P$ determines the total number of parallel MAC operations as well as the number of required multipliers. With the information of the number of MAC operations of aggregation and combination, the PEs within a subarray are proportionally allocated to the aggregation and combination engines, respectively. The selection of the unrolling parameter P is presented in Sec. 4.3.2.

4.2.2 *Inter-PE Workload Balancing*

The workload imbalance in GCN could be a major issue affecting overall system performance, due to the unbalanced distribution of the non-zeros in the adjacency and feature matrices. Therefore, we attempt to adopt a group-and-shuffle method to tackle such workload imbalance issues. Our approach groups the rows (or columns) in a sparse matrix with a similar density. As such, the grouped rows will be distributed to PEs with a close completion time.

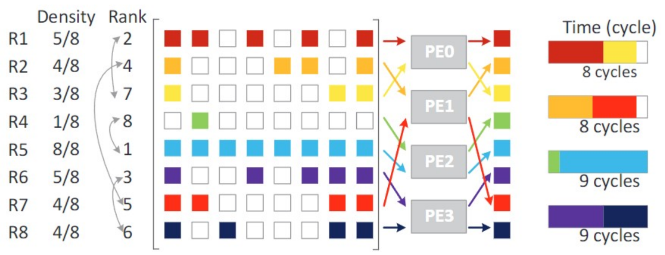


Fig. 8. An example of the proposed row grouping mechanism.

Specifically, using row-grouping as an example, the densities of each row are captured. Different from no workload balancing designs that distributes the rows to the PEs by the row index, we distribute each row by the density-sorted rank order. In our design, the dense rows and sparse rows are grouped together, so that groups with similar densities are created. Afterward, individual row groups are mapped to the processing elements in a way that the tasks allocated to each PE are completed simultaneously. An example of the row-grouping method is shown in Fig. 8. The size of the sparse matrix is 8×8. We assume there are four PEs in total so each PE will be allocated a 2×8 tile. The densities of each row, the density-sorted rank order of each row, and the utilization of the four PEs are also shown in the figure. When there is no workload balancing, the rows are distributed to the PEs by the row index (e.g., $R1$ and $R2$ to PE0, $R3$ and $R4$ to PE1). Since the row density varies, PEs with denser rows such as PE2 will take more cycles to complete the computation task, while the others have to idle because of PE synchronization. As shown in Fig. 8, to address this problem, we first group the rows by the density-sorted rank order, e.g., the densest row ($R5$) and the sparsest row ($R4$) will be in the same group, so that these row groups are similar in density (from 8/16 to 9/16). The grouping information is sent to the PE control unit. When the PEs complete their workloads, the control unit uses the grouping information to recover the outputs so that they are stored in the correct positions in the on-chip output buffer.

4.3 Dataflow Parameter Selection

4.3.1 *Search Space of Dataflow Parameters*

The dataflow of the chain Matrix Multiplications (Chain-MMs) [10] in GCNs can be generalized as six nested loops, as shown in Fig. 9. This significantly expands the GCN accelerators' design space in terms of parallelism, computation partitioning, and scheduling. In the proposed design, we select the suitable GEMM/SpMM dataflows within an individual aggregation or combination phase (Intra-phase dataflows) and the overall dataflow with both phases (Inter-phase dataflows).

Aggregation Dataflow: For aggregation dataflow, we decide loop order of three dimensions, namely vertex V, neighbors N, and input features F. The aggregation dataflow decides the data movement and data reuse [24]. As discussed in Sec. 2.1, one of the V, N, and F can be spatially mapped, while the other two can be temporally mapped. Assuming an aggregation dataflow of which V is the outermost loop and temporal, the innermost loop selection can lead to

```
//Inter-phase dataflow: Combination->Aggregation
//Combination
for(n = 0; n < N; n++){ //Outer Loop
    for(c = 0; c < C; c++){ //Middle Loop
        for(k = 0; k < K; k++){ //Inner Loop
            B[n][c] += X[n][k] * W[k][c];
        }
    }
}
//Aggregation
for(m = 0; m < M; m++){ //Outer Loop
    for(c = 0; c < C; c++){ //Middle Loop
        for(n = 0; n < N; n++){ //Inner Loop
            o[m][c] += A[m][n] * B[n][c];
        }
    }
}
```

Fig. 9. Six nested loops when the inter-phase dataflow is $A(XW)$.

different inter-PE traffic: if N is the innermost temporal loop, the aggregation can be done in a systolic manner, yet if F is the innermost temporal loop, the inter-PE traffic would be many-to-one [25]. Therefore, with different hardware constraints (number of PEs,inter-PE link bandwidth, etc.), different aggregation dataflows should be selected for optimized performance. Moreover, the aggregation dataflow can also be determined according to the hardware limitation. For example, if the GLB size is insufficient for a dimension, it must be temporal. In the proposed design, the data flow is dynamically selected by considering performance, energy, and hardware limitations simultaneously. The selection of dimensions for the outer loop, medium loop, and inner loop is performed using the greedy search algorithm as described in 4.3.2.

Combination Dataflow: For the combination phase, the three nested loops are vertex V, output features O, and input features F. For the combination phase, both the V × F and F × O matrices are streaming into the PEs. F is temporally mapped, while O and V are spatially mapped. Note that this dataflow is identical to an output stationary systolic array. However, the data can be transmitted through a systolic array or multicast, according to the available hardware resources. The selection of combination loop orders is performed using the greedy search algorithm as described in 4.3.2.

