SGCNAX: A Scalable Graph Convolutional Neural Network Accelerator With Workload Balancing

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

Abstract—Convolutional Neural Networks (GCNs) have emerged as promising tools for graph-based machine learning applications. Given that GCNs are both compute- and memory-intensive, this constitutes a major challenge for the underlying hardware to efficiently process large-scale GCNs. In this article, we introduce SGCNAX, a scalable GCN accelerator architecture for the high-performance and energy-efficient acceleration of GCNs. Unlike prior GCN accelerators that either employ limited loop optimization techniques, or determine the design variables based on random sampling, we systematically explore the loop optimization techniques for GCN acceleration and propose a flexible GCN dataflow that adapts to different GCN configurations to achieve optimal efficiency. We further propose two hardware-based techniques to address the workload imbalance problem caused by the unbalanced distribution of zeros in GCNs. Specifically, SGCNAX exploits an outer-product-based computation architecture that mitigates the intra-PE (Processing Elements) workload imbalance, and employs a group-and-shuffle approach to mitigate the inter-PE workload imbalance. Simulation results show that SGCNAX performs 9.2×, 1.6× and 1.2× better, and reduces DRAM accesses by a factor of 9.7×, 2.9× and 1.2× compared to HyGCN, AWB-GCN, and GCNAX, respectively.

Index Terms—Graph convolutional neural networks, dataflow accelerators, domain-specific accelerators, memory access optimization

1 Introduction

Recently, deep learning over graph data has achieved great success in a broad range of applications, such as traffic prediction [1], [2], object detection [3], [4], [5], [6], disease classification [7], [8], [9], and many others. One of the most successful models is Graph Convolutional Neural Network (GCN) [10], [11], [12], [13] that re-defines the notion of *convolution* for graph data. This model has been widely used in a variety of data centers including Google, Alibaba [14], and Facebook [15].

Much like traditional neural networks, training and inference of GCNs are both compute- and memory-intensive but impose new requirements on designing the underlying hardware architecture. Specifically, the execution time for graph convolutional layers is mainly dominated [16], [17] by two primary phases: *Aggregation* and *Combination*. The computation in the combination phase is similar to that in conventional neural networks. However, the aggregation phase depends on the graph structure which is often sparse and irregular. The sparsity and irregularity would be the new challenges facing the physical design of GCN architectures.

• *The authors are with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC 20052 USA. E-mail: {lijiajun, haozheng, cory, louri}@gwu.edu.*

Manuscript received 27 Aug. 2021; revised 30 Oct. 2021; accepted 24 Nov. 2021. Date of publication 10 Dec. 2021; date of current version 23 May 2022. This work was supported by National Science Foundation under Grants CCF-1702980, CCF-1812495, CCF-1901165, and CCF-2131946. (Corresponding author: Jiajun Li.) Recommended for acceptance by A.J. Peña, M. Si and J. Zhai. Digital Object Identifier no. 10.1109/TPDS.2021.3133691

The key computation function in GCNs can be abstracted as chain Sparse-dense Matrix Multiplications (SpMMs) [18]. It involves six nested loops that gives rise to a large design space for GCN accelerators in terms of parallelism, computation partitioning and scheduling. A number of loop optimization techniques [19], such as loop tiling, loop unrolling, loop interchange, and loop fusion can be used for GCNs. Recently, a few customized GCN accelerators [18], [20], [21] have leveraged these techniques to deliver gains in performance and energy efficiency. However, none of them has systematically studied the impact of these techniques on system efficiency in terms of latency and loop optimization techniques, or determine the design variables based on random sampling. As a result, they can hardly exploit data reuse efficiently, leading to increased memory accesses and performance loss.

In this paper, we propose SGCNAX, a scalable GCN accelerator architecture for high-performance and energy-efficient acceleration of GCNs. Specifically, the contributions of this paper are:

First, we provide an in-depth analysis of the four loop optimization techniques for GCN computation and use corresponding design variables to characterize the GCN dataflow. We then build analytical models to quantitatively estimate the design objectives of GCN dataflow, such as latency and the number of off-chip DRAM accesses. We found that different GCN configurations require different design variables of the dataflow to achieve optimal efficiency. Therefore, we propose a flexible dataflow that can reconfigure the design variables to adapt to different GCN configurations.

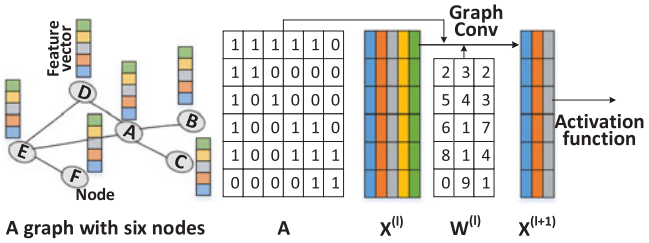

Fig. 1. Illustration of a GCN layer. The graph contains six nodes. A: adjacency matrix, $X^{(l)}$: feacture vectors of layer l, $W^{(l)}$: weight matrix of layer l, $X^{(l+1)}$: feacture vectors of layer $l+1$.

Second, we propose a hardware accelerator called SGCNAX to support the flexible dataflow. We observe that current accelerators incur performance losses because of the workload imbalance caused by the extremely sparse and unbalanced matrices in GCNs. Specifically, the current accelerators suffer from two forms of workload imbalances. The first is intra-PE (Processing Elements) workload imbalance when some Multiply-and-Accumulate (MAC) units are processing zeros while others are processing non-zeros, leading to the under-utilization of MAC units. The second is inter-PE workload imbalance, which stems from the way the dataflow partitions the work across the PEs. Since some PEs may complete the workload earlier as they are allocated sparser workloads, they have to sit idle while waiting for the laggards because of inter-PE synchronization. The proposed SGCNAX tackles both imbalances by employing two novel hardware/software co-design techniques. Specifically, SGCNAX employs an outer-product-based computation architecture for SpMMs [22], to mitigate the intra-PE workload imbalance. For the inter-PE workload imbalance, SGCNAX employs a group-and-shuffle computing approach, which groups the rows in a sparse matrix by density so that the row groups are similar in density, and then maps the rows within a group to the PEs so that all PEs complete the tasks simultaneously.

We implement the SGCNAX accelerator in RTL targeting TSMC 40 nm library. We also build a cycle-level simulator that models the microarchitectural behavior of each module while supporting different dataflows. Evaluated on five real-world graph datasets, SGCNAX performs 9.2×, 1.6× and 1.2× better, and reduces DRAM accesses by a factor of 9.7×, 2.9× and 1.2× compared to HyGCN, AWB-GCN, and GCNAX, respectively.

2 Background

2.1 GCN Computation

The typical structure of a graph convolutional layer is illustrated in Fig. 1. The prevalent computation pattern of the GCN models [13], [23], [24] can be abstracted as a chain SpMM

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

where $X^{(k)}$ is the matrix of input features in layer k; each column of X represents a feature vector while each row denotes a node. $W^{(k)}$ is the weight matrix of layer k. $\sigma(\cdot)$ denotes the non-linear activation function such as ReLU. $\hat{A}$ is a transformed matrix from the graph adjacency matrix. The transformation function varies across different GCN models.

TABLE 1
Notations in GCNs

Term	Meaning
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
$N(v)(S(v))$	(sampling subset of) neighbor set of v
$A(A_{ij})$	(element of) adjacency matrix
a_v	aggregation feature vector of v
h_v	feature vector of v
b	combination bias vectors
X	feature matrix composed by feature vectors

Since $\hat{A}$ can be computed offline from A, we hereafter use A to denote the normalized $\hat{A}$. Table 1 lists the notations used in GCNs.

The chain SpMM in GCNs consists of six loops as shown in the pseudo-code in Fig. 2. We assume that $A \in \mathbb{R}^{M\times N}$, $X \in \mathbb{R}^{N\times K}$, $W \in \mathbb{R}^{K\times C}$. Matrix $B \in \mathbb{R}^{N\times C}$ is the intermediate result of $X \times W$ and $O \in \mathbb{R}^{M\times C}$ is the final output matrix. We assume that we use the execution order of $A \times (X \times W)$ as it reduces the number of computations for most graph datasets [18].

