# SAGA: Sparsity-Agnostic Graph Convolutional Network Acceleration with Near-optimal Workload Balance

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

***Abstract*—Graph Convolutional Networks (GCNs) have shown much promise in resolving sophisticated scientific problems with non-Euclidean data, such as traffic prediction, disease classification, and many others. However, the irregular sparsity of real-world graphs remains a major challenge toward efficient GCN acceleration. In this paper, we propose SAGA, a Sparsity-Agnostic Graph Convolutional Accelerator with near-optimal workload balance.**

**Specifically, it consists of two unique features, an NZ-based scheduling, and a novel accelerator architecture. Unlike conventional GCN accelerators with uneven distribution of sparse matrix, the proposed NZ-based scheduling leverages the metadata encoded in the compression format to enable even distribution of sparse matrix at runtime, thus achieving near-optimal workload balancing. In addition, the proposed architecture, including a task scheduler, an accumulation table, and a partial row accumulation unit, can support the proposed NZ-based scheduling without data preprocessing and reformatting with low overheads. We prototyped the proposed design through FPGAs, and our evaluation results show that SAGA achieves up to 1.56× speedup and 2.05× energy savings on average as compared to the prior art [1].**

## I. Introduction

Graph Convolutional Networks (GCNs) have been widely applied to solve many contemporary data-intensive scientific problems, such as traffic prediction [2], object detection [3], disease classification [4], and many others. However, it remains a challenge to efficiently process GCN applications involving substantial compute and memory-intensive operations. Such unique GCN characteristics pose new computation and communication requirements on the underlying hardware design. Significant research efforts [1], [5]–[9] have been devoted to optimizing GCN computations, facilitating their primitive operations - Sparse-Dense Matrix Multiplication (SpMM). Even though SpMM acceleration has been well studied [10], [11], GCNs operate on real-world graphs whose adjacency matrix exhibits a large variation in the distribution of non-zero elements. Such irregular sparsity consequently leads to imbalanced workload distribution, and thus it becomes a major obstacle toward parallel GCN acceleration, especially facing large-scale graph datasets.

Despite many efforts, the crux of the workload imbalance has not been fully addressed in existing solutions devised for GCN accelerations. For example, AWB-GCN [1], like other SpMM accelerators [12], employs row-wise distribution of sparse matrices, where the sparsity varies significantly from row to row. The sparsity variation consequently leads to workload imbalance and resource underutilization. AWB-GCN, therefore, proposed a set of runtime workload distribution techniques that can redirect computations to underutilized computation resources. However, these techniques require complex all-to-all switching circuitry, large task-distribution queues, and sophisticated workload monitoring logic, which increases the accelerator complexity and further limits its scalability.

On the other hand, non-zero (NZ) based scheduling [13], [14] have been exploited to remedy the workload imbalance via even distribution of non-zero elements. However, conventional NZ-based scheduling approaches often require data preprocessing [13] or data reformatting [14]. The direct applications of such an idea in GCN accelerators are costly, as it is impractical to preprocess or reformat constantly evolving real-world graphs with millions of parameters.

In this paper, we aim to explore an efficient application of NZ-based scheduling for GCN accelerators without needing data preprocessing and reformatting. This requires NZ-based scheduling to be compatible with existing sparse formats, such as the compressed sparse row (CSR) format. However, implementing such an idea in GCN accelerations is very challenging. The NZ-based scheduling could jeopardize the intactness of the sparse matrix and distribute the non-zero elements across different compute units of the accelerator. These scattered workloads generate partial sums that must be accumulated together. However, accumulating partial sums over physically-distanced workloads incurs significant latency and area overheads, making it a prohibitive cost for NZ-based scheduling. This further limits the accelerator performance and scalability, mandating the deployment of low-overhead NZ-based scheduling. We observe that the metadata encoded in a widely-used compression format, CSR, can be leveraged to minimize the number of partial sums generated, reducing the overheads associated with NZ-based scheduling. Based on this observation, we propose low-overhead NZ-based scheduling that uses the CSR format to assist in partial sum tracking and accumulation. Following this methodology, we present SAGA, a Sparsity-Agnostic Graph Convolutional Accelerator that implements the low-overhead NZ-based scheduling. These overheads are bounded by the accelerator configuration and do not depend on the sparsity of the input matrices, making SAGA ideal for processing real-world power-law graphs.

Specifically, this paper makes the following contributions:

- To the best of our knowledge, this paper is the first work to introduce NZ-based scheduling for GCN computational kernels. The proposed scheduling can enable even distribution of CSR-compressed sparse matrix in GCN acceleration, thus achieving near-optimal resource utilization with ultra-low overheads. The proposed methodology can be generalized to support a wide range of applications where SpMM is the major computational kernel.
- We proposed an efficient and scalable hardware accelerator for graph convolutional neural networks, including a task scheduler, an accumulation table, and a partial row accumulation unit. These novel components enable an efficient non-zero scheduling by leveraging the metadata stored in the CSR compression format to alleviate the overheads of NZ-based scheduling while eliminating the need for data preprocessing and reformatting. This leads to high computational parallelism with minimal overheads.
- We prototyped the proposed design through FPGAs, and experimental results show that SAGA can consistently achieve high PE utilization ($> 99\%$), leading to 1.56× speedup and 2.05× energy savings on average as compared to the prior art [1].

*Equal Contribution.

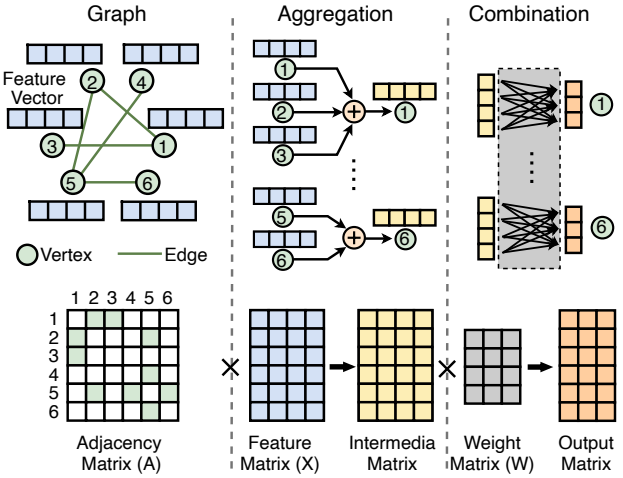



Figure 1: GCN basics. The computation in a GCN layer consists of two phases, namely *Aggregation* and *Combination*. The figure shows an example graph with six nodes, the adjacency matrix, feature vectors, and the weight matrix.

## II. Background

### A. GCN Basics

Graph convolutional networks (GCNs) have been largely used for processing non-Euclidean graph data due to their efficient neural network-based schemes. Figure 1 illustrates the two main computation phases for each graph convolution layer: Aggregation and Combination. During the aggregation phase, each vertex gathers and aggregates the features of its neighbor vertices, which will be used for updating its local features. As such, the aggregation phase can be formulated as a matrix-matrix multiplication involving the adjacency matrix of the graph and the feature matrix. During the combination phase, the local feature vectors are used as an input for a Multi-Layer Perceptron (MLP) network to transform such features to a lower dimension while retaining the structural information and characteristics of the graph. This reduced dimension feature vector of the nodes, which includes both the node feature information and the connectivity of the nodes in the graph, can be used to process various downstream tasks, including node classification, link prediction, and graph classification [15]. Equation 1 shows the overall computation for $l$th layer of a multi-layer GCN model, which can be considered as a chain matrix-matrix multiplication.