Inter-phase Dataflow: The inter-phase dataflow (the execution order of aggregation and combination) is defined as the dataflow that merges the data between aggregation and combination. This is important because it determines the number of memory accesses, data reuse, and the amount of intermediate on-chip buffers required to move data from one phase to the next. In our design, for the best GCN performance, we use a parallel inter-phase dataflow with a fixed ratio of PEs, which are decided in the previous step, and are allocated to different phases. An intermediate buffer (constructed with GLB banks) that connects the aggregation and combination engines with bi-directional links is required. However, because sub-accelerators have limited hardware resources, the intermediate buffer (GLB) size may not be always sufficient for parallel inter-phase dataflow. In this case, the PEs that computes both aggregation and combination will work sequentially. As discussed in Sec. 2, the ChainMM $A \times X \times W$ can execute aggregation or combination in the first phase and compute the rest in the second phase. For a sequential inter-phase dataflow, the output of the first phase is stored in the global buffer, and the PEs load necessary data back from the global buffer for the next phase.

4.3.2 Optimizing Design Variables with Hardware-Application Co-exploration

We propose a heuristic greedy search method [26] to decide the optimal combination of design variables, namely tile sizes, intra-phase loop order, inter-phase execution order, and loop unrolling, for concurrent GCNs, with the goal of maximizing overall performance and power efficiency under design constraints (e.g., PE numbers and buffer size).

The primary objectives of the greedy search method are threefold. First, it targets reducing the computation latency that is correlated with data sparsity and loop unrolling. Second, it tries to reduce the number of off-chip DRAM accesses that is related to GLB size and data reuse that rely on tile size and inter-phase execution order. Third, it also aims at reducing on-chip SRAM accesses that are related to loop unrolling and intra-phase dataflow. Therefore, to simultaneously achieve all design objectives, we define a *GCN accelerator cost factor* $\mathbb{O}$ by calculating the weighed log function of all three objectives. And the optimization algorithm searches the design space for the parameter values that can minimize $\mathbb{O}$. Assuming $A \in \mathbb{R}^{(M \times N)}$, $X \in \mathbb{R}^{(N \times K)}$, and $W \in \mathbb{R}^{(K \times C)}$, we define the intermediate matrix $\mathbb{I}$ as $A \times X$ or $X \times W$, and the output matrix is $O \in \mathbb{R}^{(M \times C)}$. The *GCN accelerator cost factor* for the i-th sub-accelerator $\mathbb{O}_i$ is:

$$\begin{aligned}
\underset{\mathbb{X}}{Min} \quad & \mathbb{O}_i = \log[Latency(\mathcal{X}^u)] + \log[\omega_1 \cdot V_d(\mathcal{X}^t, \mathcal{X}^{inter})] \\
& \quad + \log[\omega_2 \cdot V_s(\mathcal{X}^u, \mathcal{X}^{intra})] | N_{PE\_i}, GLB\_size_i \\
s.t. \quad & 0 < T_m \leq M, \quad 0 < T_k \leq K \\
& 0 < T_{n0} \leq N, \quad 0 < T_{n1} \leq N \\
& 0 < T_{c0} \leq C, \quad 0 < T_{c1} \leq C \\
& S_X + S_W + S_I \leq GLB\_size_i \\
& S_A + S_O + S_I \leq GLB\_size_i
\end{aligned} \tag{3}$$

We model the entire search space as $\mathbb{X} = \mathcal{X}^u \cup \mathcal{X}^t \cup \mathcal{X}^{inter} \cup \mathcal{X}^{intra}$. We define $\mathcal{X}^u$, $\mathcal{X}^t$, $\mathcal{X}^{inter}$, and $\mathcal{X}^{intra}$ as the parameter search spaces of loop unrolling factors, tile tuple, inter-phase dataflow (aggregation and combination order), and intra-phase dataflow (loop orders), respectively. Note that the interconnection of the proposed architecture dynamically morphs and adapts to the traffic. Therefore, it is not required to specially search the design space to decide the form of the interconnection network. The log functions of computation latency $Latency$, the DRAM access number V_d, and the SRAM access number V_s are accumulated. S_X, S_W, S_I, S_A, S_o are the size of on-chip buffer for the corresponding matrices. ω_1 is the adjustment parameter that estimates the energy consumption difference between DRAM access and the basic arithmetic operation, and ω_2 reflects the energy consumption difference between the arithmetic operation and SRAM access. Based on the energy model in [17], we select $\omega_1 = 206.5$ and $\omega_2 = 1.6$ to indicate that a single DRAM access and SRAM access induce 206.5×

TABLE 2
The proposed heuristic greedy search algorithm

Cases	Inter-phase Dataflow	Loop Order	Tile Size Setting Priority
$N \cdot C \geq GLB\_size_i$	Sequential	① $n_0 \rightarrow c_0 \rightarrow k$, $m \rightarrow c_1 \rightarrow n_1$; ② $m \rightarrow n_0 \rightarrow k$, $m \rightarrow k \rightarrow c_0$	①T_{n0}, Tm ②T_{c0}, T_{c1} ③T_{n1}, T_k
$N \cdot C < GLB\_size_i$	Parallel	$n_0 \rightarrow c_0 \rightarrow k \rightarrow m$	①T_{n0}, T_{n1} ②T_{c0}, T_{c1} ③T_m, T_k

and 1.6× higher energy consumption, as compared to a basic arithmetic operation, respectively.