2.2 GCN Accelerators

Recently, a few GCN accelerators have been proposed, which provide substantial improvements in performance and energy efficiency compared to generic CPU- and GPU-based solutions. Specifically, HyGCN [20] exploits two dedicated compute engines, i.e., an aggregation engine and a combination engine, to accelerate the *Aggregation* and *Combination* phases, respectively. AWB-GCN [18] is an architecture for accelerating GCNs and Sparse-dense Matrix Multiplication (SpMM) kernels, and addresses the issue of workload imbalance in processing real-world graphs. GCNAX [21] proposes a flexible dataflow for GCNs that simultaneously improves resource utilization and reduces data movement.

These accelerators can be illustrated by the typical architecture shown in Fig. 3. It consists of an accelerator chip and off-chip memory (usually DRAM). The accelerator chip is composed of a Processing Unit (PU), a global buffer (GLB), and a Scheduler. The PU can support high compute parallelism for

```
//SpMM1:B=XW
  for(n=0; n<N; n++) {→ Loop-1
    for(c=0; c<C; c++) {→ Loop-2
      for(k=0; k<K; k++) {→ Loop-3
        B[n][c]+=X[n][k]*W[k][c];
  }}}

//SpMM2:O=AB
  for(m=0; m<M; m++) {→ Loop-4
    for(c=0; c<C; c++) {→ Loop-5
      for(n=0; n<N; n++) {→ Loop-6
        O[m][c]+=A[m][n]*B[n][c];
  }}}
```

Fig. 2. Pseudo code of the chain SpMM in GCNs.

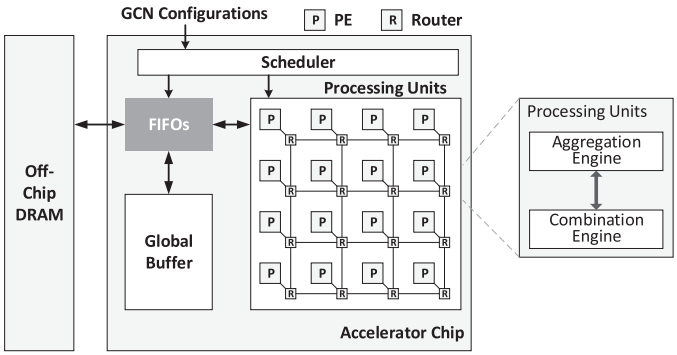

Fig. 3. A typical GCN accelerator architecture.

matrix multiplications, either consisting of two separate engines (HyGCN) or one uniform engine (AWB-GCN, GCNAX). The scheduler is used to map the GCNs onto the proposed accelerator using the computation sequences defined by the loop optimization techniques. GLB is usually a uniform software-controlled SRAM scratchpad memory that can be used to exploit input data reuse and hide DRAM access latency, or to store intermediate data. The accelerator often has three levels of memory hierarchy, including DRAM, GLB, and local registers in PEs. The energy consumption for data access depends on the memory hierarchy [25]. In this paper, we focus on the expensive off-chip DRAM accesses (between off-chip DRAM and GLB).

Although these GCN accelerators delivered considerable performance gains, few of them have considered the scalability of GCN accelerators. As GCN accelerators typically employ an array of multiply-and-accumulate (MAC) units to compute matrix multiplications, scaling the performance of accelerators would be determined by how well we handle the increased number of MAC units. To achieve this, we can either employ a monolithic array with a large number of MAC units, or employ more arrays. In this paper, we will investigate the scalability problem of GCN accelerators.

3 GCN Dataflow

A GCN's dataflow defines how the loops are ordered, partitioned, and parallelized. That being said, the chain SpMMs can be manipulated (e.g., ordered and partitioned.) to capture different data reuse opportunities. In this paper, we will investigate four loop optimization techniques, namely loop unrolling, loop tiling, loop interchange, and loop fusion, to optimize GCN's dataflow.

3.1 Loop Optimization Techniques

3.1.1 Loop Unrolling

Loop unrolling determines the parallelization strategy of the GCN loops, which determines the PE array scale and organization as well as the size of registers in each PE. It can be used to increase the utilization of massive computation resources. Researchers have extensively studied the methods to unroll SpMM for parallel computations. As illustrated in Fig. 4 which takes SpMM1 as an example, unrolling different loops directs parallelization of different computations, which affects the optimal PE array architecture with respect to the data reuse opportunities and memory access patterns.

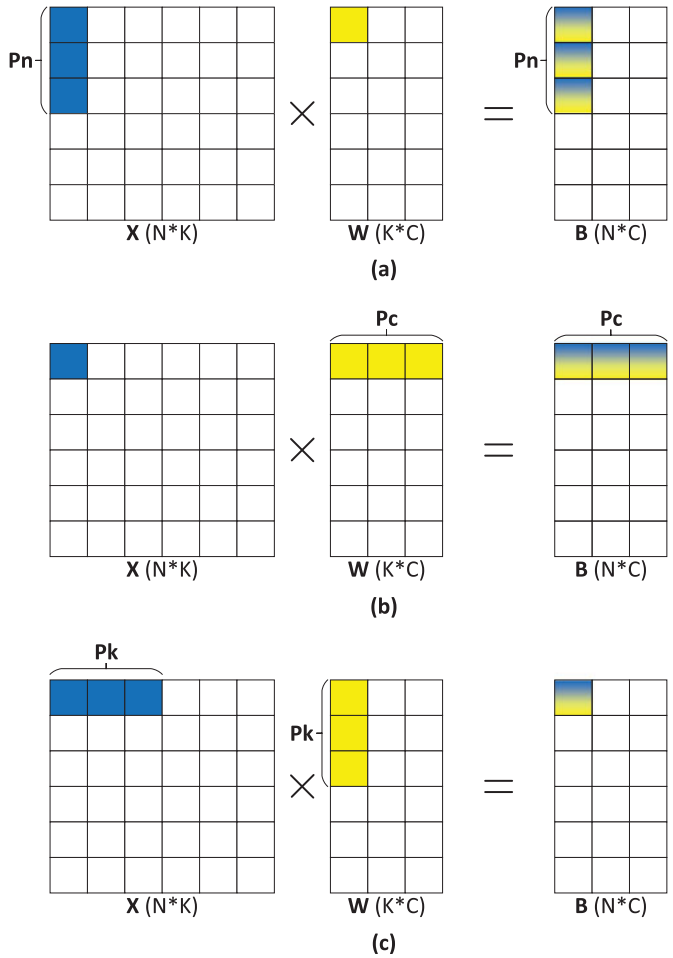

Fig. 4. Loop unrolling: (a) unroll Loop-1; (b) unroll Loop-2; (c) unroll Loop-3.

- Loop-1 unrolled (Fig. 4a): in this case, a column vector of Pn pixels from X is multiplied with a pixel from W in each cycle, and generates a column vector of Pn pixels which will be accumulated to matrix B. If data reuse in local registers is not enabled, it will involve $2 \times Pn + 1$ SRAM reads and Pn SRAM writes in each cycle.
- Loop-2 unrolled (Fig. 4b): in this case, a row vector of Pc pixels from W is multiplied with a pixel from X in each cycle, and generates a row vector of Pc pixels which will be accumulated to matrix B. If data reuse in local registers is not enabled, it will involve $2 \times Pc + 1$ SRAM reads and Pc SRAM writes in each cycle. This unrolling is also called outer-product method.
- Loop-3 unrolled (Fig. 4c): in this case, the inner product of a row vector of Pk pixels from X and a column vector of the same size from W is computed in each cycle, and generates one pixel which will be accumulated to matrix B. If data reuse in local registers is not enabled, it will involve $2 \times Pk + 1$ SRAM reads and 1 SRAM write in each cycle. This unrolling is also called inner-product method.

These unrolling factors (Pn, Pk, Pc) will determine the total number of parallel MAC operations as well as the number of required multipliers.