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

where $\hat{A} = D^{-\frac{1}{2}}\tilde{A}D^{-\frac{1}{2}}$ is the normalized adjacency matrix, $\tilde{A} = A + I$ is the self-loop adjacency matrix which shows the connections between each vertex in the graph, and $I$ is the identity matrix. D is the degree matrix, where $D_{ii} = \Sigma_j \tilde{A}_{ij}$. $X^{(l)}$ denotes the feature matrix, $W^{(l)} \in \mathbb{R}^{h^{(l)} \times h^{(l+1)}}$ indicates the shared weight matrix and $h^{(l)}$ is the feature dimension size for $l$th layer. $\sigma$ is the non-linear activation function, such as $ReLU$. Equation 2 represents the widely used two-layer GCN model.

$$X^2 = \sigma(\hat{A}\sigma(\hat{A}X^0W^1)W^2) \tag{2}$$

The order of the chain matrix-matrix multiplication decides the types of matrix multiplication depending on the matrix sparsity, which varies in the *Aggregation* phase and *Combination* phase. In the *Aggregation* phase, the computation can be generalized as sparse-sparse matrix multiplication (SpGEMM) or sparse-dense matrix multiplication (SpMM) operations, whereas the computations of the *Combination* phase are considered as SpMM or dense matrix multiplication (DenseMM) operations. Despite these differences, the computation order of $A \times (X \times W)$ has been widely considered for GCN computation, as it can unify the computation in both *Aggregation* and *Combination* phases as SpMM.

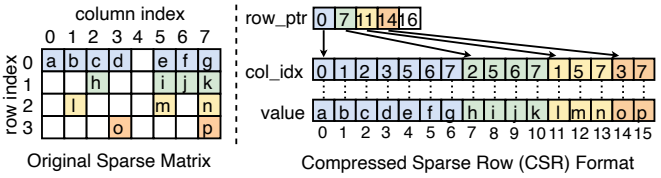



Figure 2: An example of CSR decompression.

### B. Compressed Sparse Row (CSR)

Significant research efforts have been made to exploit various compression techniques from both algorithm [16] and hardware [17] designs. The major idea of different compression formats is to eliminate zero elements from sparse matrices, reducing the excessive storage requirements. Despite much-reduced storage size, the compression formats could potentially jeopardize the data regularity and incur additional latency when reconstructing the original matrices [18]. For example, Compressed Sparse Row (CSR) stores non-zero values using column and row information, and thus it consists of three arrays to store row information ($row\_ptr$), column information ($col\_idx$), and non-zero values ($value$) as shown in Figure 2. The adjacent values in the row_ptr can infer two critical pieces of information. Firstly, its position in the pointer array can be used to determine the row number. Secondly, the values also represent the start and end indices of $col\_idx$ and $value$ arrays, and the difference of these two values indicates the number of non-zero elements at a given row. When performing SpMM, the coordinate information of non-zero elements of the sparse matrix is used for indexing the dense matrix to extract the corresponding value for multiplication at runtime. As such, $row\_ptr$ and $col\_idx$ arrays will be visited sequentially in order to locate the data value.

## III. Proposed SAGA Accelerator Design

GCN accelerators suffer from severe workload imbalance due to the power-law nature of graphs. Our proposed SAGA accelerator design aims to explore the efficient application of NZ-based scheduling without any preprocessing in GCN accelerators, achieving near-optimal workload balance with low overheads. The proposed SAGA accelerator includes two unique designs, a novel NZ-based scheduling, and an architectural design. First, we employ a widely-used CSR compression format to perform NZ-based scheduling without further reformatting the data. Then, SAGA leverages the metadata encoded in CSR to retain the intactness of sparse formats when performing NZ-based distribution of sparse matrices with low overheads.

### A. Overview of Proposed SAGA Architecture Design

As shown in Figure 3, the proposed SAGA consists of a host CPU, an off-chip DRAM, and an accelerator chip. The accelerator consists of a global buffer (GLB), a task scheduler, $N \times N$ processing elements (PEs), and a partial row accumulation unit and table. The host CPU, with a modified compiler which is used to configure the tile size, and loop order in the form of a set of instructions. The generated instructions are sent to the accelerator. The GLB is used to store both sparse and dense matrices needed for SpMM, and to store the intermediate/final data. The task scheduler dynamically decompresses

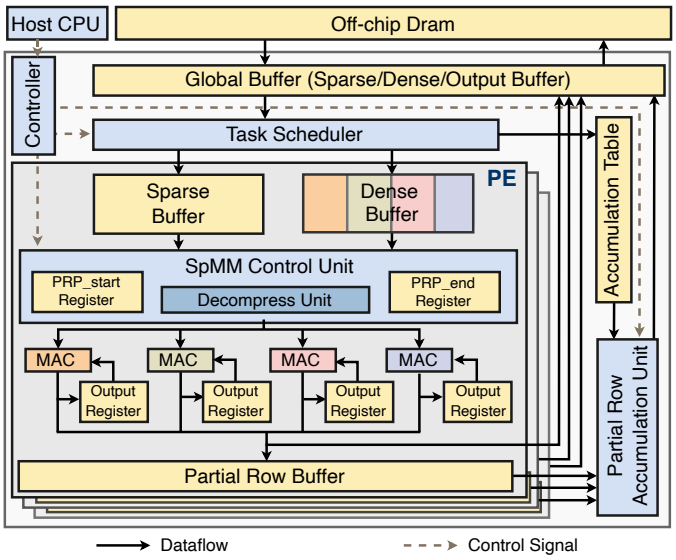



Figure 3: Proposed Micro-Architecture of SAGA.

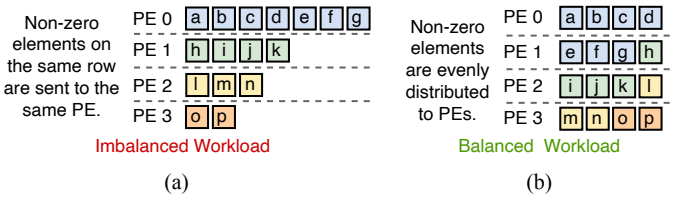



Figure 4: (a) Row-based Scheduling and (b) NZ-based Scheduling of the sparse matrix.

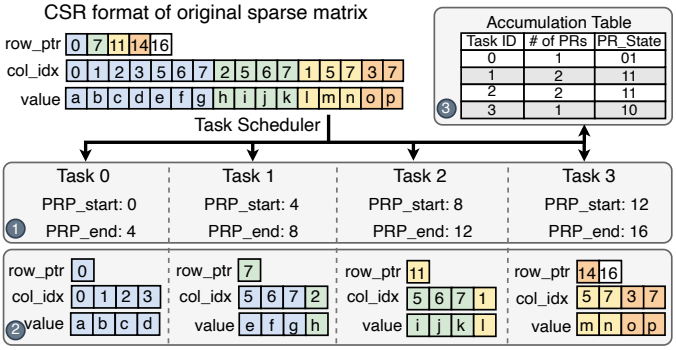



Figure 5: Proposed NZ-based Task scheduling in SAGA Design: (1) Partial Row Pointer (PRP) Generation, (2) $row\_ptr$, $col\_idx$, and $value$ arrays partitioning, and (3) Accumulation Table Update.

the sparse matrix in order to count the number of non-zero elements and distribute them to the PE array. During task scheduling, we record metadata about each task in an accumulation table. Upon receiving the assigned workload, each PE utilizes its MAC array to perform SpMM. The output of each PE will be sent directly to the global buffer or the partial row buffers based on whether the original row spans across multiple PEs. The output of such rows, called partial rows, will be sent to the partial row buffer. Then, the partial row accumulation unit uses the recorded metadata about the tasks from the accumulation table to stitch the partial rows together to produce the final desired result.