Optimizing the proposed accelerator design by exploring the entire design space is costly, as the search space $\mathcal{X}^t \in [1, Dimension]$. Therefore, we propose to reduce the search space by pruning $\mathcal{X}^t$. In this paper, we deploy a heuristic greedy-search algorithm that leverages the empirical rules that are modeled by many simulation results [26] and can significantly reduce the search time of $\mathcal{X}^t$, thus achieving a total search time within tens of seconds. Specifically, we first compare the size of the on-chip buffer for the i-th sub-accelerator with $N \cdot C$ to determine the inter-phase dataflow, and then determine the tile size setting priority. The algorithm is listed in Table 2.

We use the priority parameter (priority levels 1, 2, and 3) for the setting of each tile size. The tile with higher priority can be set to a larger number than the tiles with lower priorities. Specifically, we first guarantee the tiles with the highest priority have the maximum sizes that are allowed by the design constraints. Then the tile sizes that have the second-highest priority will be set as large as possible. The rest will be utilized by the tiles with the lowest priority. To shrink the tile size search space $\mathcal{X}^t$, we only explore the tile size that can be divided by the dimension size with minimized padding of the data block [27]. This will also improve the GLB utilization, therefore reducing off-chip DRAM accesses. By doing so, the size of the search space is reduced from $O(N)$ to $O(2\sqrt{N})$.

5 EXPERIMENTAL METHODOLOGY

We built a cycle-accurate simulator with reconfigurable design variables in C++ that captures the microarchitectural behavior of the hardware design of the proposed accelerator to evaluate the performance. For the accelerator chip, the simulator models the behaviors of all the accelerator logic, namely the MAC array, SMB/IDMB/ODMB, FIFO buffers, DRP. The access latency of the buffers is estimated using Cacti [28]. To estimate the timing and power consumption of the off-chip DRAM accesses, the simulator deploys a DRAM counter to monitor the read/write counts of SRAMs and FIFOs. The simulator is able to estimate the latency of the GCN tasks measured by the number of cycles. The energy consumption of these DRAM read/write operations is modeled by [17]. The area consumption is estimated using the Synopsys Design Compiler with 40nm library, and the power consumption is captured using Synopsys PrimeTime PX.

TABLE 3
Hardware Configurations of the Proposed Design.

PE	Value	Overall	Value
Multiplier	64 bits	Num. of PEs	256 (16 × 16)
MAC Array	1*16	Num. of Routers	256 (16 × 16)
FIFO	16 entries	Global Buffer	32MB
SMB	320 KB	DRAM	HBM, 126GB/s
IDMB	4 KB	Interconnection Network	Morphable
ODMB	256 KB		

Table 3 lists the configurations of the proposed design that we used in the simulation. The proposed accelerator consists of 256 PEs. Each PE is comprised of a 1 × 16 MAC Array using double-precision floating-point multipliers, so that the accelerator could potentially be extended for broader applications like general-purpose sparse matrix-matrix applications, which requires high-precision computations. The sizes of SMB, IDMB, ODMB are 320 KB, 4 KB, and 256 KB, respectively. We manually selected the sizes for these local buffers to ensure a fit for the matrix tiles. The global buffer (GLB) size is 32 MB. We implement a morphable interconnection network with 256 morphable routers.

We compare the proposed design with a PyTorch Geometric [29] baseline run on Intel Xeon 8168 CPU (denoted as *PyG/CPU*) and several state-of-the-art GCN accelerators, namely HyGCN, AWB-GCN, LW-GCN, GCoD, and GCNAX. Table 4 summarizes the characteristics of these baselines. The PyG/CPU has 24 cores operating at 2.7GHz with a memory bandwidth of 128.1 GB/s. The above SOTA accelerators operate at 1GHz and have 126GB/s off-chip memory.For a fair comparison, the baseline accelerators are scaled so that they integrate the same number of multipliers and DRAM bandwidth as the proposed design.

We conduct five tests using a mix of the benchmark datasets that are frequently used in the literature [30], [31], [32], [33], namely Cora, Citeseer, Pubmed, Nell, and Reddit. Each dataset has diverse vertex numbers, edge numbers, feature lengths, and data density. Each test consists of several datasets, and each dataset can be executed multiple times so that the workload of each dataset is roughly balanced. By doing so, the performance of one dataset will not dominate the entire test. The five tests are conducted as follows:

1) *Test-1 (Two Datasets):* Cora(×1), Citeseer(×1).
2) *Test-2 (Two Datasets):* Nell(×5), Reddit(×1).
3) *Test-3 (Three Datasets):* Cora(×5), Citeseer(×5), and Pubmed(×1).
4) *Test-4 (Four Datasets):* Cora(×25), Citeseer(×25), Pubmed(×5), and Nell(×1).
5) *Test-5 (Five Datasets):* Cora(×25), Citeseer(×25), Pubmed(×5), Nell(×1), and Reddit(×1).