TABLE 2
GCN Loop Parameters and Design Variables

		SpMM1 ($B = XW$)			SpMM2 ($O = AB$)		
GCN Loops		Loop-1	Loop-2	Loop-3	Loop-4	Loop-5	Loop-6
Dimensions		N	C	K	M	C	N
Without Loop Fusion	Loop Tiling	Tn0	Tc0	Tk	Tm	Tc1	Tn1
	Loop Unrolling	Pn0	Pc0	Pk	Pm	Pc1	Pn1
With Loop Fusion	Loop Tiling	Tn0	Tc0	Tk	Tm	-(equal to Tc0)	-(equal to Tn0)
	Loop Unrolling	Pn0	Pc0	Pk	Pm	Pc1	Pn1

3.1.2 Loop Tiling

Loop tiling can be applied for each SpMM to leverage data locality, and it determines the required capacity and the partitioning of GLB. As the on-chip GLB capacity is usually not large enough to hold all the data in GCNs, loop tiling can be used to divide the entire data and only fit a small portion of the data into the on-chip GLB. By properly selecting the loop tile size, the data reuse can be maximized to reduce off-chip DRAM access. This will significantly improve the overall energy efficiency, as the energy cost of off-chip memory accesses is orders of magnitude higher than that of arithmetic operations. The tile size sets the lower bound of the required GLB capacity. In other words, the GLB should be sized large enough to hold the data tiles.

3.1.3 Loop Interchange

Loop interchange [26] determines the computation order of the loops, and it can be used to enable different types of data reuse to reduce external memory traffic by exchanging the order of the nested loops. There are two types of loop interchange in the GCN loops, namely intra-tiling and inter-tiling loop orders. Intra-tiling loop order determines the pattern of data movements from on-chip GLB to register files. Inter-tiling loop order determines the data movement from external memory to on-chip GLB. Loop interchange along with local memory promotion can reduce the data movements. Specifically, if the innermost loop is irrelevant to a matrix, i.e., the loop iterator does not appear in the access function of the matrix [27], there will be redundant memory operations between different loop iterations which can be eliminated to reduce memory access operations.

3.1.4 Loop Fusion

Loop fusion optimization [28] can be leveraged to reduce data transfer of intermediate data. Specifically, we can fuse the processing of SpMM1 and SpMM2 to reduce the data transfer of matrix B between off-chip DRAM and on-chip GLB. As shown in Fig. 2, if the two SpMMs are executed sequentially without fusion, the elements of matrix B are stored back to DRAM in SpMM1, and they are again fetched from DRAM to on-chip in SpMM2. Therefore, we can reduce the data transfer of these intermediate data by fusing the execution of SpMM1 and SpMM2. When SpMM1 finishes the computation of loop k and generates a B chunk, we can pause the execution of SpMM1 and proceed to the execution of SpMM2. By doing so, the data transfer of the intermediate matrix (B) is eliminated. Notably, although loop fusion reduces data transfer of intermediate results, it somehow sacrifices the freedom of loop interchange. Specifically, the iteration k in SpMM1 must be the innermost loop to ensure that matrix B finishes all its computations (not a PartialMat) before being forwarded to SpMM2. Moreover, as m becomes the innermost loop in the communication part of SpMM2, matrix O has to be frequently transferred between on-chip and off-chip. Since O is the result matrix, the volume of data transfer is doubled compared to the input matrix such as matrix A because the result matrix has to be written back to the main memory when being replaced, whereas the input matrix can be directly replaced without being written back.

Table 2 lists the parameters in GCNs and the design variables used by the four loop optimization techniques, where variables with a prefix of capital T denotes the tile size, and P for unrolling factors. Since both SpMM1 and SpMM2 contain the loop n and c, we hereafter use $n0, c0$ as the loop iterator in SpMM1, and $n1, c1$ as the loop iterator in SpMM2.

3.2 Flexible GCN Dataflow

Although we have concluded the key design factors of GCN dataflow, it is not easy to decide which combination of design variables is optimal for a given GCN layer. Simply using static design variables by random sampling for all layers as many prior works did [18], [20] is far from optimal due to the dimension and sparsity variance across different layers. Therefore, in this subsection, we introduce how to determine the design variables for a given graph convolutional layer. We first formulate the selection of design variables as an optimization problem, which aims at finding the best combination of design variables that maximizes the design objectives (e.g., minimizing the number of off-chip DRAM accesses and latency) under certain design constraints (e.g., on-chip storage size and the number of multipliers). We found that it is an NP-hard problem because of the large design space, thus requiring heuristic solutions in practice. Therefore, we propose a greedy search algorithm to address this problem.

3.2.1 Design Objectives

We are primarily targeting substantial improvements for the following three design objectives:

- Computation latency which depends on the loop unrolling factors and the sparsity of data.
- The number of off-chip DRAM accesses which primarily relies on the size of GLB and the degree of

data reuse. These latter are determined by the tile size, inter-tiling loop order, and loop fusion strategy.
- The number of on-chip SRAM accesses, which is determined by the loop unrolling strategies and intra-tiling loop order, since they determine the reuse patterns of the data transfer from GLB to local registers.

To simultaneously achieve all the design objectives might be infeasible as the best combination of design variables for off-chip DRAM accesses may not be optimal for on-chip SRAM accesses, and vice versa. Therefore, to optimize the overall efficiency, we combine the three design objectives into one by calculating their weighted total as follows:

$$\begin{aligned} \underset{\mathbb{X}}{Minimize}\ & J = L(\mathcal{X}^u) + \omega_1 \cdot V_d(\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f) \\ & \quad + \omega_2 \cdot V_s(\mathcal{X}^u, \mathcal{X}^{oi}) \\ s.t.\quad & 0 < T_m \le M, \quad 0 < T_k \le K \\ & 0 < T_{n0} \le N, \quad 0 < T_{n1} \le N \\ & 0 < T_{c0} \le C, \quad 0 < T_{c1} \le C \\ & S_X + S_W + S_{B1} <= GLBsize \\ & S_A + S_O + S_{B2} <= GLBsize \\ & P_{n0} \times P_{c0} \times P_k \le \#PEs \\ & P_{n1} \times P_{c1} \times P_k \le \#PEs, \end{aligned} \tag{2}$$

where $\mathbb{X} = \mathcal{X}^t \cup \mathcal{X}^{oo} \cup \mathcal{X}^f \cup \mathcal{X}^u \cup \mathcal{X}^{oi}$ denotes the entire search space, and $\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f, \mathcal{X}^u, \mathcal{X}^{oi}$ denote the parameter spaces of tile size, inter-tiling loop order, loop fusion strategy, unrolling factors, and intra-tiling loop order, respectively. L, V_d and V_s denote the computation latency, the number of off-chip DRAM accesses and on-chip SRAM accesses, respectively. $S_X, S_W, S_{B1}, S_A, S_o, S_{B2}$ denote the required on-chip storage size of the corresponding matrices, which are determined by the tile size. ω_1 and ω_2 are adjustment parameters that reflect the difference in the energy cost between basic arithmetic operation, DRAM access and SRAM access. According to [25], we set $\omega_1 = 206.5, \omega_2 = 1.6$ indicating a basic DRAM access operation and SRAM access consumes 206.5× and 1.6× more energy than a basic arithmetic operation does.

To solve this optimization problem, we first need to measure L, V_d, and V_s, given a combination of design variables and a GCN layer. In this paper, we use the analytical models in [21] to estimate these numbers. Then, we need to find out which combination of design variables can minimize the design objective described in Equation (2). However, this would take substantial time to explore all the potential design variables finding the optimal solution due to the large design space. According to our experiments, the exhaust search process takes tens of hours to fully explore the entire design space on an Intel I7-8650U@1.90GHz processor, which is infeasible for practical use. Therefore, it requires heuristic solutions in practice.

To simplify the search, We use outer-product-based computation architecture [22] as shown in Fig. 4b. Although this method would have a negative impact on the reuse of the output matrix, it provides additional input matrix reuse compared to the inner-product-based method. More importantly, it well supports the elimination of zero computations and avoids the intra-PE workload imbalance problem. We store the sparse matrix in Compressed Sparse Column (CSC) format, while the input DenseMat is stored in dense format in row-major order. Since the input pixel from X is the input operand for all the P_{c0} multiplications, these computations can be skipped simultaneously if the input pixel is zero.

