UWB-GCN: Accelerating Graph Convolutional Networks through Runtime Workload Rebalancing

Tong Geng[†‡], Ang Li[‡], Tianqi Wang[†], Chunshu Wu[†], Yanfei Li[¶], Antonino Tumeo[‡], Shuai Che[§], Steve Reinhardt[§] and Martin Herbordt[†]

[†]Boston University
[‡]Pacific Northwest National Laboratory
[§]Microsoft Research
[¶]Zhejiang University

***Abstract*—Deep learning systems have been applied mostly to Euclidean data such as images, video, and audio. In many applications, however, information and their relationships are better expressed with graphs. Graph convolutional networks (GCNs) appear to be a promising approach to efficiently learn from graph data structures, having shown advantages in many critical applications such as power system analysis, chemical reactivity prediction, material property prediction, E-commerce, cybersecurity, etc. As with other deep learning modalities, hardware acceleration is critical. The challenge is that real-world graphs are often extremely large and unbalanced; this poses both significant performance demands and design challenges.**

We propose an architecture that accelerates GCN inference, the Ultra Workload Balanced GCN (UWB-GCN). To address the major performance bottleneck of workload imbalance we propose two techniques, dynamic local sharing and dynamic remote switching. Both rely on hardware flexibility to autotune the system; this is effected with negligible area and delay overhead. In particular, UWB-GCN profiles the sparse graph pattern while continuously adjusting the workload distribution strategy among a large number of processing elements (PEs). Once the system converges to an ideal configuration, this configuration is used for the remaining iterations. To the best of our knowledge, UWB-GCN is the first accelerator design targeting GCNs and the first that autotunes workload balance in the accelerator in hardware rather than software. These methods result in near-ideal workload balance in processing sparse matrices. Experimental results show that UWB-GCN can perform inference of the Nell graph (66K vertices, 266K edges) in 8.1 ms; this corresponds to speedups of 199×, 16×, and 7.5× as compared with, respectively, CPU, GPU, and a baseline design with no workload rebalancing.

I. Introduction

Deep learning paradigms such as Convolutional Neural Networks (CNNs) [23] and Long Short Term Memory (LSTM) [20] have been applied to a wide range of applications, from image classification, through video processing, to speech recognition, and to natural language processing. These paradigms are only able to extract and analyze latent information from euclidean data such as images, videos, audios and texts [37]. This fact limits the real-world applications of neural networks.

In the real world, an increasing number of applications, such as E-commerce [6], [42], molecular bioactivity identification in medical research [14], social network analysis [22], [35], cascading failure prediction of national power grid, etc., use non-Euclidean data structure and are modeled as graphs with nodes referring to the objects involved in target applications and edges representing the relations between nodes. For all these applications, graphs have tremendously large numbers of nodes which degrees vary dramatically, leading to significant data irregularity.

The irregularity in graph data makes most existing deep learning algorithms fall short and critical feature extraction operations, such as convolutions, not directly applicable. As a result, Graph Neural Networks have been proposed, in various forms, to extend deep learning approaches to graph data [10], [15], [27], [30], [34], [37]. Among these, the *Graph Convolutional Network* (GCN), an approach that marries some ideas of CNNs to the distinct needs of graph data processing, has demonstrated significant potentials [7], [11], [22].

With the rapid development of GCNs, designing specialized hardware accelerators for GCN inference has become an urgent issue. GCNs have already been adopted in multiple real-world applications, including electric grid cascading failure analysis [29], prediction of chemical reactivity [9], prediction of synthesized material property [38], modeling polypharmacy side-effects [49], accurate advertisement in E-commerce [41], cybersecurity [31], etc. [26], [37]. Many of these applications require low-latency, high-throughput GCN inference. Existing platforms, however, are not well-suited to handling the irregularity of the sparse graph data, thus hindering the practical utilization of GCNs.

Although sparsity has already been addressed in many existing sparse-CNN (SCNN) accelerator designs [2], [18], [21], [45], [47], the challenges of accelerating GCNs are significantly different. The first difference is the *source of sparsity*. For SCNNs, the majority of the sparsity comes from the weights (due to model redundancy). Two types of pruning/compression techniques have been proposed for leveraging this sparsity: structural pruning condenses the weight matrix during the training [12], [36], while unstructured pruning [18], [25], [47] can pre-profile the weight matrix for design specialization, as the weight matrix has been fixed before inference. For GCNs, however, the sparsity comes from the

TABLE I
MATRIX SPARSITY AND DIMENSIONS OF THE 2-LAYER GCNS FOR THE 5 GRAPH DATASETS. F1 AND F2 ARE THE INPUT FEATURES OF LAYER-1/2. F3 IS THE OUTPUT FEATURES OF LAYER-2.

		CORA	CITESEER	PUBMED	NELL	REDDIT
Density	A	0.18%	0.11%	0.028%	0.0073%	0.043%
	W	100%	100%	100%	100%	100%
	X1	1.27%	0.85%	10.0%	0.011%	51.6%
	X2	78.0%	89.1%	77.6%	86.4%	60.0%
Dimension	Node	2708	3327	19717	65755	232965
	F1	1433	3703	500	61278	602
	F2	16	16	16	64	64
	F3	7	6	3	186	41

input data itself and is only apparent at runtime. Therefore, existing weight-focused SCNN pruning and compression techniques cannot work well. In addition, compared with deep CNNs comprising dozens or even hundreds of layers, existing GCN models are usually very shallow, most of which are no more than three layers [46] [1]. Due to tremendous information aggregation at the nodes per layer, the weight matrix is very dense and has little redundancy.

The second difference is the *degree of sparsity*. For SCNNs, the dimensions of the weights are typically from dozens to hundreds, with sparsity around 50%. Thus, it can either be assumed that the weight matrix fit in on-chip memory (after compression) [12], [18], or a small scheduling window can be used for index matching [40] when multiplying the sparse matrices (i.e., pairing the rows of the first matrix with correct columns of the second matrix). For example, Cambricon-S [45], [47] and Stitch-X [25] compare hundreds (e.g., 256) of index pairs per cycle and assume that sufficient non-zeros (i.e., `nnz`) can always be found for each PE. This may be true for SCNNs, but for GCNs, the dimensions of the sparse matrices can range from thousands to millions, with sparsity larger than 99.9% (see Table I). This leads to insurmountable problems for both SCNN approaches. For the first, the sparse matrix no longer fits into on-chip memory, leading to irregular off-chip memory access. In the second, either the window size is kept the same and the system remains mostly idle, or the window size must be tremendously expanded making the hardware unfeasible.

The third difference is the *distribution of sparsity*. As shown in Figure 1, for SCNNs the distribution of number of non-zeros per row or column is roughly balanced so the impact of workload imbalance is not very prominent [21], [47]; it can thus be resolved via a centralized task queue [18]. However, the huge real-word graphs for GCNs often follow the Power-Law distribution (see Figure 1), meaning that a small set of rows/columns can have the majority of the non-zeros. This can lead to serious runtime workload balancing scenarios for which existing SCNN accelerators were not designed. Due to these reasons, novel accelerator designs are required for the emerging GCN workloads.

We thus propose UWB-GCN, a hardware accelerator for GCN inference with runtime workload rebalancing. The processing begins by using an online profiler to accurately assess

[1]According to [26], stacking multiple GCN layers leads to over-smoothing and accuracy degradation, i.e., all nodes converge to the same value.