6 EVALUATION AND ANALYSIS

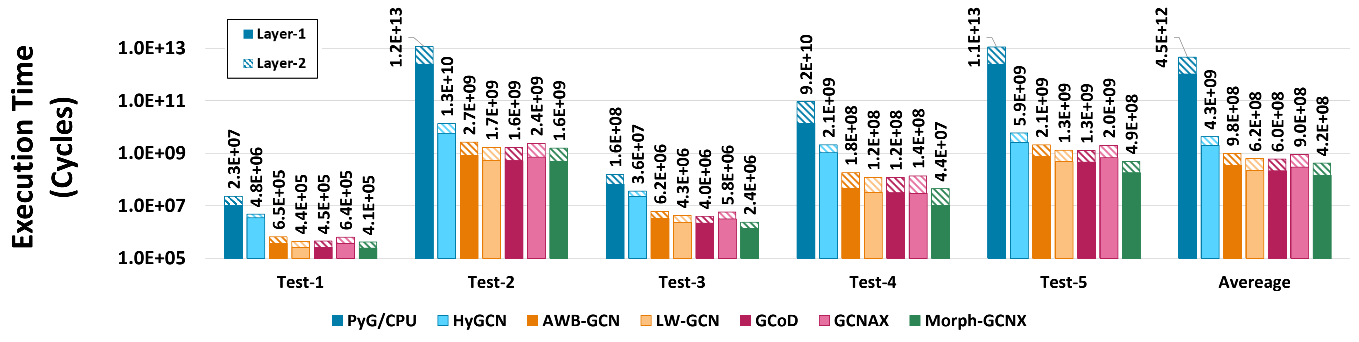
6.1 Performance

Fig. 10 compares the performance of the proposed design and the baselines measured by the total number of execution cycles and speedup. On average, the proposed design is 18.8×, 2.9×, 1.9×, 1.8×, and 2.5× faster than HyGCN,

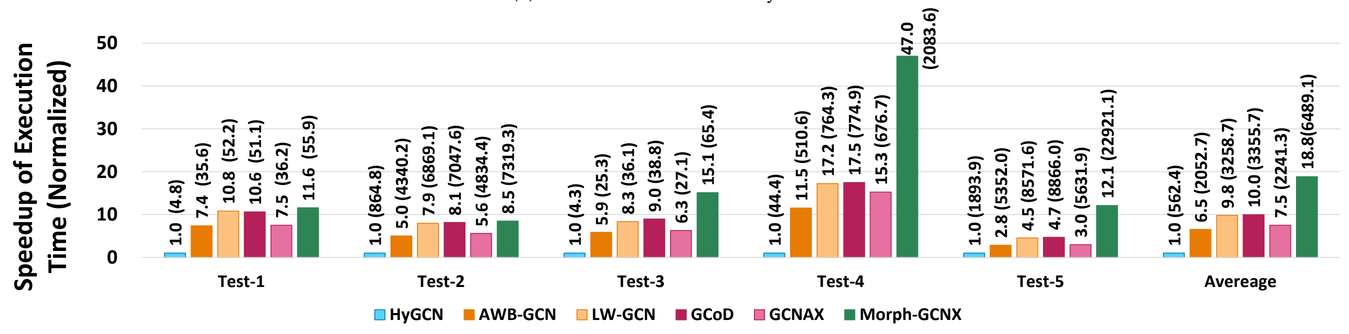
TABLE 4
Summary of different GCN accelerators.

Design	Processing Engine	Inter-phase Dataflow	Intra-phase Dataflow	Tile size	Interconnection
PyG/CPU	Tandem	$(AX)W$	Static	Static	Fixed
HyGCN	Tandem	$(AX)W$	Static	Static	Fixed
AWB-GCN	Uniform	$A(XW)$	Static	Static	Fixed
LW-GCN	Uniform	Adaptive	Static	Adaptive	Fixed
GCoD	Tandem	$A(XW)$	Static	Static	Fixed
GCNAX	Uniform	$A(XW)$	Adaptive	Adaptive	Fixed
Morph-GCNX	Uniform	Adaptive	Adaptive	Adaptive	Flexible

† As HyGCN, AWB-GCN, and GCNAX do not support concurrent execution of multiple GCN tasks, the GCN applications are executed sequentially.



(a) Full execution time in cycles.



(b) Overall Speedup of execution time, normalized to HyGCN. The values in the parentheses shows the speedup of each design over the PyG/CPU baseline.

Fig. 10. Performance analysis of the proposed design and the baseline accelerators.

AWB-GCN, LW-GCN, GCoD, and GCNAX, respectively. The proposed design outperforms the other designs on all five tests. HyGCN has the worse performance due to the use of tandem compute engines for aggregation and combination, respectively. This inevitably incurs performance loss when accommodating different datasets with different computational requirements for the aggregation and combination engines, as HyGCN can only achieve its optimal performance by carefully orchestrating the computational capacity of the combination and aggregation engines for a given dataset. AWB-GCN addresses this problem by using uniformed compute engines and achieves 2.3× speedup. LW-GCN further improves the performance with adaptive inter-phase dataflow. GCoD achieves similar speedup to LW-GCN because it enables more balanced workloads and PE utilization. GCNAX achieves an additional 14% execution time reduction over AWB-GCN thanks to the dynamic intra-phase dataflow design that increases data reuse. The proposed design achieves the highest speedup thanks to the flexible designs. Specifically, the flexible inter-phase and intra-phase dataflow design rearranges the execution order of the chainMM and reduces the total number of computational operations. Moreover, the flexible dataflow also allows the PEs to fully leverage the on-chip SRAMs without frequently accessing the offline DRAM. Finally, the morphable interconnection design adapts to the dynamically changing communication patterns thus reducing latency.