TABLE 3
The Greedy Search Algorithm to Determine the Design Variables

Conditions	Loop Fusion	Inter-tiling Loop Order	Tile Size Setting Priority
$N \cdot C \ge GLB_{size}$	No	$n_0 \rightarrow c_0 \rightarrow k$, $m \rightarrow c_1 \rightarrow n_1$	①T_{n0}, Tm ②T_{c0}, T_{c1} ③T_{n1}, T_k
$N \cdot C < GLB_{size}$	Yes	$n_0 \rightarrow c_0 \rightarrow k \rightarrow m$	①T_{n0}, T_{n1} ②T_{c0}, T_{c1} ③T_m, T_k

For the design variables in loop interchange, loop tiling, and loop fusion, we provide a greedy search algorithm that can reduce the search time to several seconds. Table 3 shows how to determine the design variables for off-chip DRAM accesses ($\mathcal{X}^t, \mathcal{X}^{oo}, \mathcal{X}^f$). This greedy algorithm leverages the empirical rules concluded from many simulation results.

The tile size setting priority indicates which tile size has the priority for larger number settings. For example, if $N \cdot C \ge GLB_{size}$, T_{n0} and T_m have the highest priority for larger number settings, which means they will be set to the maximum number while satisfying other constraints. T_{c0}, T_{c1} have the second-highest priority. When T_{n0} and T_m are already set as the largest number, then T_{c0}, T_{c1} will be set as large as possible. Furthermore, we discovered that the main reason for the large search space is that $\mathcal{X}^t$ (the search space of tile size) contains every integer value between 1 to the dimension size. Therefore, pruning the search space $\mathcal{X}^t$ can significantly shrink the entire search space. Most tile size values cannot be fully divided by the dimension size. In such cases, we need to pad the data block to simplify data movements [29], which will degrade the GLB space utilization. Clearly, the tile size that causes less padding will utilize the GLB space efficiently thus reducing unnecessary off-chip DRAM accesses. Among a set of tile size that results in the same number of iterations, the one requires minimum padding has the fewest data padding. Therefore, from 1 to dimension size, we only need to consider the smallest tile size values that yield a new number of loop iterations. For example, assume a value of 10 for dimension N, the candidate tile size will be {1,2,3,4,5,10} since they yield the number of loop iterations of {10,5,4,3,2,1}. The number of points to sweep in dimension N will be reduced from $O(N)$ to $O(2\sqrt{N})$. To better understand the greedy search, Table 4 presents the tile size, loop order and loop fusion choices for five datasets: Cora, CiteSeer, PubMed, Nell and Reddit [30] when we constrain the GLBsize at 128 KB.

4 SGCNAX Architecture

4.1 Overview

We propose an accelerator architecture, called SGCNAX, to support the flexible dataflow, which is depicted in Fig. 5. It consists of a host CPU, an off-chip DRAM, and an

TABLE 4
Design Variables (Tile Size, Inter-Tiling Loop Order and Loop Fusion) Derived From Greedy Search

Dataset	Layer	(M-N-K-C)	γ_A	γ_X	Loop fusion	Inter-tiling Loop order	Tile Size Tuple $(T_{n0}, T_{c0}, T_k, T_{n1}, T_{c1}, T_m)$
Cora	L1	(2708-2708-1433-16)	0.0018	0.0127	Yes	$n_0 \rightarrow c_0 \rightarrow k \rightarrow m$	(2708,16,1,2708,16,1)
	L2	(2708-2708-16-7)	0.0018	0.78	Yes	$n_0 \rightarrow c_0 \rightarrow k \rightarrow m$	(2708,7,1,2708,7,1)
Citeseer	L1	(3327-3327-3703-16)	0.0011	0.0085	Yes	$n_0 \rightarrow c_0 \rightarrow k \rightarrow m$	(3000,16,5,3000,16,1)
	L2	(3327-3327-16-6)	0.0011	0.0085	Yes	$n_0 \rightarrow c_0 \rightarrow k \rightarrow m$	(3000,6,1,3000,6,1)
Pubmed	L1	(19717-19717-500-16)	0.00028	0.1	No	$n_0 \rightarrow c_0 \rightarrow k, m \rightarrow c_1 \rightarrow n_1$	(3073,16,1,1,16,3073)
	L2	(19717-19717-16-3)	0.00028	0.776	No	$n_0 \rightarrow c_0 \rightarrow k, m \rightarrow c_1 \rightarrow n_1$	(3000,3,1,1025,3,3000)
Nell	L1	(65755-65755-61278-64)	0.000073	0.00011	No	$n_0 \rightarrow c_0 \rightarrow k, m \rightarrow c_1 \rightarrow n_1$	(4096,1,33,1,1,4096)
	L2	(65755-65755-64-186)	0.000073	0.864	No	$n_0 \rightarrow c_0 \rightarrow k, m \rightarrow c_1 \rightarrow n_1$	(257,186,1,1,17,2817)
Reddit	L1	(232965-232965-602-64)	0.0021	0.516	No	$n_0 \rightarrow c_0 \rightarrow k, m \rightarrow c_1 \rightarrow n_1$	(641,64,1,1,9,4096)
	L2	(232965-232965-64-41)	0.0021	0.6	No	$n_0 \rightarrow c_0 \rightarrow k, m \rightarrow c_1 \rightarrow n_1$	(1153,41,1,1,17,2817)

accelerator chip. The accelerator consists of a Control Unit (CU), a global buffer (GLB), a Data Dispatcher, a PE array, a Multi-stage Permutation Network, and an Accumulator Buffer. The host CPU is used to configure the tile size, loop order, and loop fusion strategy for different datasets, and generates corresponding commands to the Control Unit. The GLB is used to exploit input data reuse and hide DRAM access latency, or for the storage of intermediate data. The Data Dispatcher distributes the input data to the PE array. The Permutation Network collects the outputs from the PE array and sends them to the Accumulator Buffer for accumulation. The Data Dispatcher and Permutation Network work collaboratively to mitigate inter-PE workload imbalance (will be discussed in Section 4.4).

4.2 PE Architecture

Fig. 5 also shows the PE architecture. It consists of a SparseMat Buffer (SMB), Input/Output DenseMat Buffers (IDMB/ODMB), a Look-Ahead FIFO, a DenseRow Prefetcher (DRP), a MAC Array, and a PE Control Unit. To process the SpMM, a portion of the sparse matrix is fetched from DRAM into SMB in CSC format, and a portion of the input/output matrix is fetched into IDMB and ODMB in dense format. First, an old PartialMat row is fetched from ODMB to the MAC array waiting for accumulation. Meanwhile, an element value from SMB is sent to the FIFO, while its row index in the CSC format is sent to DRP. Then, the DRP fetches the corresponding input DenseMat row from IDMB according to the received index and the row index of the PartialMat row. Since the required input DenseMat row is not known until receiving the index of the SparseMat element, there is a latency between the arriving time of the SparseMat element and the input DenseMat row to the MAC Array. The look-ahead FIFO is used to hide this latency. Instead of directly sending the SparseMat element to MAC Array, it is sent to the FIFO. At the same time, the DRP calculates the required row index and prefetches rows to MAC Array. The MAC Array will then conduct the outer product between the SparseMat element and the DenseMat row and the generated row will be accumulated with the Old PartialMat row.

When possible, the PartialMat row is held consistently in the MAC Array until its related computations are finished. Upon completion of the current PartialMat row, the generated new PartialMat row is then flushed to ODMB, and it proceeds to the next PartialMat row according to the execution order defined by the dataflow. When the output DenseMat can serve as the input DenseMat for the following SpMM, which is the case when we enable loop fusion, the IDMB and ODMB are logically swapped to the two SpMMs' computation sequence.

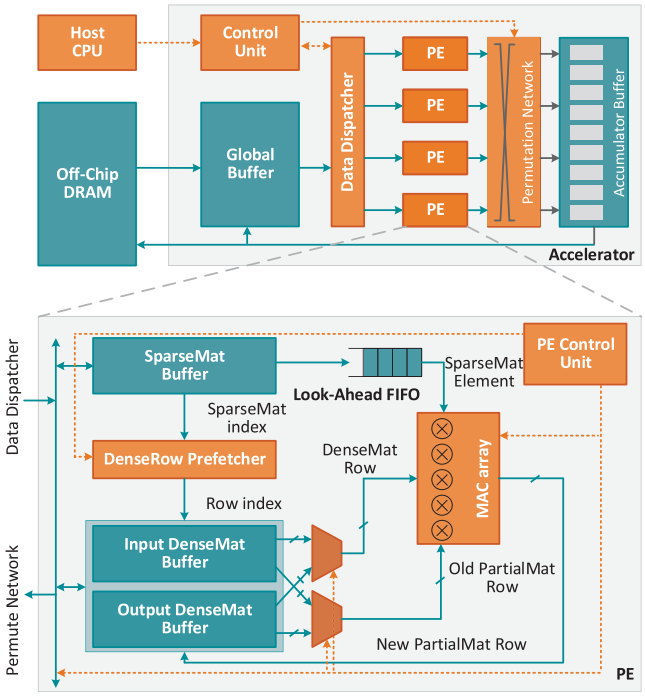

Fig. 5. The proposed SGCNAX architecture for scalable GCNs.