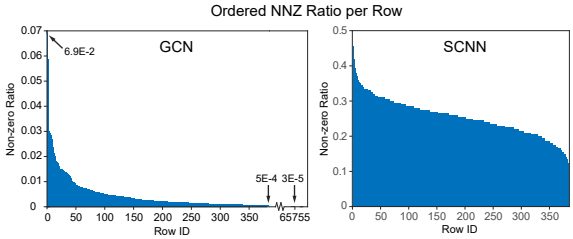

Fig. 1. Ordered `nnz` density-per-row histogram for GCN and SCNN. (1) Adjacency matrix of the NELL graph for GCN, following the power-law with most `nnz` clustered in 100 out of 65755 rows. (2) Unstructured compressed AlexNet weight matrix for SCNN, which is roughly balanced across 384 rows.

the workload imbalance. The imbalance information is used by two strategies for rebalancing the workload: *local sharing* and *remote switching*. Local sharing rebalances the workload among neighbors. However, given only local sharing, when elements are clustered, it may take many iterations for the autotuner to converge. We solve this problem with *remote switching*, which shuffles larger regions. Figure 2 illustrates the autotuning process among 10 PEs. The color of the PEs indicates their utilization. *Utilization*=100% means the workload processed by this PE is the same as if the entire workload is evenly distributed among all PEs. *Utilization*$<$ 100% (red shift) and *Utilization*$>$ 100% (blue shift) indicate over-utilization and under-utilization, respectively.

The goal is to balance the workload among all PEs. As can be seen in Figure 2 (starting top left and moving anticlockwise), in each autotuning iteration, we first perform remote switching to balance the workload of large regions (top left to bottom left) and then we perform local sharing for fine-grained tuning among neighbors (bottom left to bottom right). We again measure the workload distribution and, if necessary, start a new rebalancing iteration (bottom right to top right). After several iterations, the system converges to the ideal balanced state, which is then used for the remainder of the computation. All profiling and adjustment are performed by hardware at runtime with negligible area and delay overhead.

We implement UWB-GCN in Verilog-HDL and evaluate it on a Xilinx VCU-118 FPGA. This is only for demonstration purposes since UWB-GCN does not rely on any FPGA-specific features. The results show that UWB-GCN can enhance the average PE utilization from 60% to 92% as compared to a baseline design without workload balancing. Overall, this paper makes the following contributions:

- We propose the first hardware accelerator for graph convolutional network inference.
- To handle the ultra workload imbalance issue, we propose a hardware-based workload distribution autotuning framework, which includes an efficient workload profiling technique and two workload rebalancing strategies.
- Evaluations on an FPGA show that the autotuning framework can quickly converge to the optimal workload distribution and thus achieve significant performance improvement. Compared with CPU (Intel Xeon-E2698v4 + MKL-v2018), GPU (NVIDIA Tesla-P100 + cuSparse-v10.0), and a baseline accelerator without workload rebalancing,

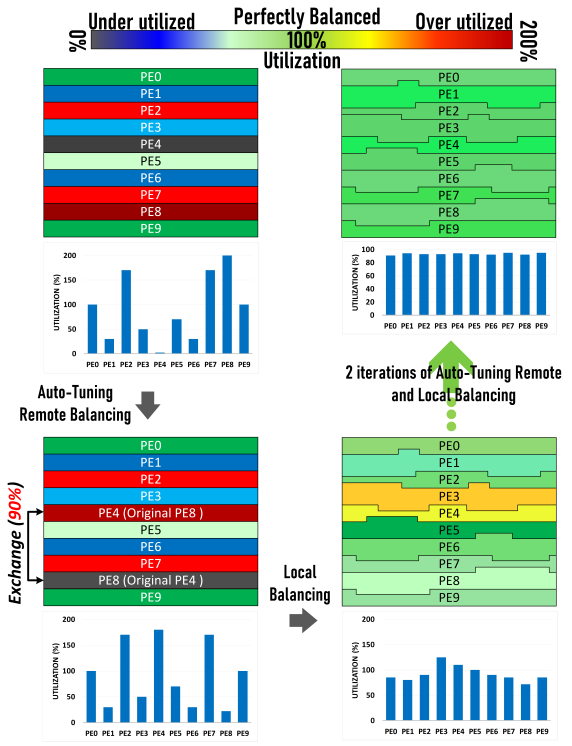

Fig. 2. Neighbor PE workload sharing and remote PE workload switching.

UWB-GCN achieves average speedups of 248.2×, 79.0×, and 2.8×, respectively.

II. Motivation

In this section we briefly introduce GCNs, showing their differences from traditional network models like CNNs and the challenges on hardware design arising from these differences.

A. Graph Convolutional Network Structure

Equation 1 shows the layer-wise forward propagation of a multi-layer spectral GCN [22], [37]:

$$X^{(l+1)} = \sigma(AX^{(l)}W^{(l)}) \quad (1)$$

A is the graph adjacency matrix with each row delineating the connection of a vertex with all the other vertices in the graph. $X^{(l)}$ is the matrix of input features in layer-l; each column of X represents a feature while each row denotes a node. W^l is the weight matrix of layer-l. $\sigma(.)$ denotes the non-linear activation function such as *ReLU* [23]. In general, A needs to be normalized: $\tilde{A} = D^{-\frac{1}{2}} \times (A+I) \times D^{-\frac{1}{2}}$ where I is the identity matrix, and $D_{ii} = \sum A_{ij}$. The reason is that, without normalization, multiplying the feature vector $X^{(l)}$ by A will change its scale: those nodes with more neighbors tend to have larger values under feature extraction. Note that during both training and inference of GCN, $\tilde{A}$ remains constant. Since $\tilde{A}$ can be computed offline from A, in the remainder of this paper we use A to denote the normalized $\tilde{A}$. In general, A is multiplied only once per layer. However, when multi-hop neighboring information is to be collected, A can be multiplied twice or more (i.e., A^2, A^3, etc.) per layer.

Equation 1 is derived from graph signal processing theory: convolutions on a graph can be converted to a multiplication of

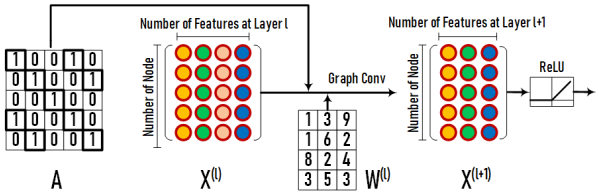

Fig. 3. The structure of a single layer in GCN.

signal $x \in R^N$ (i.e., a scalar for each node) and a filter $g \in R^N$ in the frequency domain via the Fourier transform:

$$CONV(g,x) = \mathscr{F}^{-1}(\mathscr{F}(x) \odot \mathscr{F}(w)) = U(U^T x \odot U^T g) \quad (2)$$

where $\odot$ denotes the Hadamard product. U is a collection of eigenvectors for the normalized graph Laplacian $\mathscr{L} = I_N - D^{-\frac{1}{2}}AD^{-\frac{1}{2}} = U\Lambda U$. The diagonal matrix Λ comprises the eigenvalues. If a frequency domain filter $g_W = diag(W)$ is defined, then Equation 2 can be simplified [7] as:

$$CONV(g_W,x) = Ug_W U^T x \quad (3)$$

Equation 3 can be further simplified by defining the filter as the Chebyshev polynomials of the diagonal matrix Λ [11], [22] to obtain Equation 1.