6.2 DRAM Accesses

Fig. 11 shows the number of DRAM accesses of the SIX accelerators. Overall, on average, the proposed design achieves 10.8×, 3.7×, 2.2×, 2.5×, and 1.3× reduction on DRAM accesses, as compared to HyGCN, AWB-GCN, LW-GCN, GCoD, and GCNAX, respectively. This is because our proposed design deploys the dynamic partitioning strategy, fine-tuned tile size tuple, optimized dataflow selection, and tailored interconnection network. Specifically, the flexible dataflow design and adaptive-tile size selection enable the GCN tasks to be executed in an efficient order. The morphable interconnection network can provide efficient inter-

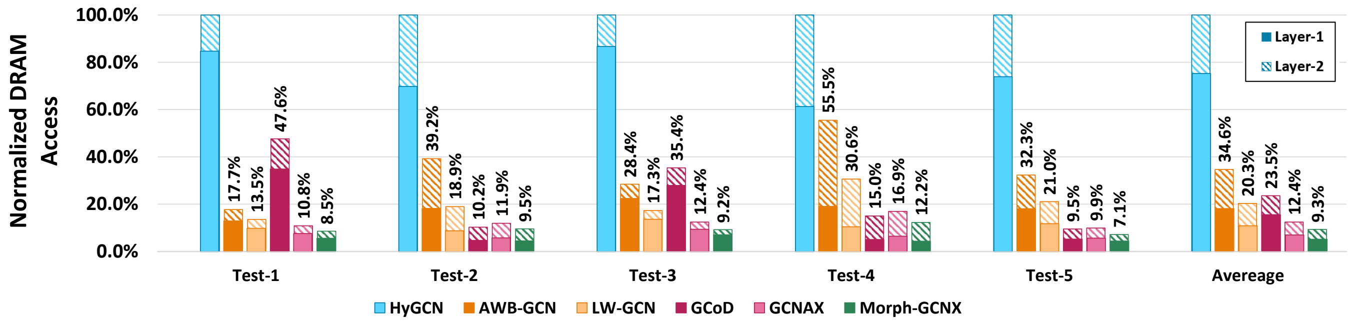


Fig. 11. DRAM access comparison, normalized to HyGCN (lower is better).

PE communication and involve more data reuse, which leads to fewer computations and DRAM accesses.

6.3 Energy Consumption

Fig. 12 shows the energy consumption of different accelerators, normalized to the energy consumption of the proposed Morph-GCNX design. Overall, the proposed design achieves 13.2×, 5.6×, 2.1×, 2.5×, and 1.3× energy savings over HyGCN, AWB-GCN, LW-GCN, GCoD, and GCNAX, respectively, thanks to the reduced DRAM accesses and improved computing resource utilization. Moreover, it should be noted that although the morphable interconnection design in the proposed accelerator does not reduce DRAM access numbers in the proposed Morph-GCNX design, it provides improved data transmission and data reuse between PEs, thus SRAM access number is reduced. We also compares the proposed Morph-GCNX design to PyG/CPU in terms of energy consumption(for readability, we do not show these results in Fig. 12). The proposed design achieves 942.3×, 2037.9×, 983.2×, 2740.5×, and 4340.5× energy savings in tests 1 to 5, respectively.

6.4 Energy-delay Product (EDP)

We use EDP to show if the energy improvement is achieved by sacrificing execution time, as shown in Fig. 13. Results are normalized to the EDP of the proposed Morph-GCNX design. As shown in Fig. 13, on average, the proposed design achieves 235.4×, 16.4×, 4.2×, 3.9×, and 3.3× energy savings over HyGCN, AWB-GCN, LW-GCN, GCoD, and GCNAX, respectively.

6.5 Area Consumption

We evaluate the area consumption of the proposed architecture under TSMC 40 nm technology, as shown in Table 5. As shown in Table 5, the MAC array consumes only 7.1% of the total PE area, while the memory hierarchy, SMB and IDMB/ODMB, consumes a majority fraction, 82.9%, of the total area. The PE control unit consumes 3.7% of the total PE area. For the entire proposed accelerator, the PE array, which consists of 256 PEs consumes a major fraction of the overall chip area. The controller consumes negligible area overhead. The optimizer consumes 11.3 mm^2 area, which implies 0.5% area overhead. The additional components for the morphable interconnection consume 233.4 mm^2 area, which implies 10.0% total chip area.

TABLE 5
Area Consumption of the Proposed Design

PE Components	Area (mm^2)
MAC array	0.46 (7.1%)
SMB	2.60 (39.9%)
IDMB	0.10 (1.5%)
ODMB	2.70 (41.5%)
DRP	0.41 (6.3%)
PE Control Unit	0.24 (3.7%)
Total	6.51 (100%)
Overall	Area (mm^2)
PEs	1666.6
GLB	269.4
Controller	2.24
Optimizer	11.3
Morphable Interconnection	233.4

7 Related Work

There has been considerable work devoted to accelerating matrix multiplication, which is the key computation pattern in GCNs. In what follows we briefly highlight some of the directly relevant work.