### B. PE Micro-architecture

The proposed PE architecture aims to facilitate SpMM. Figure 3 shows the PE architecture of our proposed SAGA accelerator. It consists of both sparse and dense buffers, an SpMM control unit, several MAC units (4 MACs in Figure 3), partial row pointer (PRP) registers, output registers, and partial row buffers. During the task scheduling, a part of the sparse matrix (e.g., adjacency matrix) is fetched to the sparse buffers from GLB in CSR format, and a part of the dense matrix (e.g., weighted features) is loaded into dense buffers in a dense format. The SpMM control unit first decompresses the sparse matrix to retrieve the coordinates of the non-zero value in the sparse matrix, then the corresponding elements of the dense matrix are read from the dense buffers and distributed to MAC units. The CSR decompression and MAC operations are pipelined to hide the latency incurred due to decompression as described in III-D.

### C. Proposed NZ-based Scheduling

Current GCN accelerators mostly adopt row-based scheduling, in which the sparse matrices are partitioned and assigned from row to row as shown in Figure 4(a). Unfortunately, the sparsity varies significantly in the sparse matrix, thus leading to unbalanced workload distribution at the PE level. The key idea of NZ-based scheduling is to partition the workload by using the compression metadata to count the non-zero elements so that the workload can be estimated. Despite the promising benefits, the direct deployment of NZ-based scheduling is challenging in GCN accelerators, as non-zero values from the same matrix row could be scattered and assigned to multiple processing elements. Figure 4(b) shows the non-zero elements of the same row are assigned to PE 0 and PE 1. Such distribution of non-zero elements creates two major issues, namely (1) partial sums created by different PEs must be accumulated across physically-distributed workloads and (2) complex tracking of non-zero elements across multiple PEs. The non-zero elements of the adjacency matrix refer to the edges in the graph. As the number of edges increases quadratically with increased vertices, the overheads caused by such scattered distribution can become a prohibitive cost for NZ-based scheduling.

To minimize these overheads, rather than tracking non-zero elements (edges) individually, we take a higher granular approach where we track on a per-row basis (vertices). We propose modified NZ-based scheduling alongside the observation that the CSR compression format encodes information about the rows of the sparse matrix that can be leveraged to assist in a higher granular NZ scheduling.

*1) Proposed Task Partitioning:* The proposed NZ-based scheduling requires generating tasks based on rows rather than individual non-zero elements. To facilitate this, we directly partition the sparse matrix in its CSR format into workload-balanced partitions called tasks. The task partitioning is performed inside the proposed task scheduler. Firstly, The task scheduler receives a tiled sparse matrix and a tiled dense matrix from the global buffer. Then, it partitions the sparse matrix into tasks that are then assigned to multiple PEs.

To generate workload-balanced tasks, the $row\_ptr$ array can be used to identify the non-zero elements associated with each row. However, naively partitioning the $row\_ptr$ array could compromise the CSR compression format. For example, a $4\times8$ sparse matrix is partitioned into four tasks as shown in Figure 5, and each task contains four non-zero elements. Consequently, $row\_ptr$, $col\_idx$, and $value$ arrays are partitioned into four tasks. In task 0, the original partitioned $row\_ptr$ (0) could no longer be used to reconstruct the partitioned value array according to the CSR decompression.

To resolve this issue, we introduce a new concept called partial row pointer (PRP) to retain the coordinate information of partitioned rows without compromising its original sparse format. If recalled, the $row\_ptr$ records the indexes (starting and ending positions in the value array) of non-zero elements associated with the same row in the

**Algorithm 1:** Non-zero element-based Task Partitioning in CSR format

**Input :** Number of PEs: $p$; CSR format of sparse matrix: $spm.val$, $spm.col\_idx$, and $spm.row\_ptr$; NNZ in the sparse matrix: $spm\_nnz$

**Output:** CSR matrix of each task: $tasks[\,].val$, $tasks[\,].col\_idx$, and $tasks[\,].row\_ptr$

1 **Function** *Tasks_CSR_Matrices_Generation(p, spm.val, spm.col_idx, spm.row_ptr, spm_nnz)*
2 $task\_nnz \leftarrow spm\_nnz/p$; // NNZ per task
3 $sched\_nnz \leftarrow 0$; // NNZ scheduled so far
4 $spm\_row\_idx \leftarrow 0$;
5 **for** $i \in [0, p)$ **do**
6 // Move part of $val$, $col\_idx$ to task $i$
7 **for** $j \in [0, task\_nnz)$ **do**
8 $tasks[i].val[j] \leftarrow spm.val[j + sched\_nnz]$;
9 $tasks[i].col\_idx[j] \leftarrow spm.col\_idx[j + sched\_nnz]$;
10 **end**
11 $task\_row\_idx \leftarrow 0$;
12 // Move $row\_ptr$ to task $i$ based on PRP
13 **while** $spm\_row\_idx < len(spm\_row\_ptr)$
14 *and* $spm.row\_ptr[spm\_row\_idx] >= PRP_i^{start}$
15 *and* $spm.row\_ptr[spm\_row\_idx] <= PRP_i^{end}$ **do**
16 $tasks[i].row\_ptr[task\_row\_idx] \leftarrow spm.row\_ptr[spm\_row\_idx]$;
17 $task\_row\_idx \leftarrow task\_row\_idx + 1$;
18 $spm\_row\_idx \leftarrow spm\_row\_idx + 1$;
19 **end**
20 $sched\_nnz \leftarrow sched\_nnz + task\_nnz$;
21 **end**
22 **return** $tasks$;

value array. Resizing the $value$ array has to reformat the starting and ending indexes in $row\_ptr$ array. As such, PRP_start and PRP_end are designed to record the row information for the partitioned sparse matrix. PRP_start and PRP_end will be sent to each PE for SpMM computation which will be further discussed in Section III-D.

In general, the PRP_start and PRP_end for each task can be automatically generated by using Equation 3, where $N_{nzp}$ is the task size (i.e. NNZ elements per PE), and $i$ is the PE ID.

$$PRP_i^{start} = i \times N_{nzp},\ PRP_i^{end} = (i+1) \times N_{nzp} \tag{3}$$

Furthermore, the $row\_ptr$, $col\_idx$, and $value$ arrays of each task have to be partitioned in accordance with the task size. Given this, a task partitioning algorithm is proposed to automatically partition the sparse matrix, as illustrated in Algorithm 1. The $col\_idx$ and $value$ arrays are partitioned based on the task size (lines 5-10). The $row\_ptr$ array will be sized based on PRP_start and PRP_end (lines 13-19). Any row pointers within the range of PRP_start and PRP_end will be included in the task. For example, as shown in Figure 5, only the first row pointer (0) is within the PRP_start (0) and PRP_end (4) in task 0. As a result, the row pointer (0) is included in task 0.