Figure 3 illustrates the structure of a GCN layer. By multiplying A and $X^{(l)}$, information from 1-hop connected neighboring nodes are integrated. By multiplying $AX^{(l)}$ with $W^{(l)}$, and going through the non-linear activation function $\sigma(.)$, we obtain the output of this layer, which is also the feature matrix for the next layer $X^{(l+1)}$. The matrix A in different layers can be the same (e.g., normal graphs) or different (e.g., hypergraphs). After multiple layers, the GCN is able to extract very high-level abstracted features for various learning purposes.

B. Workload Imbalance from Power-Law Graphs

Real-word graphs typically follow the *power-law* distribution, which states that the number of nodes y of a given degree x is in proportional to $x^{-\beta}$ for some constant $\beta > 0$. This implies that in the adjacency matrix A, a small number of the rows (or columns) include the majority of non-zeros whereas the majority of the rows (or columns) are almost empty. Figure 4 shows the distribution of *non-zero* elements for the five publicly available datasets that are widely used for GCN evaluation [22]. The *power-law* effect is quite prominent for `Cora`, `Citeseer`, and `Nell`.

Matrix X is also sparse. For the first layer, the sparsity is usually larger than 90%. This is because, in a large graph (e.g., with millions of vertices), many of the features are local features. Therefore, the entries in A corresponding to these local features for remote nodes are zero (explaining the sparsity). As the weight matrix W is usually dense, the output of AXW is also dense. However, because of the *ReLU* activation function, more zeros are generated; thus the final output (also the input of the next layer) is sparse but with sparsity usually less than 50%.

The sizes of the matrices in GCNs depend on the dataset and can range from thousands to millions or more. Therefore, A can be extremely large and is stored in a sparse format. A in different layers can be identical or distinct (e.g., evolving

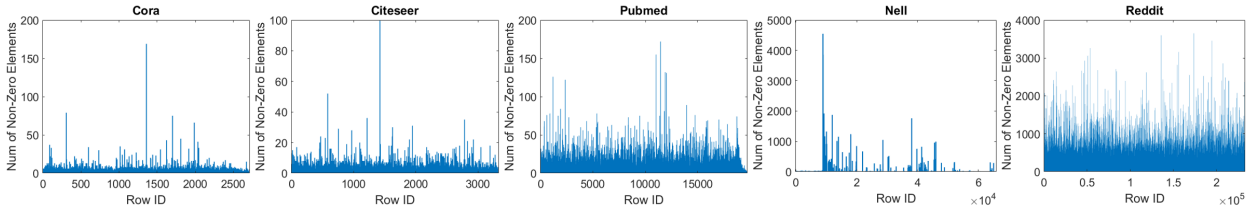

Fig. 4. Non-zero distribution imbalance of Adjacency matrices in Cora, Citeseer, Pubmed, Nell and Reddit datasets

TABLE II
OPERATIONS REQUIRED UNDER DIFFERENT EXECUTION ORDERS

Layer	Order	CORA	CITESEER	PUBMED	NELL	REDDIT
Layer1	$(A\times X)\times W$	62.3M	197.5M	163.2M	257G	16.3G
	$A\times(X\times W)$	999.7K	1.87M	17.5M	47M	6.1G
Layer2	$(A\times X)\times W$	468.2K	493.0K	2.3M	800M	764.3M
	$A\times(X\times W)$	329.3K	357.6K	1.06M	735M	530.3M
ALL	$(A\times X)\times W$	62.8M	198.0M	165.5M	258G	17.1G
	$A\times(X\times W)$	1.33M	2.23M	18.6M	782M	6.6G

or samples from the same graph). X is very large for the first layer. However, in contrast with CNNs where the number of features per layer is roughly similar or increasing, the number of features in GCNs often reduces drastically by layer. It is possible for there to be thousands of features in the first layer, but only a few dozen in the second. X is stored in a sparse format and the output matrix after *ReLU* is also sparse.

III. GCN Baseline Architecture

In this section we describe an initial architecture for GCN acceleration. This "baseline" design shares some similarities with existing sparse CNN accelerator designs, but in addition it can support ultra-high sparsity and large matrix sizes. In the next section we augment this design to achieve near-optimal workload balancing.

A. Matrix Computation Order

To compute AXW, there are two alternative computation orders: $(A\times X)\times W$ and $A\times(X\times W)$. The choice is significant as it dictates the volume of non-zero multiplications. Based on our profiling, A is ultra sparse and large, X is generally sparse and usually has a large number of columns, and W is small and dense. Looking first at $(A\times X)\times W$: since multiplying A and X requires complex sparse-sparse-matrix-multiplication and produces a very large dense matrix, multiplying by another dense matrix W leads to significant computation workload and long delay. Alternatively, for $A\times(X\times W)$, both are sparse-dense matrix multiplications (SpMM) and the scale of computation is drastically smaller. Table II lists the amount of computation for the five datasets following the two approaches. Since the difference is quite obvious, in our design we first perform $X\times W$ and then left multiply with A.

B. Baseline SpMM Design

Given $A\times B=C$, if A is $(m\times n)$, B is $(n\times k)$, and C is $(m\times k)$, then we can reformulate C as:

$$C=[\sum_{j=0}^{n-1}A_j b_{(j,0)},\sum_{j=0}^{n-1}A_j b_{(j,1)},\ldots,\sum_{j=0}^{n-1}A_j b_{(j,k-1)}] \quad (4)$$

where A_j is the jth column of A and $b_{j,k}$ is an element of B at row-j and column-k. In other words, by broadcasting

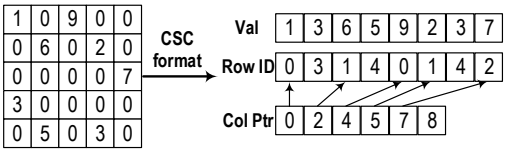

Fig. 5. Compressed-Sparse-Column sparse matrix format.

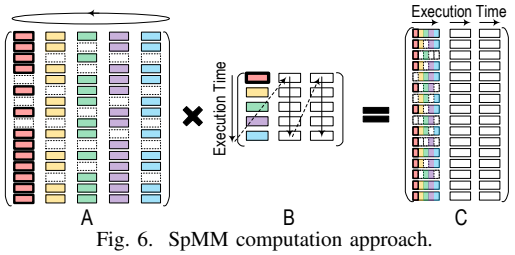

Fig. 6. SpMM computation approach.

the jth element from column-k of B to the entire column-j of A, we can obtain a partial column of C. Essentially, B is processed in a streaming fashion: each element $b_{(j,k)}$ finishes all computation it involves at once, and then is done. In this way, we reuse the entire sparse matrix A for each column of C (k times in total). Such a design brings additional advantages when A and C are stored in *Compressed-Sparse-Column* (CSC) format (see Figure 5). Further benefit is that it provides opportunities to pipeline multiple SpMM operations, as will be discussed later. Since a complete result element of C requires an entire corresponding row of A, to avoid expensive parallel reduction in hardware, we partition A and C along the rows and assign them to PEs. Figure 6 shows the procedure for calculating C. The columns of A and elements of B in the same color are to be multiplied and stored as partial results in C with the same color. To reduce the memory access demand for matrix A, we propose an inter-layer interleaved data forwarding technique (discussed in Section III.D).

Workload Mapping: In the baseline design, with the assumption that non-zeros are evenly distributed among the rows, we use a direct and static mapping from matrix rows to PEs. In Figure 7 every two rows of A are mapped to a separated PE; each PE eventually processes three non-zeros of A.