Sparse Matrix Multiplication Accelerators. Existing designs exploit CPU-based [34], [35], [36], GPU-based [37], [38], [39], [40], FPGA-based [41], [42], and ASIC-based [43], [44], [45] solutions for accelerating sparse matrix multiplication (SpGEMM and SpMM). CPU- and GPU-based solutions [34], [35], [36], [37] are mostly based on a multiply-insert basis to increase computational parallelism. However, these techniques utilize general-purpose computing units thus result in exceeding power and latency. FPGA-based solutions [41], [42] introduce tailored PE and interconnection architectures to improve data locality and energy efficiency. OuterSpace [43] proposes an outer-product-based SpGEMM with enhanced input reuse. SpArch [44] introduces a specialized ASIC accelerator design to improve data reuse of both the input matrix and the output matrix. However, these solutions provide limited insight in how to efficiently support chain matrix multiplication in GCNs.

Neural Network Accelerators. General Neural Network Accelerators [46], [47], [48], [49], [50], [51], [52] can be used for accelerating matrix multiplication in GCNs. For instance, accelerators that leverage massive parallelism [46], [47] can be beneficial to the GEMM and DenseMM operations that exist in the combination phase of GCNs, while other accelerators [48], [49], [50], [51] that aim to reduce computation operations for sparse matrices can deliver performance improvements to SpMM and SpGEMM in the aggregation

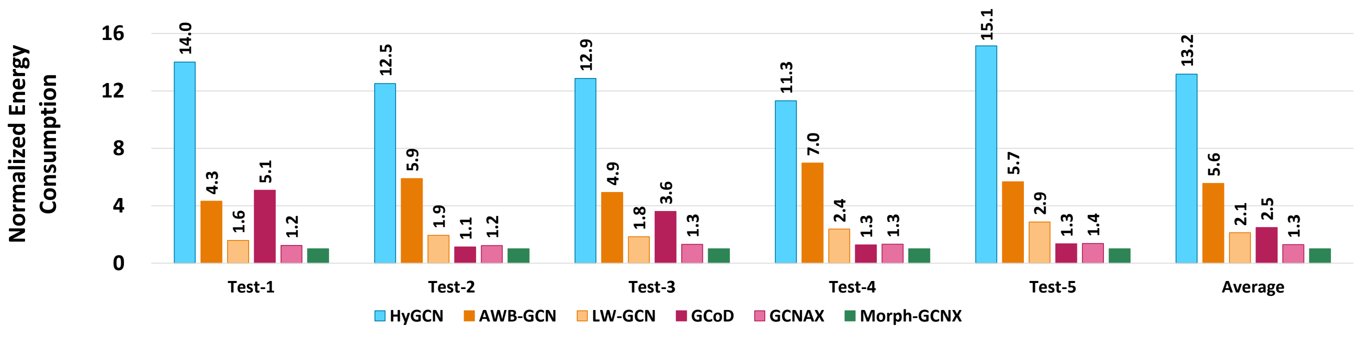


Fig. 12. Energy consumption comparison, normalized to the proposed Morph-GCNX design (lower is better).

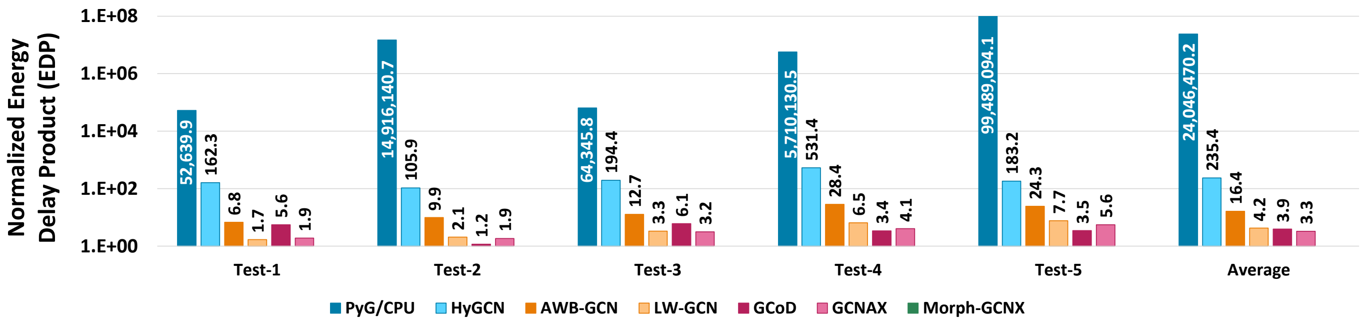


Fig. 13. Energy delay product (EDP) comparison, normalized to the proposed Morph-GCNX design (lower is better).

phase. Unfortunately, few of the existing general neural network accelerators have exploited both directions simultaneously to accelerate the chain matrix multiplication in GCNs that consists of both sparse and dense matrix multiplication operations.