*2) Accumulation Table:* Due to our NZ-based scheduling, the original rows may be segmented and assigned to multiple PEs. These segmented rows generate partial sums that must be accumulated to produce the desired result. This requires additional information to record the partial rows that are assigned to the PEs for partial sum accumulation. To this end, we propose an Accumulation Table to keep track of those partial rows.

As shown in Figure 5, the proposed accumulation table includes Task ID, the number of partial rows (# of PRs), and the status of partial rows (PR_State). The task ID is used to indicate which PE the task is assigned to. The number of partial rows is used to indicate how many partial rows we have for each task. The PR_State is used to indicate whether the task needs to be accumulated with its two adjacent tasks. We note that each task has two partial rows shared between other adjacent tasks. As such, two bits are utilized to indicate the partial row status, one for each potential partial row. For each bit, '0' means no partial row, and '1' means a partial row. To generate the 'PR_State' bits during the task scheduling phase, the first and the last value of the $row\_ptr$ array are compared with the 'PRP_start' and 'PRP_end' bits, respectively. If they do not match, then the corresponding 'PR_State' bit is set to '1'. For example, in Figure 5, the $row\_ptr$ array of Task 3 has 2 elements. Since the first element, 14, does not match the 'PRP_start' bit, 12, of the corresponding task, the first bit of 'PR_state' is set to 1. Similarly, as the last element of the $row\_ptr$ array, 16, matches with the 'PRP_end' bit, 16, the second bit of 'PR_state' is set to 0. The '# of PRs' of Task 3 is '1'. The accumulation table is used for inter-PE accumulation, which will be further discussed in Section III-E.

### D. SpMM Computation in SAGA

Due to our NZ-based scheduling, the sparse matrices of each task are in the CSR format. Therefore, the SpMM computation must first decompress the sparse matrix and then perform the inner product matrix multiplication. The decompression unit is used to pair the sparse matrix and dense matrix for SpMM, and the MAC array is used to parallelize the matrix multiplication. As described in section III-C1, the partitioning of CSR-compressed sparse matrix in the task scheduling phase has modified the $row\_ptr$ array and cannot be decompressed using conventional CSR decompression. Therefore, the $row\_ptr$ array must be reconstructed before decompressing the sparse matrix.

Specifically, the task scheduler sends the partitioned sparse matrix along with PRP_start and PRP_end to PEs. Such information will be used to reformat the $row\_ptr$ information, thus enabling to retrieve the coordinate information of the non-zero elements of the sparse matrix. As shown in Figure 6 (a), it is not possible to decompress CSR format with partitioned $row\_ptr$ directly. The $row\_ptr$ has to be reformed with PRP. There are two steps required for decompression. Firstly, the partitioned $row\_ptr$ should be combined with related PRP. For example, in task 1, PRP_start (4) and PRP_end (8) are appended into $row\_ptr$ vector, so the $row\_ptr$ is $[4, 7, 8]$. Afterward, all elements in the $row\_ptr$ are subtracted by the PRP_start (4). After the subtraction, the reconstructed $row\_ptr$ becomes $[0, 3, 4]$, and it will be utilized to perform the CSR format decompression.

To overlap the decompression latency, we propose a four-stage decompression pipeline. An example of the proposed design is shown in Figure 6 (b), and an illustration of the pipeline is depicted in Figure 6 (c). These four stages include D1, D2, DR, and MAC. In D1, we fetch two values from the $row\_ptr$ vector to obtain the row information, and then the row information will be recorded and sent to retrieve column information. In D2, a few values will be fetched from the $col\_idx$ vector based on row information. This gives us the coordinate information of the non-zero elements within the sparse matrix. In the DR stage, based on the coordinate information of the sparse element, the operands from both sparse and dense matrices will be read for SpMM. In the proposed PE architecture, we design an array of MACs to support inner product matrix multiplication, as the sparse matrix will be temporarily reused to avoid costly compression/decompression operations. In the inner product matrix multiplication, each row of the sparse matrix will be multiplied by each column of the dense matrix. Each column of the dense matrix is assigned to a MAC unit from the MAC array. Therefore, the operands

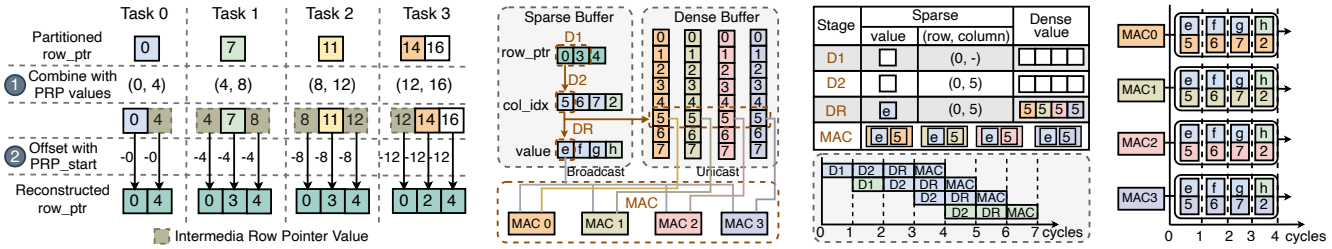



Figure 6: (a) Example of reconstructing new row pointers with PRP values for all four tasks, (b) example of four-stage decompression with reconstructed row pointers for Task 1 in PE 1, (c) illustration of pipelined four-stage decompression, and (d) SpMM computations in PE 1, where elements from sparse and dense matrices are partitioned into four MAC units.

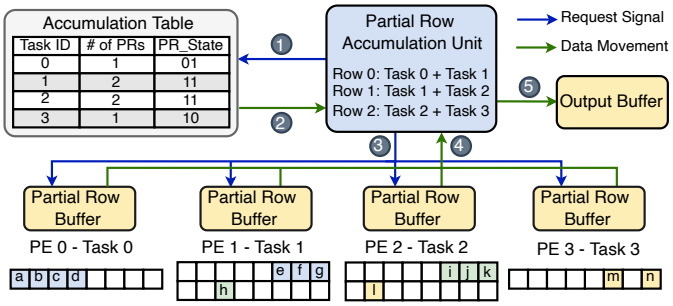



Figure 7: A workflow of partial row accumulation in SAGA.

in the sparse buffer will be broadcast to all MAC units, and the corresponding operands of the dense matrix for each column will be unicast to different MAC units to parallelize the SpMM. In the MAC stage, MAC operations are performed. Each MAC unit multiplies the paired sparse and dense elements, and it skips the zero elements to reduce power consumption and latency. As such, the number of MAC operations depends on the non-zero elements, and due to the NZ-based scheduling, which balances the non-zero elements per task, the utilization rate of PE is near-optimal. Figure 6 (d) shows all SpMM operations for Task 1 in MAC units of PE 1.

### *E. Proposed Partial Row Accumulation*

Due to the presence of partial rows in tasks, the partial sum generated by these partial rows must be accumulated spatially across the MAC array or PEs. The information recorded in the accumulation table can be used to accumulate the partial sums.