C. Baseline Architecture Design

Figure 8 illustrates the baseline design, comprising the modules of *sparse-matrix-memory* (SpMMeM), *dense-column-memory* (DCM), *task-distributor & Queue* (TDQ), *PE-array*, and an *accumulation-buffers-array* (ACC). SpMMeM buffers the input sparse matrix A. DCM buffers the input dense matrix B. TDQ is for task distribution to the PEs. PE-array is for concurrent multiplication. Finally, ACC buffers the

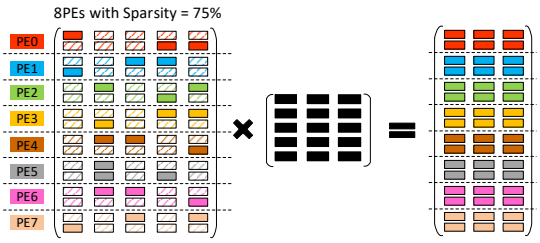

Fig. 7. Partitioning the sparse matrix rows among 8 PEs.

partial results of the resulting matrix C for accumulation. Depending on the sparsity and storage format of A, we have two alternative designs for TDQ:

TDQ-1 (left side of Figure 8) is used when A is generally sparse and stored in dense format. We perform the direct row partition as discussed and map to the input buffer of a PE (see Figure 7). Each cycle, $NPE/(1-Sparsity)$ data are forwarded to a PE given evenly distributed non-zeros. As one PE may account for more than a single row of A, we allocate multiple Task Queues (TQs) per PE. As shown in Figure 8-(left), in each cycle a PE can receive up to 4 non-zero elements. We have four queues to buffer these non-zeros from different rows of A. Each cycle, an arbitrator selects a non-empty queue, pops an element, checks for a Read-after-Write (RaW) hazard (discussed later), and forwards it to the PE for processing.

TDQ-2 (right side of Figure 8) is used when A is ultra-sparse and stored in CSC format. Since in CSC the non-zeros are contiguous in a dense array, if we can directly process the dense array, we gain from avoiding all the zeros. However, we suffer from the overhead of navigating to the correct PE as the indices are no longer continuous and (essentially) stored in another index array. We use a multi-stage Omega-network for routing the non-zero data stream to the correct PE according to their row indices from the index array. Each router in the *Omega-network* has a local buffer in case the buffer of the next stage is saturated. Our design attempts to balance the data forwarding rate and the processing capability of the PEs. This is achieved when non-zero elements are distributed evenly among rows. Compared with a global crossbar network, the Omega-network design incurs much less area and hardware complexity; this is especially the case when we have a large number of PEs. Meanwhile, TDQ also accepts streaming data from a particular column of the dense matrix B in DCM.

PEs fetch partial results of C from ACC, perform the new multiplication task, add to the partial results, and save back to ACC. Each PE is coupled with a bank of ACC to store the rows of C it accounts for. A PE has two units: a *multiply-accumulate-unit* (MAC), and an *address-generation-unit* (AGU) for result address generation and forwarding. Since C is roughly a dense matrix and stored in dense format, the rows of C are statically partitioned among ACC buffers. Synchronization is only needed when an entire column of the resulting matrix C is completely calculated. Consequently, the imbalanced distribution of non-zeros across columns does not cause any performance problems.

An important issue here is the *Read-after-Write* (RaW) hazard. Since the computations are all floating-point, the pipelined MAC unit usually takes several cycles to process, but can still accept new tasks while processing. If the new task tries to accumulate the same partial result of C (i.e., from the same row of A), it actually fetches a stale partial result from ACC, and a RaW hazard occurs. To avoid this hazard, we implement a stall buffer of size T, where T is the delay of the MAC units. We track the row indices currently being processed by the MAC and check whether the current element is targeting the same row in the *RaW-check-unit* (see Figure 8). If so, we buffer that job and delay (for a few cycles) until the hazard is resolved. This is similar to the role of the scoreboard for register RaW hazards in processor design.

Overall, for each layer of GCN, we first execute SpMM on $X \times W$. Since X is generally sparse and stored in dense format, we use TDQ-1. The result of XW is dense. We then compute $A \times (XW)$ which again is SpMM. However, as A is ultra-sparse and stored in CSC format, we use TDQ-2. The result is dense, but after *ReLU*, a large fraction of the entries become zero and we again have a sparse matrix as the input feature matrix for the next layer.

D. Pipelining SpMM Chains

Intra-Layer SpMM Pipelining: One can exploit the parallelism between consecutive SpMMs (i.e., $X \times W$ and $A \times (XW)$) in a layer. This is based on the observation that when a column of (XW) has finished computing, and A is constant and ready, we can start the multiplication of A with that column without waiting for the entire XW (Figure 9). This design has two major benefits: (i) we gain extra parallelism and reduce the overall delay through coarse-grained pipelining, and (ii) instead of needing large off-chip storage to cache the resulting XW matrix, we only need to buffer a single column of XW; this can be done on-chip. This method can be reused within a GCN layer if left-multiplied by other sparse matrices. For example, some GCNs collect information from 2-hop neighbors so the layer formulation becomes $A \times (A \times (X \times W))$ and the three multiplications can be pipelined.

Inter-Layer SpMM Pipelining: SpMMs from different layers can also be pipelined. To avoid bubbles and large intermediate buffers, we allocate hardware resources (i.e., number of PEs) in proportion to the workload of each SpMM stage (see Figure 10). In this way the output generation rate of the previous SpMM matches the data consumption rate of the current SpMM. Pipelining SpMMs from different layers has three benefits: (i) being able to exploit inter-layer parallelism; (ii) no off-chip memory access is required on the intermediate result matrix X since data are processed by SpMM engines in a streaming manner; and (iii) if A is the same for all layers, it can be reused by SpMM engines across the layers thus avoiding extra off-chip accesses. This is done by forwarding elements of A through the layers. If the access rates to A are different across the layers, we forward elements of A with interleaving to match the ratio of the access rates.

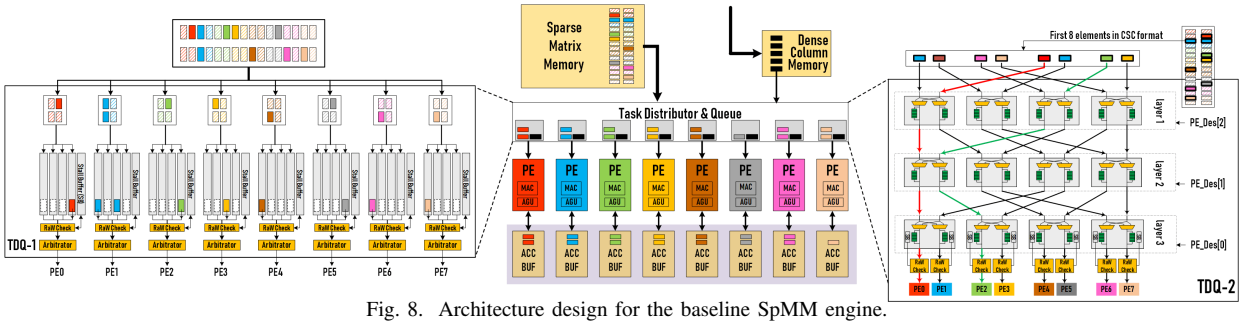

Fig. 8. Architecture design for the baseline SpMM engine.