4.3 Spatial Tiling Strategy

As we scaled the accelerator with multiple PEs based on the GCNAX architecture [21], we will need a spatial tiling strategy to spread the work across the PE array so that each PE can operate independently. There are many spatial tiling strategies, such as partitioning the adjacency matrix (A), feature matrix (X), or weight matrix (W) along the row and/or column dimension into smaller tiles and distributing them to the PEs. Different spatial tiling strategies lead to different computation and communication patterns. For example, For example, as shown in Fig. 2, if we partition matrix X along dimension n in SpMM1 into smaller tiles, the PEs will

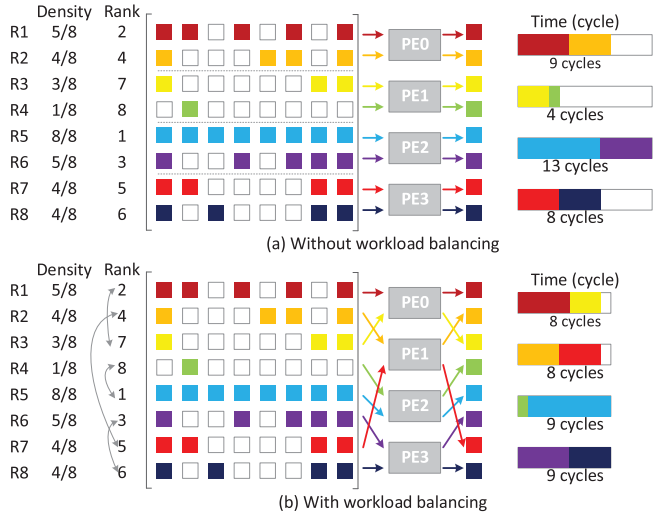

Fig. 6. An example of group-and-shuffle workload balancing mechanism.

share the matrix W and the tiles of X will be distributed to different PEs. By contrast, if we partition matrix W along dimension c, the PEs will share the matrix X and the tiles of W will be distributed to the PEs. For GCNs, considering that the dimension k and c are usually small, partitioning along these two dimensions will cause under-utilization of PEs because we may not have enough useful work to fully populate the PEs. Hence, we choose to partition the matrix X and A along the dimension n. Consequently, matrix W and matrix B are broadcast to the PEs, and each PE operates on its own subset of the computation task.

4.4 Inter-PE Workload Balancing

Since the distribution of the non-zeros in the adjacency matrix and feature matrix is extremely unbalanced as the majority of non-zeros are in only a small set of rows/columns, it can lead to severe workload imbalance across the PEs and consequently a severe performance degradation. To this end, we employ a group-and-shuffle approach to mitigate this inter-PE workload imbalance, which groups the rows (or columns depending on the spatial strategy, we assume rows hereafter for simplicity) in a sparse matrix by density so that the row groups are similar in density, and then map the rows within a group to the PEs so that each PE will complete its workload almost simultaneously.

Fig. 6 shows a simple example to better illustrate the concept of group-and-shuffle. 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 is also shown in the figure. When there is no workload balancing as shown in Fig. 6a, 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. 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). Then each row group will be mapped to a PE so that each PE will complete the task almost simultaneously. Since the grouping shuffles the output positions in the output matrix, we need to "unshuffle" the outputs to recover the correct positions. In SGCNAX, the Data Dispatcher performs the sorting and grouping of the rows, and it sends the "shuffle" information to the Control Unit. The Control Unit configures Permutation Network to "unshuffle" the outputs to the Accumulator Buffer.

Researchers have also proposed various techniques to address the load imbalance problem in sparse neural network accelerators. For example, Procrustes [31] addresses the load imbalance across a 2D PE array by distributing the non-sparse minibatch dimension across one hardware dimension and the sparse tensor dimension across the other hardware dimension. However, it's dedicated to DNNs and might need some augmentation to work with GCNs, which can be our future work.

TABLE 5
SGCNAX Configurations

PE parameters	Value		GCNAX parameters	Value
Multiplier width	64 bits		# PEs	8
MAC Array	1×16		# Multipliers	128
Look-ahead FIFO	16 entries		Global buffer	1 MB
SMB	320 KB		Accumulator Buffer Banks	16
IDMB	4 KB		Accumulator Buffer Entries	512
ODMB	256 KB		DRAM	HBM@128GB/s
			Permutation Network	16*16 Omega Network

5 Experimental Methodology

Hardware Simulator. To evaluate the performance of our design, we built a cycle-level simulator in C++ to model the behavior of the hardware. The simulator models the micro-architectural behaviors of each module, and supports the dataflows with reconfigurable design variables. The simulator counts the exact amount of DRAM reads and writes, which is used to estimate the DRAM access energy consumption according to [25].

SGCNAX Configurations. Table 5 lists the major configurations of the SGCNAX that we explore. SGCNAX is equipped with eight PEs, each with a 1×16 MAC Array using double-precision floating-point multipliers. The SMB/IDMB/ODMB in each PE is sized so that the tiles of the matrix can fit into these local buffers. The accumulator buffer has 16 banks, each with 512 entries. The global buffer size is 1 MB.

ASIC Synthesis. To measure the area and power consumption, we model all the PE logic including the MAC Array, FIFOs, DRP, and DRAM. We use the Synopsys Design Compiler with the TSMC $40nm$ library for the synthesis, and estimate the power using Synopsys PrimeTime PX. We set the clock frequency at 1 GHz. We use Cacti [32] to estimate the area, power, and access latency of the on-chip buffers and FIFOs.

Baselines. We compare SGCNAX with three GCN accelerators (HyGCN, AWB-GCN, and GCNAX), and an SpMM

TABLE 6
Hardware Characteristics of SGCNAX

PE Component	Area (mm^2)		GCNAX Component	Area (mm^2)
MAC Array	0.46 (7.1%)		8 PEs	52.08
SMB	2.60 (39.9%)		Global Buffer	8.42
IDMB	0.10 (1.5%)		Accumulator Buffer	0.31
ODMB	2.70 (41.5%)		Control Unit	0.28
DRP	0.41 (6.3%)		Data Dispatcher	1.6
PE Control Unit	0.24 (3.7%)		Permutation Network	2.3
Total	6.51 (100%)			

accelerator (SpArch). Table 7 summarizes the characteristics of these baselines and SGCNAX.

The baseline accelerators are scaled to be equipped with the same number of multipliers and DRAM bandwidth as GCNAX. Since HyGCN and AWB-GCN use single-precision floating-point numbers (32-bit) whereas SpArch uses double-precision (64-bit), we uniformly use double-precision for all accelerators to provide a fair comparison. As HyGCN uses a tandem-engine architecture consisting of SIMD cores for the aggregation phase and systolic modules for the combination phase, the multipliers are divided into two groups in a ratio of 1:8 for the two engines according to its original configuration. We also resized the baseline accelerators to be equipped with the on-chip storage capacity. The DRAM bandwidth for all the accelerators is scaled to 128 GB/s. Note that as HyGCN uses edge-centric programming model for the aggregation phase, their computation in the aggregation phase is not matrix multiplication. Our simulator takes this into account and estimates the execution cycles and DRAM accesses according to HyGCN's dataflow. Although SpArch is not customized for GCNs, it is still selected as a baseline since it supports the key computations in GCNs. As SpArch does not mention how to support chain SpMM, we assume that it processes the chain SpMM sequentially without loop fusion. Hereafter we denote SpArchG as our simulated accelerator that uses SpArch to process chain SpMM.

6 Experimental Results

6.1 Area and Power

We obtain the area and power consumption of the SGCNAX PE under TSMC 40 nm technology. Table 6 summarizes the area and power of the major components in SGCNAX and its PE. A significant fraction of the PE area is contributed by memories (SMB, IDMB, and ODMB), which consume 82.9% of the total area, while the MAC array only consumes 7.1%. IDMB and ODMB are heavily banked for parallelization so they consume more area than SMB. The Data Dispatcher and Permutation Network only consumes 6.0% of the total area, which shows that the workload balancing module incurs a negligible area overhead in exchange for higher resource utilization.