After each PE performs the SpMM, if any partial rows exist, the partial results of these rows will be stored in the partial row buffers. Otherwise, the final result of a row will be sent to the output buffer in the global buffer. To perform the partial sum accumulation, the partial row accumulation unit will send a request to the accumulation table to go over all the tasks in a pipelined fashion. By reading each entry, the partial row accumulation unit will send related requests to PEs. Upon receiving the request, the PE will send the data stored in partial row buffers to the accumulation unit. The accumulation unit will further perform partial row accumulation. For example, Figure 7 shows that the 'PR_State' bits for Task 0 are '01'. As the second bit is '1', the final row of Task 0 must be accumulated with the first row of the next task. Similarly, the last row of Task 1 must be accumulated with the first row of Task 2. The number of partial sums generated by the partial row is equal to the number of columns in the dense matrix. As such, the accumulation of the partial sums can be parallelized in the accumulation unit by using multiple adders. The final results will eventually be sent to the output buffer.

*1) Upperbound of Partial row accumulation:* As we are operating on a row-wise granularity during the NZ scheduling, the sparse matrix of each task contains only two partial rows: one partial row at the beginning with the previous task and one partial row at the end with the next task. As such, the number of partial rows depends on the number of tasks, $T_n$. As the first task cannot have a partial row with the previous task and the final task cannot have a partial row with the next task, the total number of partial rows is bounded by $2T_n - 2$. Generally, we assign one task per PE. Therefore the partial row accumulation overhead scales linearly with the number of PEs and is independent of the sparsity of the sparse matrix. This leads to low partial row accumulation overhead as discussed in Section IV-C. Furthermore, this overhead can be overlapped with the SpMM computation of the next sparse tile.

## IV. Evaluation

We implemented SAGA using Verilog RTL, synthesized and implemented the design using Xilinx Vivado 2020.2 on Xilinx Alveo U200 FPGA board with 892,000 LUTs, 5867 DSP slices, 35MB of on-chip memory storage and 77 GB/s off-chip bandwidth. We evaluated the performance of SAGA using five datasets, including three citation graphs (Cora, CiteSeer, and Pubmed) [19], one knowledge graph (Nell) [20], and one large graph (Reddit) [21]. The detailed sparsity and size of data sets are summarised in Table I. We used the standard GCN algorithm as described in the paper [22]. We measured the execution time and PE utilization of SAGA by running each dataset 200 times at 250 MHz and taking the average of all the runs. Even though we evaluated SAGA on FPGA, we did not leverage any FPGA-specific properties to speed up SAGA performance. SAGA is proposed as an ASIC accelerator.

**Baseline:** To demonstrate the efficiency and scalability of SAGA, we compare it with prior GCN accelerators AWB-GCN [1], HyGCN [5], GCNAX [6] and FlowGNN [23]. For a fair comparison, all the baseline accelerators are scaled to have the same number of multiply-and-accumulate units as SAGA. As HyGCN uses a hybrid engine consisting of SIMD cores and systolic modules for the aggregation phase and combination phase, respectively. We configured the systolic array with 8× more multipliers than the SIMD cores. Similarly, FlowGNN utilizes a hybrid engine that supports multiple GNN models. However, we primarily compare against the GCN variant of FlowGNN. We scaled the bandwidth and on-chip memory to match SAGA configured with 5 MB of Global Buffer and 7 MB of Local Buffers. Like HyGCN, we use fixed point multipliers, whereas AWB-GCN and GCNAX use floating point multipliers and were scaled accordingly.

**Energy and Area Evaluation:** To estimate the power and area of SAGA as an ASIC accelerator, we used Synopsys Design Compiler

Table I: Size and Sparsity of the Graph Datasets

| Datasets | Vertexes | Features | Density (A, X1, X2, W) |
|---|---|---|---|
| Cora | 2,708 | 1,433 | 0.18%, 1.27%, 78.0%, 100% |
| CiteSeer | 3,327 | 3,703 | 0.11%, 0.85%, 89.1%, 100% |
| PubMed | 19,717 | 500 | 0.023%, 10.0%, 77.6%, 100% |
| Nell | 65,755 | 61,278 | 0.0073%, 0.011%, 86.4%, 100% |
| Reddit | 232,965 | 602 | 0.17%, 51.6%, 60.0%, 100% |

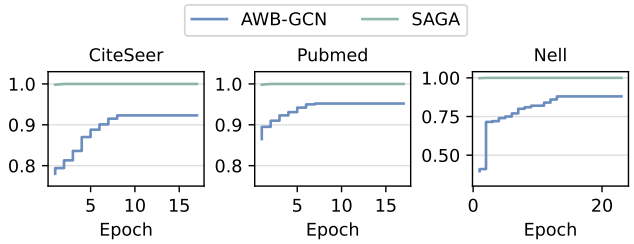



Figure 8: Average PE utilization of AWB-GCN (PE = 1K) and SAGA (PE = 64, MAC = 16) per unit time

with the TSMC 40 nm standard library to synthesize and generate the waveform activity file to capture the dynamic switching activity of the logic gates. Afterward, we use Synopsys PrimeTime PX with the waveform activity file to measure the dynamic and static power consumption of SAGA. For the area and power of on-chip buffers, we use Cacti 6.0 [24] with 40 nm technology.

### A. Performance Evaluation

*1) Workload Balance Analysis:* To evaluate workload balance techniques, we compare the PE utilization of SAGA with AWB-GCN. We use performance measure counters in each PE to keep track of the number of cycles when the MAC units are active. We divide the execution time of each dataset into epochs following AWB-GCN's computational order. Each epoch roughly equals the execution of an entire column of the dense matrix according to AWB-GCNs column-wise computations. We configure SAGA to have 64 PE and 16 MAC units per PE to equal 1024 MAC units in total and compare with AWB-GCN with 1K PEs. We plot the average utilization of all PEs for every epoch and compare AWB-GCN and SAGA in Figure 8. Since other GCN accelerators deploy static workload balancing approaches, we do not include them here.

Figure 8 shows that for Nell, AWB-GCN achieved a maximum PE utilization of 0.85 while SAGA has a utilization close to 1. SAGA can achieve up to 18% higher PE utilization for Nell and around 12% on average compared to the PE utilization of AWB-GCN. This is because Nell has extremely irregular sparsity, exacerbating the workload balance issue. As such, AWB-GCN struggles to improve PE utilization above 0.85. Notice that AWB-GCN requires a handful of epochs to achieve a constant sub-optimal PE utilization. Each epoch can span over a few clock cycles depending on the number of rows in the dense matrix. Moreover, AWB-GCN uses matrix blocking to accommodate large graphs, which divides the matrices into smaller tiles and performs computation on each smaller tile separately. This forces AWB-GCN to rebalance the workload when a new tile of the sparse matrix is loaded. Such a frequent need for workload balancing can limit computational parallelism. In contrast, although SAGA performs matrix blocking, due to its NZ-based scheduling, we see constant PE utilization of close to 1 across tiles of the sparse matrix making it desirable to perform workload balance during scheduling.