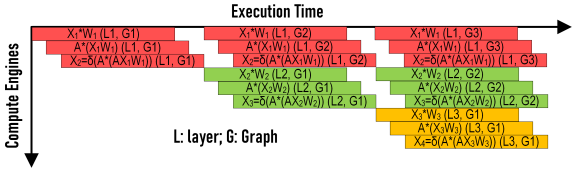

Fig. 9. Exploiting extra parallelism across consecutive SpMM computation through pipelining.

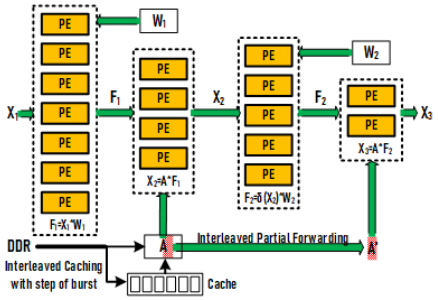

Fig. 10. Pipelining SpMM chains. Data production and consumption rates are balanced among consecutive SpMMs.

Bandwidth Analysis: Off-chip data access of the big adjacency matrix A can be a concern. However, as the access to A is always continuous in our design, the performance benefits greatly from the DRAM burst mode. If A can fit in the on-chip memory, we reuse A across layers. Otherwise, we cache part of A on-chip to mitigate the pressure on the off-chip bus. Based on our evaluation, the UWB-GCN accelerator requires around 227 Gbps off-chip bandwidth to keep the hardware busy with 1024 PEs for the 5 datasets evaluated, which can be generally satisfied by modern computation platforms (e.g., Xilinx VCU-118 FPGA provides 384 Gbps off-chip bandwidth, VCU-128 provides 3680 Gbps HBM bandwidth, NVIDIA V100 provides 7176 Gbps HBM bandwidth).

E. The Workload Balance Problem

The baseline architecture works well when non-zeros are evenly distributed among the rows of A. However, when this assumption does not hold, the performance of the baseline architecture can degrade considerably due to workload imbalance among PEs and network congestion. Figures 11-(A) and (B) illustrate two types of workload imbalance, *local* and *remote*, and also the histogram from mapping to eight PEs. Note that both types of imbalance lead to significant performance degradation, from the expected 2 cycles to 5 cycles and 7 cycles, respectively.

This imbalance issue is unique for GCNs and has not been faced or resolved by existing work such as with sparse-CNNs

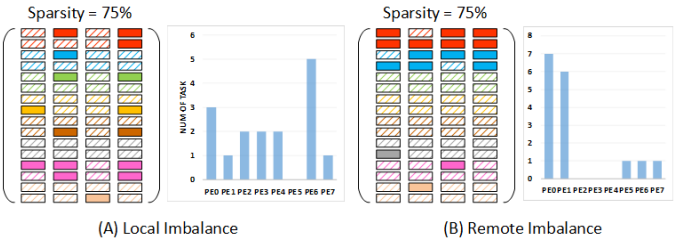

Fig. 11. Local and remote workload imbalance among 8 PEs with 75% sparsity.

[2], [18], [21], [45]. The reason is that non-zeros in those sparse matrices are more or less evenly distributed. However, when dealing with huge and ultra sparse matrices such as the adjacency matrix of a social-network graph following a power-law distribution, the condition is quite different. Efficiently handling of this unique workload balance problem from this new GCN application is the major research problem for this work. Typically, when dealing with sparse data structures such as sparse matrices/tensors, trees and graphs, etc., to achieve workload balance, the software approach is to profile or scan the structure through, for example, symbolic analysis, in a preprocessing stage, and then use the sampled information to guide the partition strategy later for real processing. In this work, we show how to dynamically adjust hardware configuration for continuous workload rebalancing. Our design can be applied to a variety of specialized accelerators for processing sparse data structures.

IV. UWB-GCN Architecture

We treat the two types of imbalance problems (shown in Figure 11) separately. For local imbalance, we propose *dynamic local sharing*; for remote imbalance, we propose *dynamic remote switching*. Both are dynamic techniques that measure and adjust for a better task distribution configuration each time a column of the dense input matrix is processed. After several columns, the optimal configuration for best matching the non-zero structure of the sparse input matrix is obtained. This configuration is then reused for the processing of the remaining columns of the dense matrix.

The difference is granularity. As described in Section I, Figure 2 shows the design flow. Initially, we employ equal partitioning of the baseline design. Some PEs are over-utilized while others are under-utilized. The ultimate purpose of the design is to balance the colors (i.e., utilization) by adjusting or exchanging the workloads of PEs. At the processing of every column of matrix B (called *around*), we employ local

balancing by averaging out some of the overloaded work to neighbors, improving the situation. However, the offloaded work needs to be returned for aggregation after processing. The architecture is able to track the runtime utilization information of PEs with local balancing at a current iteration. Due to chip area and design complexity restrictions, we may exchange workload between direct neighbors, 2-hop neighbors, or even 3-hop neighbors, but not all of them. The local strategy is not efficient enough when non-zeros are clustered in a region across several PEs.

To address the non-zero clustering issues, we propose remote PE switching, making remote workload (partially or completely) exchange between overloaded PEs and underloaded PEs. At the end of each round, the utilization information tracked after local workload sharing is analyzed by specific hardware and used to optimize the remote switch strategy. By interchanging workloads between remote over-utilized and under-utilized PEs, followed by another round of local sharing, we can significantly improve load balancing. Note, the local and remote rebalancing occur in the processing of every column of the dense matrix B. As the same sparse matrix A is reused during the processing of every column of matrix B, the remote switch strategy generated in prior rounds is valuable and can guide the processing of the later rounds. Our accelerator remembers the remote switch plan at the end of each round and incrementally adjust it when processing every next column based on the newly obtained utilization information from local balancing. After several rounds, the configuration best matching the sparse structure of A is obtained, and we use it for the remaining rounds with almost perfect workload balancing. In the following, we discuss how to realize this strategy in hardware.

A. Dynamic Local Sharing

PE utilization differences must be estimated before the workload can be adjusted. Figure 12 illustrates 1-hop local sharing for TDQ-1 and TDQ-2.

TDQ-1: Before a new task is pushed into a PE's TQ, the PE compares the number of pending tasks with those in the neighboring TQs. The task is then forwarded to the TQ with the fewest pending tasks. If forwarded to a neighbor, the result needs to be returned to the ACC buffer of its original PE for accumulation after the multiplication (see Figure 12-(B)). The valid return address is calculated in the AGU unit of a PE.