Graph Convolutional Network Accelerators. GCN accelerators are tailored for accelerating chain matrix multiplications. HyGCN [9] exploits two dedicated compute engines for aggregation and combination. However, the rigid design of the two engines can lead to workload imbalance. AWB-GCN [10] therefore addresses this problem by leveraging a unified engine. Auten *et al.* [11] accelerates GCN executions using an accelerator design that handles irregular data movement efficiently. GCoD [14] develop a dedicated two-pronged accelerator with a separated engine to process each of the aforementioned denser and sparser workloads. LW-GCN [15] decomposes the main GCN operations into Sparse Matrix-Matrix Multiplication (SpMM) and Matrix-Matrix Multiplication (MM). LW-GCN proposes a workload balancing algorithm and applies data quantization and workload tiling to map both SpMM and MM of GCN inference onto a uniform architecture. EnGN [53] abstracts the typical computing patterns for some GCN applications and accelerates the key stages of GCN propagation GRIP [54] splits the GCN inference into two phases (vertex-centric and edge-centric execution) and designed specialized hardware units for each phase. GraphACT [55] is dedicated to the acceleration of GCN training on heterogeneous systems using CPU and FPGA, which incorporates multiple architecture-algorithm co-optimizations. GCNAX [16] and SGCNAX [26] propose a flexible GCN architecture that supports flexible dataflow to improve resource utilization. Although existing GCN accelerators achieve considerable performance and energy efficiency improvements, these designs are optimized for specific GCN models, and thus have restricted flexibility. As different GCNs prefer different data reuse and parallelization strategies to achieve optimal efficiency, these architectures would degrade the performance when accommodating multiple GCNs.

8 Conclusions

As Graph Convolutional Networks (GCNs) are both memory-intensive and compute-intensive, customized accelerators for GCNs have been proposed to deliver substantial performance speedups. However, as different GCNs prefer different data reuse and parallelization strategies to achieve optimal efficiency, existing architectures that are tailored for specific GCNs can degrade the performance when accommodating multiple GCN applications. To address this limitation, we propose a morphable GCN accelerator design framework that adapts to diverse GCNs for improved performance and energy. The proposed accelerator consists of a flexible PE array design and a morphable interconnection design to support a wide range of GCN dataflows with various parallelization and data reuse strategies for multi-GCN execution. Specifically, the proposed accelerator can be partitioned into multiple sub-accelerators with different sized PE-array, global buffers, and interconnects. These components can be configured to support any desired inter-/intra-dataflows needed by the GCN tasks. As compared to existing monolithic designs that have fixed control policies for PE partition, dataflow parameters (i.e., tile sizes, inter- and intra- phase dataflows, and buffering strategies), the proposed Morth-GCNX can be dynamically reconfigured to meet the diverse computation and communication requirements of different GCNs. We also propose a hardware-application co-exploration technique that explores the GCN and hardware design spaces to identify the best PE partition, workload allocation, dataflow, and interconnection configurations, with the goal of improving overall performance and energy. Simulation results show that the proposed design achieves better performance, reduced DRAM accesses, and lower energy consumption as compared to state-of-the-art accelerator designs. In this paper, we demonstrate the amplifying and synergistic effects of integrating architectural innovations with optimization algorithm designs in a morphable GCN accelerator design. The proposed architecture can be automatically tailored and adapt to diverse GCN applications' communication and computation requirements for improved general matrix multiplications (SpMM and GEMM) in the aggregation and combination phases. In the future, we will expand our hardware design to support a wide range of evolving GNN models that integrates other operations such as edge embedding, attention, mixed neighborhood aggregation, and many others. We will further explore the use of more efficient optimization algorithms, including machine learning, to further improve performance and reduce overheads.

ACKNOWLEDGMENTS

This research was partially supported by National Science Foundation (NSF).

REFERENCES

[1] Chong-Yaw Wee et al. Cortical graph neural network for ad and mci diagnosis and transfer learning across populations. *NeuroImage: Clinical*, 23:101929, 2019.

[2] Tzu-An Song et al. Graph convolutional neural networks for alzheimer's disease classification. In *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*, pages 414–417. IEEE, 2019.

[3] Shui-Hua Wang, Vishnu Varthanan Govindaraj, Juan Manuel Górriz, Xin Zhang, and Yu-Dong Zhang. Covid-19 classification by fgcnet with deep feature fusion from graph convolutional network and convolutional neural network. *Information Fusion*, 67:208–229, 2021.

[4] Xiaoyang Wang et al. Traffic flow prediction via spatial temporal graph neural network. In *Proceedings of The Web Conference 2020*, pages 1082–1092, 2020.

[5] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *arXiv preprint arXiv:2101.11174*, 2021.

[6] 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*, pages 1711–1719, 2020.

[7] Yantao Shen, Hongsheng Li, Shuai Yi, Dapeng Chen, and Xiaogang Wang. Person re-identification with deep similarity-guided graph neural network. In *Proceedings of the European conference on computer vision*, 2018.

[8] Ao Luo et al. Cascade graph neural networks for rgb-d salient object detection. In *Proceedings of European Conference on Computer Vision*, pages 346–364. Springer, 2020.

[9] M. Yan et al. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29, 2020.

[10] Tong Geng et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[11] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[12] Lei He. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *arXiv preprint arXiv:1909.00155*, 2019.

[13] Xinkai Song et al. Cambricon-g: A polyvalent energy-efficient accelerator for dynamic graph neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

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

[15] Zhuofu Tao, Chen Wu, Yuan Liang, Kun Wang, and Lei He. Lw-gcn: A lightweight fpga-based graph convolutional network accelerator. *ACM Trans. Reconfigurable Technol. Syst.*, 16(1), dec 2022.