*2) Execution Time Analysis:* We compare SAGA (PE = 256, MAC/PE = 16) with AWB-GCN (4096 PEs), HyGCN, GCNAX, and FlowGNN by running GCN-inference on the five datasets. Additionally, all baselines were scaled to have the same MAC units. Figure 9 shows the speedup of SAGA and other baseline accelerators normalized to the execution time of AWB-GCN. Results show that SAGA provides an average speedup of 1.59× compared to AW-GCN, whereas GCNAX, FlowGNN, and HyGCN have an average speedup of 0.852×, 0.905×, and 0.36× respectively. SAGA performs over 2× better than AWB-GCN for Nell and just under 1.06× for Reddit. This variation in speedup is due to the sparsity in the Nell and Reddit graphs. Nell has extremely high and variable sparsity which causes PE underutilization and, subsequently, higher execution time for AWB-GCN, Meanwhile, Reddit is more regular making GCN processing inherently balanced with high PE utilization. However, unlike Reddit, most real-world power-law graphs are extremely imbalanced. Therefore it is desirable to have a GCN accelerator like SAGA that performs well with highly irregular graphs. Note that while GCNAX and FlowGNN optimize other aspects of GCN processing such as data reuse and parallelization strategies, the PE utilization reduces significantly when scaling up the design leading to lower overall performance. The speedup improvement for SAGA compared to HyGCN, FlowGNN, and GCNAX is primarily due to its near-optimal workload balance and low-hardware complexity.

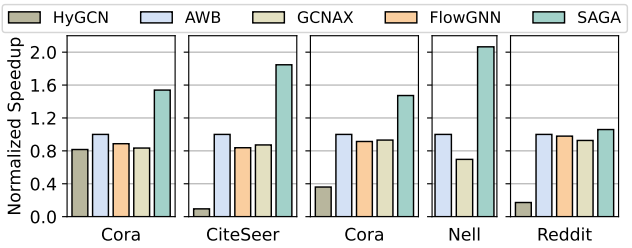



Figure 9: Normalized speedup comparison of SAGA and baseline accelerators for different datasets.

### B. Scalability Study

To evaluate the scalability of SAGA, we run GCN inference on the five datasets and vary the number of PEs. To provide a fair comparison with AWB-GCN, HyGCN, GCNAX, and FlowGNN, we equate the overall number of MAC units in SAGA with a fixed 16 MAC/PE. Additionally, we normalize the execution time of all configurations of accelerators to the execution time of AWB-GCN with 512 PEs configuration. FlowGNN and GCNAX use smaller evaluation designs which alleviate PE underutilization. To make a realistic comparison, when scaling up their design, we estimate the PE underutilization, given their workload balance scheme and PE mapping strategies, and normalized their performance metrics accordingly.

*1) PE Utilization:* Figure 10 shows the PE utilization of SAGA with various baseline accelerators. The average PE utilization of SAGA, AWB-GCN, GCNAX, HyGCN, and FlowGNN reduces by 0.2%, 4.875%, 20.569%, 13.712%, 27.143% respectively across different data sets when scaling up from 512 to 4K PEs. While AWB-GCN's PE utilization degradation is lower than the other baselines owing to its runtime workload rebalancing, For massively parallel architectures, such reduction in PE utilization can cause significant performance degradation. For highly imbalanced graphs such as Nell, the PE utilization of SAGA, AWB-GCN, GCNAX, HyGCN, and FlowGNN reduces by 0.19%, 9.01%, 23.41%, 24.10%, 49.7% respectively. Such severe PE underutilization for graphs is undesirable as real-world graphs follow the power-law distribution, which makes them highly imbalanced. SAGA's reduction in PE utilization is minimal due to its non-zero scheduling.

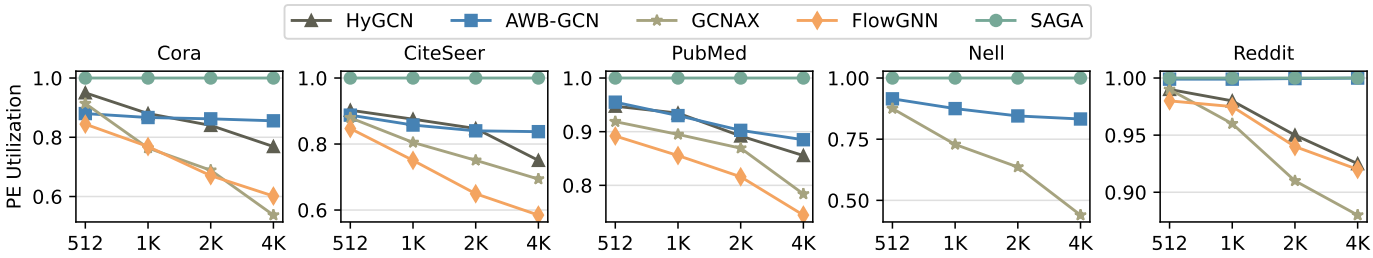



Figure 10: Scalability evaluation: Overall PE utilization of SAGA, AWB-GCN, GCNAX, FlowGNN and HyGCN with varying PE number. We use a fixed MAC/PE = 16 for SAGA and vary the number of PEs to have the same number of MAC units as others

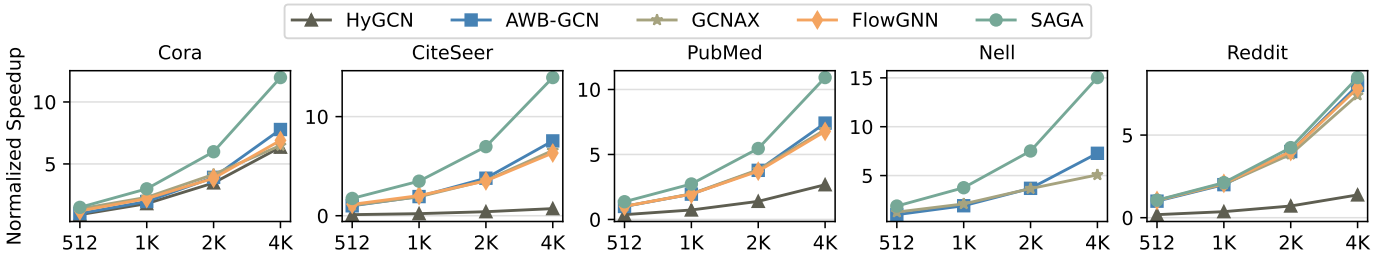
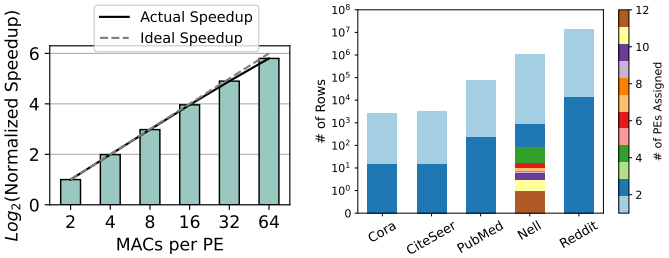



Figure 11: Scalability evaluation: Overall speedup of SAGA, AWB-GCN, GCNAX, FlowGNN, and HyGCN with varying PE numbers. We use a fixed MAC/PE = 16 for SAGA and vary the number of PEs to have the same number of MAC units as others




(a) (b)

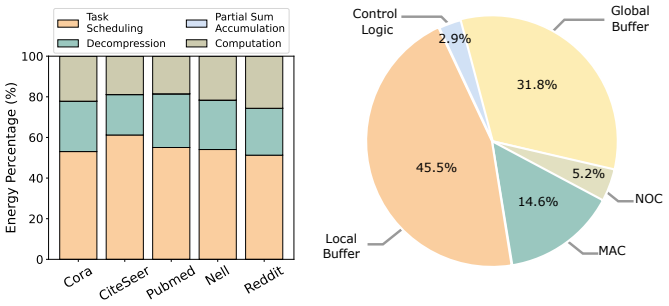
Figure 12: (a) Performance scaling with an increase in MACs/PE with 64 PEs (b) Number of rows for each dataset assigned to varying number of PEs.