6.2 Performance

Fig. 7 compares the performance of SGCNAX and the baselines measured by the total number of execution cycles. On average, SGCNAX is 9.2×, 11.5×, 1.6×, and 1.2× faster than HyGCN, SpArchG, AWB-GCN and GCNAX, respectively. SGCNAX outperforms the baselines on all five datasets. The reasons for the high performance of SGCNAX are threefold. First, the execution order of the chain-SpMM of SGCNAX

TABLE 7
Characteristics of the Accelerators

Accelerator	Execution order	Compute engine	Loop fusion	Loop order	Tile size	Inner Spatial Dataflow
HyGCN	$(AX)W^{\dagger}$	Tandem engine	Yes	Static	Static	Inner product
AWB-GCN	$A(XW)$	Uniform engine	Yes	Static	Static	Inner product
SpArchG [§]	$(AX)W$	Uniform engine	No	Static	Static	Outer product
GCNAX	$A(XW)$	Uniform engine	Adaptive	Adaptive	Static	Outer product
SGCNAX	$A(XW)$	Uniform engine	Adaptive	Adaptive	Adaptive	Outer product

[†]*HyGCN uses edge-centric programming model for the aggregation phase, so their computation in the aggregation phase is not matrix multiplication. Nevertheless, the result of the aggregation phase is a large matrix that is used as the input to perform matrix multiplication in the combination phase.*
[§]*SpArchG uses SpArch [33] (an SpGEMM accelerator) to process matrix multiplications in GCNs.*

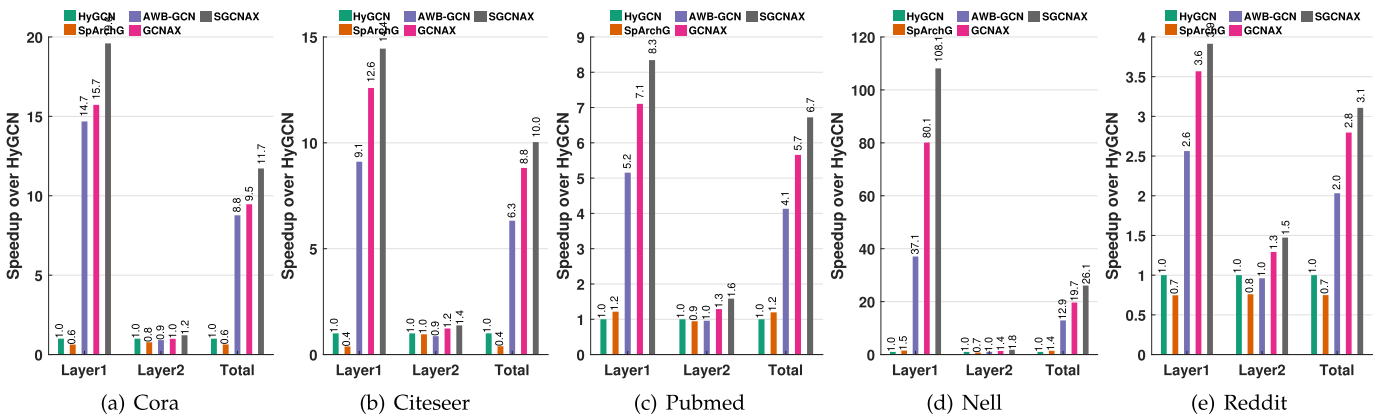

Fig. 7. Speedup of SGCNAX and the baseline accelerators over HyGCN.

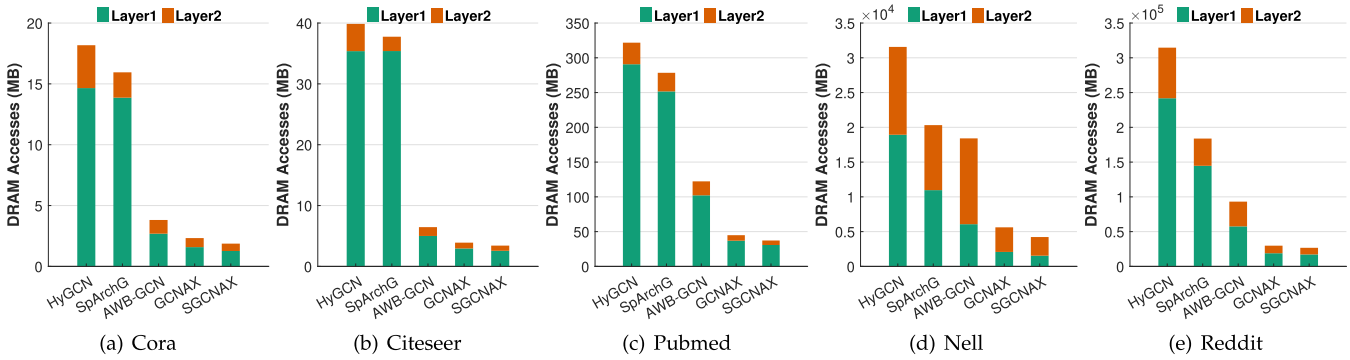

Fig. 8. Number of DRAM accesses of SGCNAX and the baseline accelerators.

reduces the number of operations compared to that of HyGCN. Second, SGCNAX uses a uniform-engine architecture that can avoid the workload imbalance incurred by tandem-engine architectures. HyGCN can only achieve optimal performance by carefully orchestrating the computational capacity of the combination and aggregation engines for a given dataset, but it inevitably incurs performance loss when accommodating different datasets with different computational requirements for the aggregation and combination engines. Finally, SGCNAX achieves the lowest DRAM accesses by adaptively configuring the dataflow for different datasets, which also explains why SGCNAX outperforms AWB-GCN and SpArch. The number of DRAM accesses has a strong impact on performance since it might be the system bottleneck. AWB-GCN uses the inner-product-based method for SpMM which incurs workload imbalance between PEs thereby degrading the performance. AWB-GCN addresses this inefficiency by a software scheduler and additional hardware modules that increase hardware complexity and introduce extra overhead. Since SpArchG is customized for sparse-sparse matrix multiplication, it achieves high performance when performing AX. However, the performance gain of SpArchG is hindered because 1) its processing order results in larger computation volume; 2) SpArchG is not good at processing dense-dense matrix multiplication. Furthermore, by alleviating inter-PE workload imbalance and using adaptive tile size, SGCNAX performs 1.2× better than GCNAX on average.

As for the speedup for specific datasets, SGCNAX performs 3.1–26.1× better over HyGCN, 4.1–25.6× better over SpArchG, 1.3–2.0× better over AWB-GCN, and 1.1–1.3× better over GCNAX. The performance gain on the Reddit dataset is not so significant because the execution order reduces computations by only 2.9× which is not that much compared to other datasets. Besides, the density of feature vectors in Reddit (larger than 50%) is higher than that of other datasets, which hinders the performance gains of SGCNAX because it still performs SpMM even though the input matrix is not that sparse.

6.3 DRAM Accesses

Fig. 8 shows the number of DRAM accesses of the five accelerators. Overall, SGCNAX achieves on average 9.7×, 7.5×, 2.9× and 1.2× reduction on DRAM accesses over HyGCN, SpArchG, AWB-GCN, and GCNAX, respectively. This benefits from the optimal tile size tuple, the data reuse optimization, and the adaptive loop fusion strategy. The DRAM access reduction varies across the datasets. Specifically, SGCNAX reduces DRAM accesses by a factor of 7.5–11.8× over HyGCN, 4.8–11.0× over SpArchG, 1.9–4.4× over AWB-GCN, and 1.1–1.4× over GCNAX. Since HyGCN and SpArchG use an inefficient execution order, they involve more computations that result in more DRAM accesses. AWB-GCN optimizes the reuse of the intermediate matrix. However, it sacrifices the reuse of the output matrix due to the limited on-chip storage size. Moreover, the tile sizes are not carefully tailored in the AWB-GCN accelerator. SGCNAX uses adaptive tile size thus saving DRAM accesses compare to GCNAX that uses static tile size.