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

[17] Mark Horowitz. Energy table for 45nm process, 2012.

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

[19] Mohammad Rasool Izadi, Yihao Fang, Robert Stevenson, and Lizhen Lin. Optimization of graph neural networks with natural gradient descent. In *2020 IEEE international conference on big data (big data)*, pages 171–179. IEEE, 2020.

[20] 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*, pages 869–877, 2018.

[21] 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.

[22] D. DiTomaso, A. Kodi, and A. Louri. Qore: A fault tolerant network-on-chip architecture with power-efficient quad-function channel (qfc) buffers. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 320–331. IEEE, 2014.

[23] 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. *arXiv preprint arXiv:2108.11571*, 2021.

[24] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *field programmable gate arrays*, pages 161–170, 2015.

[25] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 461–475, 3173176, 2018.

[26] Jiajun Li, Hao Zheng, Ke Wang, and Ahmed Louri. Sgcnax: A scalable graph convolutional neural network accelerator with workload balancing. *IEEE Transactions on Parallel & Distributed Systems*, (01):1–1, 2021.

[27] Qi Nie. Memory-driven data-flow optimization for neural processing accelerators. 2020.

[28] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to understand large caches. *HP Laboratories*, 2009.

[29] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[30] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[31] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[32] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[33] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka, and Tom M Mitchell. Toward an architecture for

never-ending language learning. In *Twenty-Fourth AAAI conference on artificial intelligence*, 2010.

[34] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[35] Nathan Bell and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations. *Version 0.3. 0*, 35, 2012.

[36] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016.

[37] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381, 2014.

[38] Rajib Nath, Stanimire Tomov, Tingxing" Tim" Dong, and Jack Dongarra. Optimizing symmetric dense matrix-vector multiplication on gpus. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2011.

[39] Yusuke Nagasaka, Akira Nukada, Ryosuke Kojima, and Satoshi Matsuoka. Batched sparse matrix multiplication for accelerating graph convolutional networks. *arXiv preprint arXiv:1903.11409*, 2019.

[40] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. *Acm Sigplan Notices*, 46(8):267–276, 2011.

[41] Colin Yu Lin, Ngai Wong, and Hayden Kwok-Hay So. Design space exploration for sparse matrix-matrix multiplication on fpgas. *International Journal of Circuit Theory and Applications*, 41(2):205–219, 2013.

[42] Ling Zhuo and Viktor K Prasanna. Sparse matrix-vector multiplication on fpgas. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, 2005.

[43] Subhankar Pal et al. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[44] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.

[45] E. Montagne and R. Surós. Systolic sparse matrix vector multiply in the age of tpus and accelerators. In *2019 Spring Simulation Conference (SpringSim)*, pages 1–10, 2019.

[46] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12, 3080246, 2017. ACM.

[47] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 367–379. IEEE, 2016.

[48] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: Ineffectual-neuron-free deep neural network computing. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 1–13. IEEE Press, 2016.

[49] Shijin Zhang et al. Cambricon-X: An accelerator for sparse neural networks. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–12. IEEE, 2016.

[50] Angshuman Parashar et al. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 27–40. ACM, 2017.

[51] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–165, 3358291, 2019. ACM.

[52] Song Han et al. EIE: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 243–254. IEEE Press, 2016.

[53] Shengwen Liang et al. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 2020.

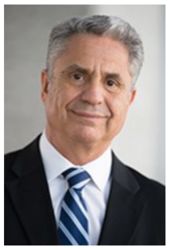
[54] Kevin Kiningham, Christopher Re, and Philip Levis. Grip: A graph neural network acceleratorarchitecture. *arXiv preprint arXiv:2007.13828*, 2020.

[55] Hanqing Zeng and Viktor Prasanna. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 255–265, 2020.

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.

Hao Zheng received the Ph.D. in Computer Engineering from George Washington University, Washington, DC, in 2021. He is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Central Florida. His research interests are in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks, AI chips for emerging applications, and energy-efficient manycore architectures.

Jiajun Li received the B.E. degree from the Department of Automation, Tsinghua University in 2013. He received the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences in 2019. From 2019 to 2021, he was a postdoc researcher with the Department of Electrical and Computer Engineering, George Washington University. He is currently an Associate Professor with the School of Astronautics, Beihang University. His current research interests include machine learning and heterogeneous computer architecture.



Ahmed Louri is the David and Marilyn Karlgaard Endowed Chair Professor of Electrical and Computer Engineering at the George Washington University, which he joined in August 2015. He is also the director of the High Performance Computing Architectures and Technologies Laboratory. Dr. Louri received the Ph.D. degree in Computer Engineering from the University of Southern California, Los Angeles, California in 1988. From 1988 to 2015, he was a professor of Electrical and Computer Engineering at the University of Arizona, and during that time, he served six years (2000 to 2006) as the Chair of the Computer Engineering Program. From 2010 to 2013, Dr. Louri 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. Dr. Louri conducts research in the broad area of computer architecture and parallel computing, with emphasis on interconnection networks, optical interconnects for scalable 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 the 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 manycore architectures." Dr. Louri is a Fellow of the IEEE, and he is currently the Editor-in-Chief of the IEEE Transactions on Computers. More information can be found at https://hpcat.seas.gwu.edu/Director.html.