*2) Performance:* As shown in Figure 11, the overall speedup scales well for SAGA compared to AWB-GCN, HyGCN, GCNAX, and FlowGNN, which is similar to the PE utilization scaling, indicating the correlation between performance and PE utilization. HyGCN does not completely mitigate the workload imbalance among the SIMD PE cores during the aggregation phase, which is the majority of execution time, leading to poor performance speedup with an increase in the number of PEs. On the other hand, AWB-GCN has a better speed-up improvement with an increase in PE cores as it rebalances workload distribution during runtime. However, for a large number of PEs, the performance gain is sub-optimal, especially for highly irregular graph data such as Nell. This is due to the inability of AWB-GCN's runtime workload auto-tuning to converge to a more balanced workload. On the other hand, the performance of SAGA scales linearly with an increase in the number of PEs. This is because SAGA does not suffer from the same drawbacks as other accelerators. Therefore SAGA has an average normalized speedup of 11.83× compared to AWB-GCNs speedup of 7.60×. Irrespective of the sparsity, SAGA can schedule balanced workloads, thus, achieving a normalized speedup of 15.03× compared to AWB-GCN's speedup of 7.28× for the highly irregular Nell dataset when configured with 4K PEs.

Finally, SAGA can exploit feature-level parallelism by increasing the number of MAC units per PE. Each MAC in a PE operates on a different feature (column of the dense matrix) of the same vertex. To evaluate the performance scaling by increasing the number of MACs/PE, we run inference using the five datasets and report the average execution time for MACs/PE values of 2, 4, 8, 16, 32, and 64. Figure 12 (a) shows the normalized speedup in logarithmic scale. Due to the computation scheduling among the MAC units within a PE, we see a near-ideal performance scaling. For a large number of MACs per PE, the speedup is lower than ideal. This is due to the limited number of features for different layers of GCN models. For example, while layer 1 for Cora has 64 features, layer 2 has only 16.

### C. NZ-Scheduling Overhead Analysis

The microarchitecture of SAGA allows the task scheduling, decompression, and partial sum accumulation to be overlapped with the computation. The latency from task scheduling is due to internal on-chip data movement from the global buffer to the local buffers of each PE. This latency is minimal due to three reasons. (i) The sparse data is stored on-chip in compressed format, (ii) Task scheduling performs sequential reads and writes to the global buffer and local buffer respectively, and (iii) The global buffer and local buffer are highly banked increasing their bandwidth. The decompression is performed at the PE level and is pipelined with the MAC computation. Therefore, most of the decompression latency is hidden. Finally, most of the partial sums are accumulated within the PE. The remaining partial sum accumulation across PEs due to the presence of partial rows is small in number and can be overlapped with the execution of the next set of tasks. Figure 12 (b) shows the number of rows that are assigned to multiple PEs. Due to our NZ scheduling, we observe that more than 99% rows are assigned to one PE, and less than 1%, will be split across multiple PEs. For Nell, due to the irregular sparsity, there are few rows that exhibit high density. Our NZ scheduling distributes such rows across multiple PEs with one of them assigned to 12 PEs. However, the number of partial row accumulations is limited by the accelerator configuration and is negligible compared to the total execution time.




(a) (b)

Figure 13: (a) Energy Breakdown and (b) Area Breakdown.

Table II: Energy and Latency Analysis

| Platform | Metrics | Cora | CiteSeer | Pubmed | Nell | Reddit |
|---|---|---|---|---|---|---|
| AWB-GCN | Latency (ms) | 2.3E-3 | 4E-3 | 3E-2 | 1.6 | 31.8 |
| | EE (Graphs/kJ) | 3.1E6 | 1.9E6 | 2.5E5 | 4.1E3 | 2.1E2 |
| SAGA | Latency (ms) | 1.49E-3 | 2.22E-3 | 2.04E-2 | 0.77 | 30.0 |
| | EE (Graphs/kJ) | 6.23E6 | 3.67E6 | 5.08E5 | 1.11E4 | 3.32E2 |

### D. Energy Analysis

Figure 13 (a) shows the breakdown of energy consumed by each operation of SAGA. Task Scheduling, decompression, computation, and partial sum accumulation account for 56%, 23%, 20%, and 1% of total energy on average, respectively. Task scheduling involves moving the data from the global buffer to the local buffer and decompression involves reading the indirection tables of the compression format stored in the on-chip buffers. Since both of the components reads and write to the on-chip buffers, it accounts for around 79% of the overall energy consumed. Meanwhile, computation involves MAC operations, and partial sum accumulation involves aggregation which consumes around 21% of total on-chip energy. Table II shows the energy efficiency of SAGA compared to AWB-GCN. SAGA achieved an average energy improvement of 2.05× and up to 2.68× for Nell Dataset. This can be attributed to the following reasons (i) Unlike AWB-GCN's large all-to-all network, SAGA's is lightweight and consumes much less dynamic power, and (ii) AWB-GCN's runtime workload rebalancing and monitoring of workload consume a large portion of the dynamic power, whereas SAGA does not require additional monitoring logic.

### E. Area Analysis

Figure 13 (b) shows the area consumed by each logical resource of SAGA's architecture. Global buffer and local buffers account for over 77% of the total area, whereas MAC units consume 14.6%. Due to SAGA's lightweight networks, the area occupied by the network is around 5.2%. Finally, the control logic takes 2.9% of the total area.

## V. Related Work

### A. Sparse Matrix Multiplication Accelerators

A lot of effort has been put into accelerating sparse matrix multiplication (SpGEMM and SpMM), resulting in the development of various solutions which include CPU-based [25], GPU-based [26], FPGA-based [27], and ASIC-based [28] designs. CPU- and GPU-based solutions [25], [26] aim to increase computational parallelism through the use of a multiply-insert approach but often result in high power consumption and latency due to the utilization of general-purpose computing units. To address this issue, FPGA-based designs [27] incorporate customized PE and interconnection architectures to enhance data locality and energy efficiency. The approaches proposed in OuterSpace [28] and SpArch [29], which are based on outer-product SpGEMM and ASIC accelerator designs, respectively, aim to improve data reuse. However, these existing solutions offer limited guidance on how to support chain matrix multiplication in GCNs efficiently.

### B. Graph Processing Accelerators

Graph processing accelerators are specialized hardware designed for a set of algorithms and techniques used for analyzing and manipulating graphs, such as graph traversal algorithms (BFS or DFS), graph clustering algorithms (k-means or spectral clustering), and graph partitioning algorithms (METIS or KaHIP). Different from GCNs, each vertex is associated with a single scalar instead of a vector. Memory bandwidth is the main bottleneck for graph processing, and most graph processing accelerators are PIM-based with different graph partitioning and mapping algorithms. GraphR [30] is ReRAM-based and performs SpMV using ReRAM crossbars. Graphicionado [31] and GraphQ [32] use destination-oriented mapping to optimize off-chip bandwidth consumption. ScalaGraph [33] proposes row-oriented mapping to reduce NoC communication. All of them only support simple operations in the scatter phase. Therefore, they can't be migrated directly to GCNs without significant changes.