TDQ-2: For the Omega network, the final layer of the multi-stage network handles forwarding. In Figure 12-(C), two PEs share the same final-layer switch; we refer to this pair as a *group*. In Figure 13, a group has four PEs sharing the same final-layer switch so we focus on the TQs of the final layer. After determining the pending task status, the id of the destination PE is known. We adjust the address tag of the task before it is pushed to the TQs of the final layer. To enable PEs on the group edge (i.e. the leftmost or rightmost PEs) to communicate with their out-of-the-group neighbors, we add extra links in the final layer, as shown in Figure 12-(D). Note

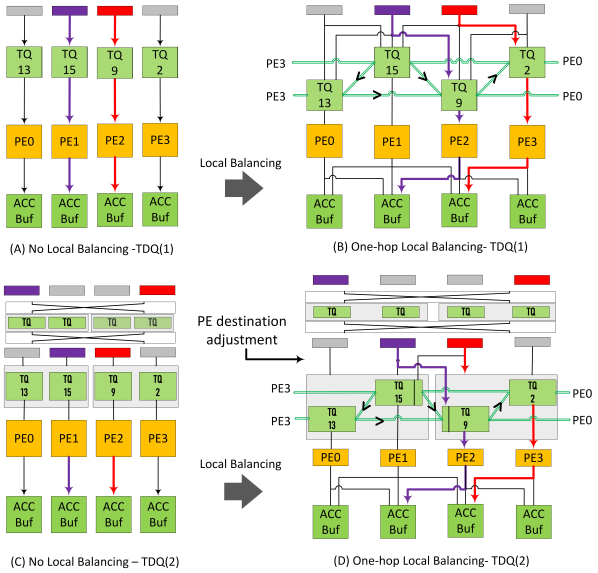

Fig. 12. Architecture design for local workload sharing.

that Figure 12-(D) shows sharing only among 1-hop neighbors. By considering more distant hop neighbors, a more balanced design is obtained at the cost of higher hardware complexity and area. This is discussed in the evaluation section.

B. Dynamic Remote Switching

For remote switching, a multi-round autotuning approach is again used. The idea is to find the most over-utilized PE and the most under-utilized PE per round (i.e., for a column of B), and switch a part, or all of their workloads. The fraction to be switched depends on their utilization gap. The design is shown in Figure 13.

The most over-utilized and under-utilized PEs are identified by using the *PE Status Monitor* (PESM). Recall that each TQ has a counter to track the number of pending tasks; these can trigger an *empty* signal when reaching zero. The counters are all connected to a multi-stage MUX-tree with output signal Ψ. After the jobs of the current round are dispatched Ψ monitoring begins. When Ψ triggers, some PEs have become idle. By voting at each level, the MUX-tree is able to identify the PE group with the highest number of empty signals, i.e., the coldspot. When all PEs have triggered the empty signal, the last to finish is the hotspot.

Having identified hotspot and coldspot PE-tuples with id T_j for the current round i, to avoid thrashing, we only exchange a portion of the workload between them. We propose the following formula to calculate the number of jobs (i.e., rows of A) to be switched in the i-th round (i.e., a column of B) N_i:

$$N_i = \begin{cases} 0 & if\ i = 1 \\ N_{i-1} + G_i/G_1 \times (R/2) & otherwise \end{cases} \quad (5)$$

where G_i is the largest workload gap (i.e., the workload difference between hot-spot and cold-spot PEs) for the i-th round, and R is the initial workload under equal partition. In the current design, each T_j tuple is tracked for two rounds using the PE Status Monitor in Figure 13. Each PE-tuple being tracked is updated every round according to Equation 5. The

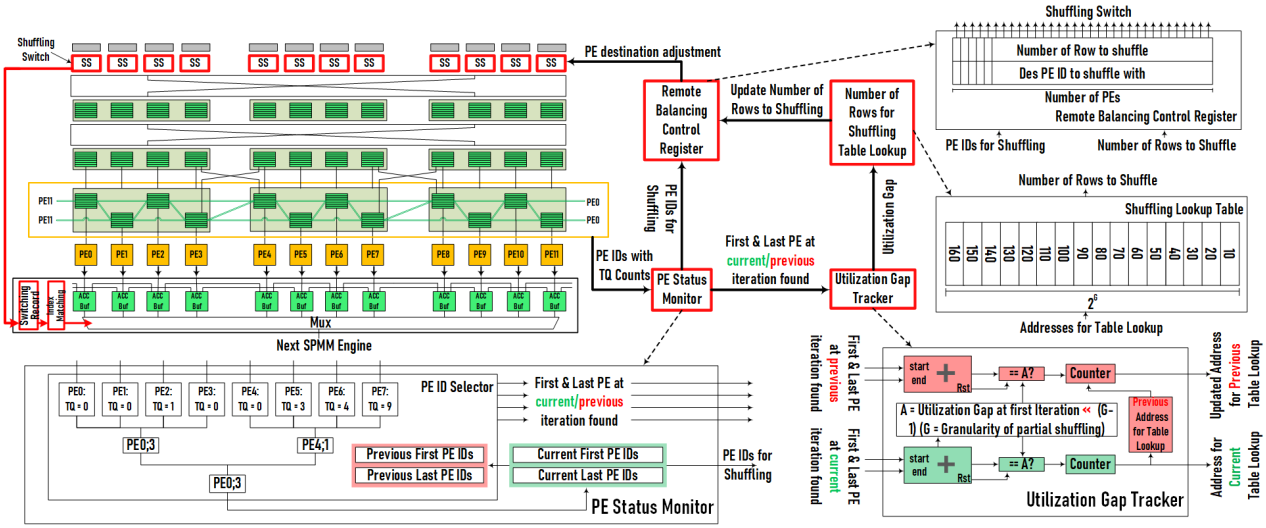

Fig. 13. Overall SpMM engine architecture in UWB-GCN with local sharing and remote switching. Modules with red border are for remote switching.

number of rounds tracked simultaneously can be customized. In Figure 13, two consecutive rounds are tracked.

The workload switching ratio for each tracked PE-tuple is then adjusted for two or more rounds and is highly likely to converge. The number of rounds we can track depends on the size of the tracking window in the PESM and is an area/performance tradeoff. Calculating Equation 5 is done in the *Utilization Gap Tracker* (Figure 13). To reduce the hardware cost of calculating $G_i/G_1 \times (R/2)$ we use a hardware-efficient approximation; details are omitted due to space limitations. Once the number of rows to be switched between remote PEs is known, the *Shuffle Lookup Table* (SLT) is used to determine which rows are to be interchanged. The IDs of these rows are forwarded to the *Remote Balancing Control Register* (RBCR). In the next round, the destination PE of these rows is updated in the *Shuffle Switches* (SS).

For routing data from PEs to ACC buffers note that remote switching does not require an extra interconnect. The data at the rows selected to be switched do not need to be routed back to the original ACC. Instead, at the end of each iteration, the SpMM engines access the data cached in ACC of the previous SpMM engines (through a MUX set based on the remote switch decision). The overhead is low: storage for a copy of the switch record and a comparator for the index check.

V. Evaluation

We evaluate the baseline and UWB-GCN designs and compare them with the same GCN networks on other platforms.

A. Evaluation Configuration

We implement the RTL of the baseline and UWB-GCN in Verilog HDL. We measure the PE utilization, performance, energy efficiency, and hardware resource consumption on a Xilinx Virtex UltraScale+ VCU118 FPGA board. Note that the FPGA is only used as an evaluation platform to demonstrate the performance of UWB-GCN. The design is a general architecture that does not leverage any FPGA-specific features or units.

To measure utilization we add a counter to each PE to track the number of idle cycles. The number of operating cycles (i.e., execution delay) is measured with a cycle-accurate hardware counter. The counter triggers when the first data is forwarded to UWB-GCN and stops when the last output feature is received. The hardware consumption and operating frequency are reported by Vivado Design Suite-2019.1 after synthesis and implementation. The board-level power consumption is measured by a power meter. To perform the cross-platform comparison, we implement the reference GCN networks in *PyTorch* and run them on a high-end server-class CPU Intel Xeon E5-2698-V4 and an NVIDIA Tesla P100 GPU. For the SpMM computation PyTorch uses the MKL-v2018 and cuSPARSE-v10.0 libraries. We also adapt two existing SCNN designs EIE [18] and Cambricon-S [47] for the GCN workload and make a comparison. All latency results reported in our paper are end-to-end. Data access from DDR/GDDR/HBM of CPU/GPU/FPGA is included.