6.4 Energy Consumption

Fig. 9 shows the normalized energy consumption of the four accelerators. Overall, SGCNAX achieves 12.3×, 9.9×, 3.0× and 1.3× energy savings compared to HyGCN, SpArchG, AWB-GCN, and GCNAX, respectively. This is because our proposed accelerator has fewer DRAM accesses and better utilization of the computing resources.

Energy-Delay Product. Energy-delay product is used to verify that a dataflow does not achieve high energy efficiency by sacrificing processing parallelism. Fig. 10 shows the normalized EDP of the four accelerators. The delay is calculated as the reciprocal of the number of execution cycles. Compared with the baseline accelerators, SGCNAX is 152.9×, 159.1×, 4.8×, and 1.5× better in EDP averaged on the five datasets.

6.5 PE Granularity

As described in Section 4, two factors may lead to performance degradation of SGCNAX. One is the workload

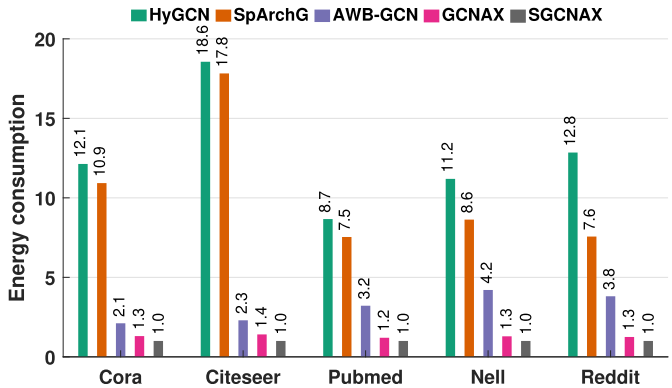

Fig. 9. Energy consumption of SGCNAX and the baselines (the energy consumption of SGCNAX is normalized to 1).

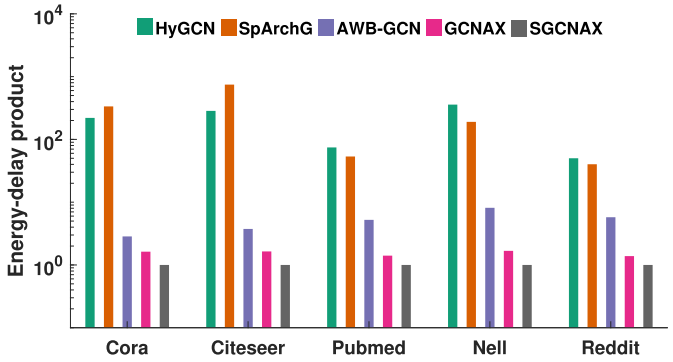

Fig. 10. Energy delay product (EDP) of SGCNAX and the baselines accelerators (the EDP of SGCNAX is normalized to 1).

imbalance across the PEs, the other is intra-PE MAC array fragmentation when we don't have enough work to fully populate the MAC array in each PE. we conduct a sensitivity analysis to quantify the effects of both factors on the accelerator performance. Assuming a fixed 128 multipliers for the accelerator, we sweep the total number of PEs from 16 (16 PEs, 8 MAC units per PE) down to 4 (4 PEs, 32 MAC units per PE). Clearly, SGCNAX with 4 PEs is more likely to suffer from intra-PE fragmentation because we may not have a large enough working set to fully populate the large MAC array. However, SGCNAX with 16 PEs is more likely to suffer inter-PE workload imbalance and requires a more complicated data dispatcher and permutation network. We found that SGCNAX with 8 PEs strikes a balance between the two factors.

7 Related Work

Graph Neural Network (GNN) Accelerators. Besides the GCN accelerators mentioned in Section 2, there are also a few other GNN accelerators in the literature. Auten *et al.* [34] proposed a GNN accelerator to efficiently execute the irregular data movement required for graph computation in GNNs, while also providing a high compute throughput required by GNN models. EnGN [35] is designed to accelerate the three key stages of GNN propagation, which is abstracted as common computing patterns shared by typical GNNs. GRIP [36] is designed for low-latency inference of GNNs, which splits GNN inference into a fixed set of edge- and vertex-centric execution phases that can be implemented in hardware, and then specialize each unit for the unique computational structure found in each phase. GraphACT [37] is dedicated to the acceleration of training GCNs on CPU-FPGA heterogeneous systems, which incorporates multiple algorithm-architecture co-optimizations. VersaGNN [38] is a systolic-array-based versatile GNN accelerator that unifies dense and sparse matrix multiplication in GNNs.

Graph Analytics Accelerators. With the emergence of applications on graph analytics, many accelerators are proposed to efficiently support these workloads [39], [40], [41], [42], [43], [44], [45], [46], [47], [48], [49], [50], [51], [52], [53]. Hong *et al.* [39], [54] propose a warp centric execution model for graph applications. Ozdal *et al.* [41] propose a configurable architecture template that is specifically optimized for iterative vertex-centric graph applications with irregular access patterns and asymmetric convergence. Graphicionado [44] exploits not only data structure-centric datapath specialization, but also memory subsystem specialization for efficient graph analytics processing. Medusa [45] is a parallel graph processing system on GPUs that enables developers to leverage the massive parallelism and other hardware features. GraphR [46] is a ReRAM-based graph processing accelerator that leverages the near-data processing and explores the opportunity of performing massive parallel analog operations with low hardware and energy cost. GraphABCD [47] is an asynchronous heterogeneous graph analytic framework that offers algorithm and architectural supports for asynchronous execution, without undermining its fast convergence properties. Yan *et al.* [48] propose a hardware/software co-design with decoupled datapath and data-aware dynamic scheduling to alleviate irregularity in graph analytics accelerators. Although these accelerators deliver considerable performance and energy efficiency improvement, they are inefficient when handling GCNs because even though they are designed to alleviate the irregularity of graph data, they do not leverage the regularity in GCNs.

Neural Network Accelerators. There have been many works devoted to accelerating neural networks [55], [56], [57], [58], [59], [60], [61]. For dense neural networks, the accelerators mainly focus on leveraging the massive parallelism to improve performance and utilization, such as TPU [55] and Eyeriss [56]. Due to the intrinsic sparsity structure, many accelerator [57], [58], [59], [60] have been proposed to reduce operations from sparsity. However, GCNs contain two-phase matrix multiplications that enable new kinds of parallelisms and data reuse patterns that are not exploited in these neural network accelerators. Although we can extend CNN accelerators to run SpMMs by equalizing the input and filter dimensions, it weakens the advantages of CNN accelerators since they are specialized for convolutions rather than matrix multiplications.

8 Conclusion

In this paper, we propose a scalable accelerator architecture for GCNs called SGCNAX. The salient feature of the proposed architecture is that the dataflow can reconfigure the loop optimization variables to adapt to different GCN configurations, which simultaneously improves resource utilization and reduces data movement. The SGCNAX accelerator tailors the compute engine, buffer structure and size to support the optimized dataflow. Furthermore SGCNAX is capable to mitigate workload imbalances through hardware/software co-design approaches namely the use of an outer-product-based computation architecture for SpMM computation, and a group-and-shuffle computing approach for concurrent PEs computation and simultaneous completion. We evaluated our proposed architecture on five real-world graph datasets. The simulation results show that SGCNAX performs 9.2×, 1.6× and 1.2× better, and reduces DRAM accesses by a factor of 9.7×, 2.9×, and 1.2× compared to HyGCN, AWB-GCN, and GCNAX, respectively.

References

[1] X. Wang *et al.*, "Traffic flow prediction via spatial temporal graph neural network," in *Proc. The Web Conf.*, 2020, pp. 1082–1092.

[2] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey," 2021, *arXiv:2101.11174*.

[3] W. Shi and R. Rajkumar, "Point-GNN: Graph neural network for 3D object detection in a point cloud," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 1711–1719.

[4] Y. Shen *et al.*, "Person re-identification with deep similarity-guided graph neural network," in *Proc. Eur. Conf. Comput. Vis.*, 2018, pp. 486–504.

[5] P.-E. Sarlin, D. DeTone, T. Malisiewicz, and A. Rabinovich, "SuperGlue: Learning feature matching with graph neural networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2020, pp. 4938–4947.