### C. Graph Convolutional Network Accelerators

GCN accelerators are designed to enhance the performance of chain matrix multiplications. For instance, HyGCN [5] utilizes two separate computing engines to perform aggregation and combination operations. The aggregation engine suffers from an unbalanced workload due to varying degrees of the graph. To tackle this, all the cores of the aggregation engine operate on different features of the same vertex. However, the parallelism of the aggregation engine is bounded by the feature number. For example, the number of features for the second layer of GCN for Cora, CiteSeer, and Pubmed is only 16. On the other hand, AWB-GCN [1] assigns workload to each PE based on the row number. Then, it performs a runtime re-balancing of the workload by continuously monitoring the PE workload. However, it often suffers from PE underutilization as the sparsity can vary tremendously within the rows causing it to rebalance the workload very often. Furthermore, it employs complex workload monitoring logic and all-to-all switching circuitry that limits its scalability and energy efficiency. The accelerator design proposed by Auten et al. [34] focuses on efficiently handling irregular data movement to accelerate GCN executions. GRIP [35] divides GCN inference into two phases, vertex-centric and edge-centric execution, and employs specialized hardware units for each phase. GraphACT [36] aims to accelerate GCN training on heterogeneous systems with CPU and FPGA and utilize multiple hardware-software co-optimizations. GCNAX [6] and SGCNAX [37] provide a flexible GCN architecture that supports adaptive dataflow and improves resource utilization.

## VI. Conclusion

In this paper, we propose SAGA, an efficient and scalable GCN accelerator with near-optimal workload balancing. Specifically, SAGA consists of two unique designs, a NZ-based scheduling, and a novel accelerator architecture. The proposed NZ-based scheduling enables efficient distribution of sparse matrix at runtime, thus achieving near-optimal workload balancing. In addition, the proposed accelerator architecture, including a task scheduler, an accumulation table, and a partial row accumulation unit, can support the proposed NZ-based scheduling without data preprocessing and reformatting. We prototyped the proposed design through FPGAs, and our simulation results show that SAGA achieves 1.56× speedup and 2.05× energy savings on average as compared to the prior art [1].

## REFERENCES


[1] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. Herbordt. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.

[2] X. Wang, Y. Ma, Y. Wang, W. Jin, X. Wang, J. Tang, C. Jia, and J. Yu. Traffic flow prediction via spatial temporal graph neural network. In *Proceedings of The Web Conference (WWW)*, pages 1082–1092. IEEE, 2020.

[3] P. Sarlin, D. DeTone, T. Malisiewicz, and A. Rabinovich. SuperGlue: Learning feature matching with graph neural networks. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4937–4946. IEEE, 2020.

[4] S. Wang, V. Govindaraj, J. Górriz, X. Zhang, and Y. Zhang. Covid-19 classification by FGCNet with deep feature fusion from graph convolutional network and convolutional neural network. In *Information Fusion*, pages 208–229. Elsevier, 2021.

[5] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. HyGCN: A GCN accelerator with hybrid architecture. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

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

[7] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1051–1063. IEEE, 2021.

[8] L. Yin, J. Wang, and H. Zheng. Exploring architecture, dataflow, and sparsity for gcn accelerators: A holistic framework. In *Proceedings of Great Lakes Symposium on VLSI (GLSVLSI)*, page 489–495. ACM, 2023.

[9] B. Zhang, R. Kannan, and V. Prasanna. BoostGCN: A framework for optimizing GCN inference on FPGA. In *Proceedings of International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 29–39. IEEE, 2021.

[10] G. Huang, G. Dai, Y. Wang, and H. Yang. GE-SpMM: General-purpose sparse matrix-matrix multiplication on GPUs for graph neural networks. In *Proceedings of International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–12. IEEE, 2020.

[11] M. Zhu, T. Zhang, Z. Gu, and Y. Xie. Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 359–371. IEEE, 2019.

[12] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of International Conference on High Performance Computing Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2009.

[13] L. Song, Y. Chi, A. Sohrabizadeh, Y. Choi, J. Lau, and J. Cong. Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 65–77. ACM, 2022.

[14] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 689–702. IEEE, 2020.

[15] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. Wiltschko. A gentle introduction to graph neural networks. In *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[16] S. Kwon, D. Lee, B. Kim, P. Kapoor, B. Park, and G. Wei. Structured compression by weight encryption for unstructured pruning and quantization. In *Proceedings of IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1909–1918. IEEE, 2020.

[17] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. SIGMA: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[18] E. Qin, G. Jeong, W. Won, S. Kao, H. Kwon, S. Srinivasan, D. Das, G. Moon, S. Rajamanickam, and T. Krishna. Extending sparse tensor accelerators to support multiple compression formats. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1014–1024. IEEE, 2021.

[19] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, and T. Eliassi-Rad. Collective classification in network data. In *AI magazine*, pages 93–106, 2008.

[20] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka, and T. Mitchell. Toward an architecture for never-ending language learning. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, pages 1306–1313. AAAI, 2010.

[21] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Proceedings of Advances in neural information processing systems (NeurIPS)*, pages 1024–1034. MIT Press, 2017.

[22] T. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *arXiv preprint arXiv:1609.02907*, 2016.

[23] R. Sarkar, S. Abi-Karam, Y. He, L. Sathidevi, and C. Hao. FlowGNN: A dataflow architecture for real-time workload-agnostic graph neural network inference. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1099–1112. IEEE, 2023.

[24] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. CACTI 6.0: A tool to understand large caches. In *University of Utah and Hewlett Packard Laboratories, Technical Report*, 2009.

[25] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. In *SIAM Journal on Scientific Computing*, pages C624–C651, 2016.

[26] W. Liu and B. Vinter. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, pages 370–381. IEEE, 2014.

[27] C. Lin, N. Wong, and H. So. Design space exploration for sparse matrix-matrix multiplication on FPGAs. In *International Journal of Circuit Theory and Applications*, pages 205–219, 2013.

[28] S. Pal, J. Beaumont, D. Park, A. Amarnath, S. Feng, C. Chakrabarti, H. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. OuterSPACE: An outer product based sparse matrix multiplication accelerator. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[29] Z. Zhang, H. Wang, S. Han, and W. Dally. SpArch: Efficient architecture for sparse matrix multiplication. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.

[30] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen. GraphR: Accelerating graph processing using reram. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2017.

[31] T. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[32] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian. GraphQ: Scalable pim-based graph processing. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 712–725. IEEE, 2019.

[33] P. Yao, L. Zheng, Y. Huang, Q. Wang, C. Gui, Z. Zeng, X. Liao, H. Jin, and J. Xue. ScalaGraph: A scalable accelerator for massively parallel graph processing. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 199–212. IEEE, 2022.

[34] A. Auten, M. Tomei, and R. Kumar. Hardware acceleration of graph neural networks. In *Proceedings of ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

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

[36] H. Zeng and V. Prasanna. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of ACM International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 255–265. ACM, 2020.

[37] J. Li, H. Zheng, K. Wang, and A. Louri. SGCNAX: A scalable graph convolutional neural network accelerator with workload balancing. In *IEEE Transactions on Parallel & Distributed Systems*, pages 2834–2845, 2022.