The datasets used for evaluation are *Cora*, *Citeseer*, *Pubmed*, *Nell* and *Reddit*; these are the five most widely used publicly available datasets in GCN research.

B. UWB-GCN Evaluation

The efficiency of the design is evaluated by comparing the performance, hardware resource consumption, and PE utilization of the baseline design (i.e., *Base*) with the four different design choices of UWB-GCNs: (i) 1-hop local sharing (i.e., *Design(A)*), (ii) 2-hop local sharing (i.e., *Design(B)*), (iii) 1-hop local sharing plus remote switching (i.e., *Design(C)*), and (iv) 2-hop local sharing plus remote switching (i.e., *Design(D)*). The only exception is for *Nell* where we use 2-hop and 3-hop local sharing; reasons are given below.

Figure 14 compares the overall GCN inference delay, including SpMM kernels and other operations (e.g. ReLU), and average utilization of PEs for the five designs over the five datasets. The lines show the overall PE utilization. The bars show the break-down of delay cycles according to GCN layer. We also mark the latency lower bound assuming full

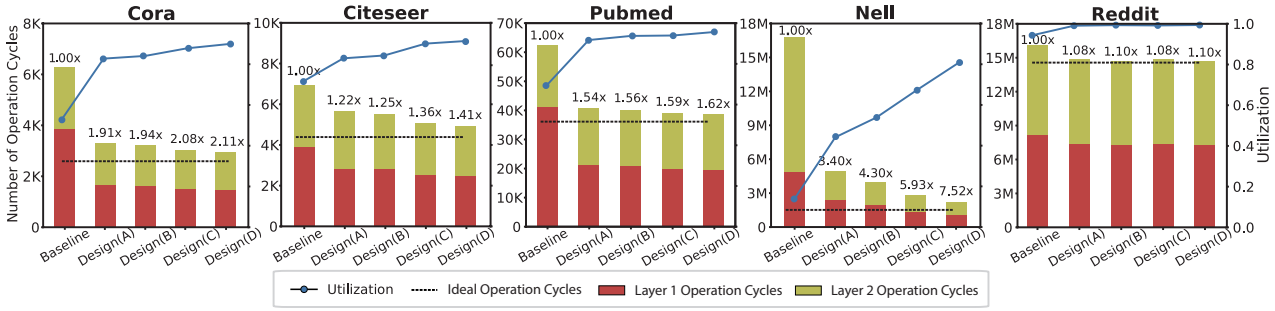

Fig. 14. Overall performance and PE utilization of the proposed five designs.

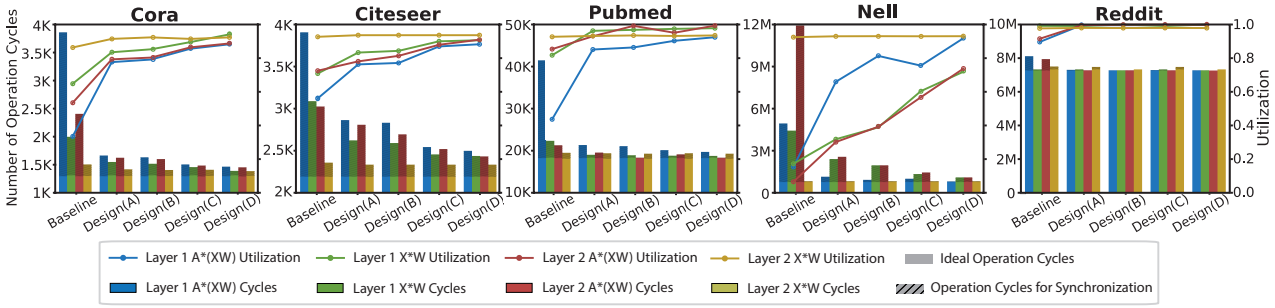

Fig. 15. Per-SpMM performance and PE utilization of the proposed five designs.

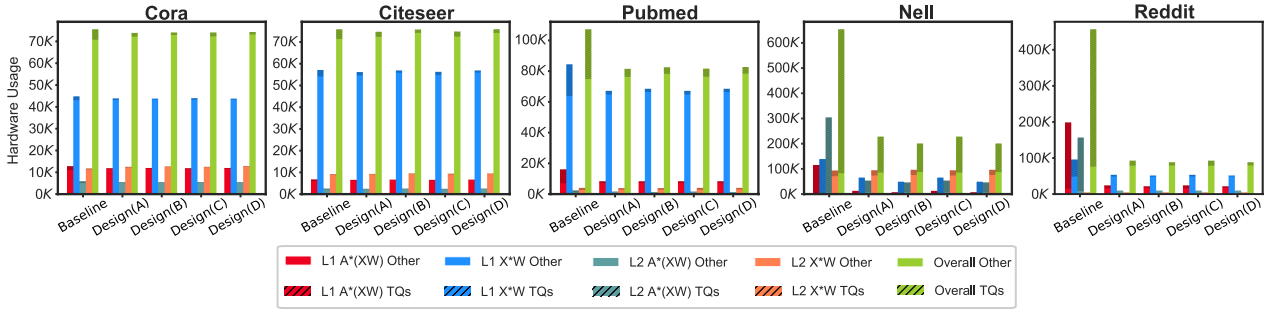

Fig. 16. Hardware resource consumption normalized to the number of CLBs of the five designs.

PE utilization. For *Cora*, *Citeseer* and *Pubmed*, using 2-hop local sharing can improve PE utilization from 53%, 71%, and 69%, to 83%, 83%, and 93%, respectively, leading to 1.94×, 1.25×, and 1.56× performance improvement. Enabling remote switching can further improve PE utilization to 90%, 91%, and 96%, respectively, bringing performance gain to 2.11×, 1.41×, and 1.62×.

After analysis, we found that the remaining 4-10% utilization gap is due to PE under-utilization in the autotuning phase. For *Nell*, as shown in Figure 4, the non-zeros are highly clustered. In this case, one or two PEs are extremely over-utilized in the baseline design, leading to only 13% overall utilization. In this case, even 2-hop local sharing is still insufficient to rebalance the workload. Therefore, for the *Nell* dataset only, we use 2-hop and 3-hop local sharing (rather than 1-hop and 2-hop) in our evaluation. The results show that 2-hop and 3-hop local sharing can enhance PE utilization from 13% to 44%, and 53% respectively, resulting in 3.4× and 4.3× performance improvements. With remote switching enabled, the utilization further increases to 69% and 81%, leading to 5.9× and 7.5× performance gains. For *Reddit*, through local sharing, the utilization is already 99% (from 93% in the baseline). In UWB-GCN, hardware resources allocated to different layers are in proportion to their volume of operations. Thus, when perfect utilization is achieved, the same execution delay can be observed for all the layers. As shown in Figure 14, the green and red bars have similar length at Design(D), while their lengths vary significantly at Baseline.