[6] A. Luo *et al.*, "Cascade graph neural networks for RGB-D salient object detection," in *Proc. Eur. Conf. Comput. Vis.*, 2020, pp. 346–364.

[7] C.-Y. Wee *et al.*, "Cortical graph neural network for ad and MCI diagnosis and transfer learning across populations," *NeuroImage: Clin.*, vol. 23, 2019, Art. no. 101929.

[8] T.-A. Song *et al.*, "Graph convolutional neural networks for alzheimer's disease classification," in *Proc. IEEE 16th Int. Symp. Biomed. Imag.*, 2019, pp. 414–417.

[9] S.-H. Wang *et al.*, "Covid-19 classification by FGCNet with deep feature fusion from graph convolutional network and convolutional neural network," *Inf. Fusion*, vol. 67, pp. 208–229, 2021.

[10] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 1, pp. 249–270, Jan. 2022.

[11] M. Henaff, J. Bruna, and Y. LeCun, "Deep convolutional networks on graph-structured data," 2015, *arXiv:1506.05163*.

[12] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proc. 30th Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 3844–3852.

[13] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*.

[14] H. Yang, "AliGraph: A comprehensive graph neural network platform," in *Proc. 25th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2019, pp. 3165–3166.

[15] A. Lerer *et al.*, "PyTorch-BigGraph: A large-scale graph embedding system," 2019, *arXiv:1903.12287*.

[16] M. Yan *et al.*, "Characterizing and understanding GCNs on GPU," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 22–25, Jan.–Jun. 2020.

[17] Z. Zhang, J. Leng, L. Ma, Y. Miao, C. Li, and M. Guo, "Architectural implications of graph neural networks," *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 59–62, Jan.–Jun. 2020.

[18] T. Geng *et al.*, "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 922–936.

[19] Y. Ma *et al.*, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 45–54.

[20] M. Yan *et al.*, "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 15–29.

[21] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit.*, 2021, pp. 775–788.

[22] S. Pal *et al.*, "OuterSPACE: An outer product based sparse matrix multiplication accelerator," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 724–736.

[23] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1024–1034.

[24] K. Xu *et al.*, "How powerful are graph neural networks?," 2018, *arXiv:1810.00826*.

[25] M. Horowitz, "Energy table for 45nm process," Stanford VLSI wiki, 2012. [Online]. Available: http://vlsiweb.stanford.edu/

[26] J. R. Allen and K. Kennedy, "Automatic loop interchange," in *Proc. SIGPLAN Symp. Compiler Construction*, 1984, pp. 233–246.

[27] C. Zhang *et al.*, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, 2015, pp. 161–170.

[28] W. Pugh, "Uniform techniques for loop optimization," in *Proc. 5th Int. Conf. Supercomputing*, 1991, pp. 341–352.

[29] Q. Nie, "Memory-driven data-flow optimization for neural processing accelerators," Ph.D Dissertation, Elect. Eng. Dept., Princeton Univ., Princeton, NJ, USA, 2020. [Online]. Available: https://dataspace.princeton.edu/handle/88435/dsp01cf95jf42w

[30] Z. Wu *et al.*, "A comprehensive survey on graph neural networks," 2019, *arXiv:1901.00596*.

[31] D. Yang, A. Ghasemazar, X. Ren, M. Golub, G. Lemieux, and M. Lis, "Procrustes: A dataflow and accelerator for sparse deep neural network training," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2020, pp. 711–724.

[32] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to understand large caches," Univ. Utah Hewlett Packard Lab., Tech. Rep., 2009. [Online]. Available: https://www.cs.utah.edu/~rajeev/cacti6/cacti6-tr.pdf

[33] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "SpArch: Efficient architecture for sparse matrix multiplication," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 261–274.

[34] A. Auten, M. Tomei, and R. Kumar, "Hardware acceleration of graph neural networks," in *Proc. 57th ACM/IEEE Des. Autom. Conf.*, 2020, pp. 1–6.

[35] S. Liang *et al.*, "EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1511–1525, Sep. 2021.

[36] K. Kiningham, C. Re, and P. Levis, "GRIP: A graph neural network accelerator architecture," 2020, *arXiv:2007.13828*.

[37] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2020, pp. 255–265.

[38] F. Shi, A. Y. Jin, and S.-C. Zhu, "VersaGNN: A versatile accelerator for graph neural networks," 2021, *arXiv:2105.01280*.

[39] S. Hong *et al.*, "Accelerating CUDA graph algorithms at maximum warp," *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[40] V. Balaji and B. Lucia, "Combining data duplication and graph reordering to accelerate parallel graph processing," in *Proc. 28th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2019, pp. 133–144.

[41] M. M. Ozdal *et al.*, "Energy efficient architecture for graph analytics accelerators," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 166–177, 2016.

[42] G. Dai *et al.*, "FPGP: Graph processing framework on FPGA a case study of breadth-first search," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 105–110.

[43] T. Oguntebi and K. Olukotun, "GraphOps: A dataflow library for graph analytics acceleration," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 111–117.

[44] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–13.

[45] J. Zhong and B. He, "Medusa: A parallel graph processing system on graphics processors," *ACM SIGMOD Rec.*, vol. 43, no. 2, pp. 35–40, 2014.

[46] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR: Accelerating graph processing using ReRAM," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 531–543.

[47] Y. Yang *et al.*, "GraphABCD: Scaling out graph analytics with asynchronous block coordinate descent," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 419–432.

[48] M. Yan *et al.*, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 615–628.

[49] A. Abdolrashidi, D. Tripathy, M. E. Belviranli, L. N. Bhuyan, and D. Wong, "WIREFRAME: Supporting data-dependent parallelism through dependency graph execution in GPUs," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 600–611.

[50] X. Ma, D. Zhang, and D. Chiou, "FPGA-accelerated transactional execution of graph workloads," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 227–236.

[51] F. Sadi *et al.*, "PageRank acceleration for large graphs with scalable hardware and two-step SpMV," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2018, pp. 1–7.

[52] Y. Wang, J. C. Hoe, and E. Nurvitadhi, "Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform," in *Proc. IEEE 27th Annu. Int. Symp. Field-Programmable Custom Comput. Machines*, 2019, pp. 136–144.

[53] Y. Zhuo *et al.*, "GraphQ: Scalable PIM-based graph processing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 712–725.

[54] M. Zhang *et al.*, "GraphP: Reducing communication for PIM-based graph processing with efficient data partition," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 544–557.

[55] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
[56] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit.*, 2016, pp. 367–379.
[57] J. Albericio *et al.*, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Comput. Archit. News*, vol. 44, pp. 1–13, 2016.
[58] S. Zhang *et al.*, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2016, pp. 1–12.
[59] A. Parashar *et al.*, "SCNN: An accelerator for compressed-sparse convolutional neural networks," in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 27–40.
[60] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 151–165.
[61] S. Han *et al.*, "EIE: Efficient inference engine on compressed deep neural network," in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 243–254.

Jiajun Li (Member, IEEE) received the BE degree from the Department of Automation, Tsinghua University, China, in 2013,and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, China, in 2019. From 2019 to 2021, he was a postdoctoral researcher with the Department of Electrical and Computer Engineering, George Washington University, Washington, DC. He is currently an associate professor with the School of Astronautics, Beihang University, China. His current research interests include machine learning and heterogeneous computer architecture.

Hao Zheng (Member, IEEE) received the BS degree in electrical engineering from Beijing Jiaotong University, Beijing, China, and the PhD degree in computer engineering from George Washington University, Washington, DC. He is currently an assistant professor of electrical and computer engineering at the University of Central Florida, Orlando, Florida. His research interests include computer architecture and parallel computing, with emphasis on interconnection networks, machine learning techniques for efficient computing, and energy-efficient manycore architecture designs.

Ke Wang (Member, IEEE) received the BS degree in electrical engineering from Peking University, China, in 2013, and the MS degree in electrical engineering from Worcester Polytechnic Institute, Worcester, Massachusetts, in 2015. He is currently working toward the PhD degree in computer engineering in the School of Engineering and Applied Science, George Washington University, Washington, DC. His research interests include optimized NoC design of high performance, power efficiency and reliability using 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 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 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. He is currently the editor-in-chief of the *IEEE Transactions on Computers*. For more information, please visit https://hpcat.seas.gwu.edu/Director.html.