Figure 15 further breaks down the numbers of operating cycles and shows results for each SpMM kernel; this demonstrates the benefits of UWB-GCN on kernels with various sparsity and imbalance distributions. As shown in Figure 15, the execution times of different SpMM kernels are almost the same so that SpMM engines are rarely idle while waiting for their neighbors. The shaded area of the bars represents the *Sync* cycles due to workload imbalance; the unshaded area represents the *Ideal* cycles assuming perfect workload balance. The bars in different colors represent the cycles of the four SpMM operations in the two-layer GCNs [22], [37]: $A \times (XW)$ *of Layer-1,* $X \times W$ *of Layer-1,* $A \times (XW)$ *of Layer-2*, and $X \times W$ *of Layer-2*. The curves show the corresponding PE utilizations.

Comparing the datasets: for *Cora*, *Citeseer*, and *Pubmed*, the imbalance occurs mainly in the $A \times (XW)$ SpMM operation of the input layer, which is significantly mitigated by the

rebalancing techniques of UWB-GCN. For *Nell*, the imbalance occurs mainly in the $A \times (XW)$ SpMM of the hidden layer; this imbalance is also diminished by UWB-GCN. *Reddit* by itself is already well-balanced. Comparing now the SpMM operations, the utilization improves significantly for $A \times (XW)$ of Layer-2, $X \times W$ of Layer-1, and $A \times (XW)$ of Layer-2. For $X \times W$ of Layer-2, although X is sparse after being filtered by the *ReLU* activation function of Layer-1, its sparsity is much lower than that of X in Layer-1; the utilization is thus also high for the baseline (except for *Cora*).

Figure 16 compares the overall hardware resource consumption of the five designs over the five datasets. The hardware resources cost is normalized to the number of *Configurable Logic Blocks* (CLBs) used in the design, which is the basic component of the FPGA. In an ASIC design, this can be normalized to the number of transistors. The red area represents the CLB consumption for the TQs of the TDQ modules. The argument is that, if the task distribution is more unbalanced, TQs require more slots for the buffering queue to avoid backpressure. Therefore, by introducing the rebalancing techniques of UWB-GCN, the area cost of TQs can be reduced. This is especially the case for *Pubmed*, *Nell*, and *Reddit*. The green area represents other hardware modules excluding the TQs. It remains almost unchanged across the five datasets, which means that the area overhead from the rebalancing logic of UWB-GCN is very small – only 2.1%, 3.9%, and 1.9% of the whole baseline-design area for 1-hop local-sharing, 2-hop local sharing, and remote switching designs, respectively; for *Nell*, it is 2-hop and 3-hop. Combining the two parts, the UWB-GCN design has reduced hardware resource consumption as compared with the baseline design. This is largely due to dramatically reduced per-PE TQ size under more balanced workloads. For example, for *Nell*, the required TQ depth in the baseline design for A×(XW) of Layer-1 is 65128; in Design(D) it is just 2675.

Finally, Figure 17 shows the utilization improvement due to iterative workload autotuning. Rebalancing can be accomplished within 10 iterations, which is around only 20% of the total iterations. This means that more than 80% of the iterations can benefit from operating under the converged optimal strategy.

C. Scalability of UWB-GCN

We evaluate the scalability of UWB-GCN by running GCN inference of the five datasets on the baseline as well as Designs (B) and (D) of UWB-GCN and varying the number of PEs from 512 to 768 to 1024. In Figure 18, the bars represent the end-to-end inference execution cycles and the lines represent the PE utilizations. The dotted lines mark the full utilization (100%) upper bound.

For the baseline design, the PE utilization drops with increasing number of PEs since more PEs means fewer rows per PE, highlighting the imbalance among PEs: they have fewer opportunities to absorb inter-row imbalance. In contrast, GCNs with both local sharing and remote switching show stable (and high) PE utilization. The PE utilizations with only local sharing scale better than the baseline, but worse than with both local sharing and remote switching. Overall, by introducing the rebalancing techniques, the performance of UWB-GCN scales almost linearly with the number of PEs, and much better than the baseline design.

D. Cross-platform Comparison

Table III presents the cross-platform evaluation of UWB-GCN. We compare inference latency (*milliseconds*), energy efficiency (normalized to *Graph Inference/kJ*), and operating frequency (*MHz*). The systems evaluated are UWB-GCN (Design (D) with 1024 PEs); GCN implementations of MKL-v2018-based Intel Xeon E5-2698V4 CPU and cuSPARSE-v10.0-based NVIDIA Tesla-P100 GPU; and the baseline UWB-GCN design (with 1024 PEs) without workload balancing.

We also evaluate a reproduced EIE reference implementation [18] tweaked and optimized for GCN processing. EIE conserves the sparse weight matrices on-chip, and distributively pre-loads the weight matrices to the local buffers of the PEs. Therefore, the original EIE design cannot directly apply to the GCN task here. To compare with EIE, we augment the original EIE design with the same off-chip access and task forwarding modules of UWB-GCN. The EIE results in Table III represents the performance that can be obtained by applying existing SCNN designs for GCN workload. The comparison demonstrate the effectiveness of workload rebalancing and the necessity of designing new GCN-specific accelerators.

We also attempted to reproduce Cambricon-S [47] for the GCN workload here, and make a comparison. However, this is not feasible because Cambricon-S adopts a 256-slot scheduling window for index matching. It is assumed that, by fetching 256 elements per cycle, at least 16 elements can be successfully matched in order to keep the PE busy. This may be true for CNN workload given the relatively smaller matrix size and low sparsity. However, for GCN workloads, to guarantee 16 elements can be forwarded to the PEs per cycle, we need a window that can process a million elements per cycle, which is obviously hardware unfeasible. Alternatively, we keep the window size as 256 and measure the PE utilization of Cambricon-S on GCN workloads. As a result, the extremely high sparsity and large dimension of the matrix A leads to merely 0.0013% PE utilization, once again revealing the necessity of a GCN-specific accelerator like UWB-GCN.

Table III shows that UWB-GCN has the best speedup (7.5×) over EIE with NELL, the most unbalanced dataset. Real-world graphs mostly follow the power-law; therefore CNN designs like EIE is not directly applicable (other CNN designs are discussed in Section VI). As we can see, despite running at a relatively low frequency, UWB-GCN achieves 248.2×, 79.0×, 2.8× and 2.7× speedups on average, over the high-end CPU, GPU, the baseline design without workload balancing, and the reference EIE design, respectively, across the five GCN graph datasets. Our design also achieves 870.9×, 618.0×, 2.6× and 2.6× better energy efficiency, respectively.

In Table IV, we compare the kernel-level utilization of UWB-GCN (Design (D)) with the cuSPARSE-based Tesla-

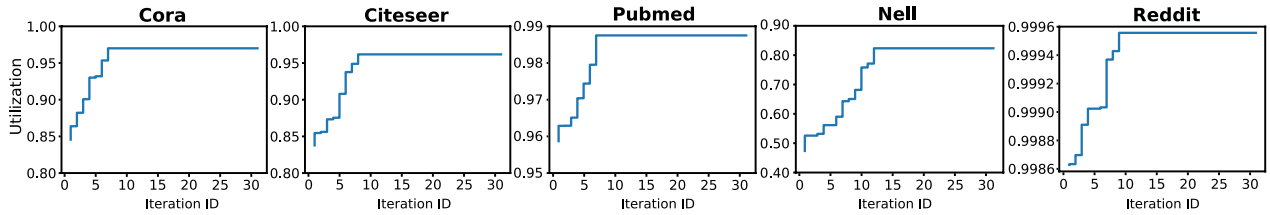

Fig. 17. Utilization improvement following the proposed iterative autotuning remote workload switching strategy.

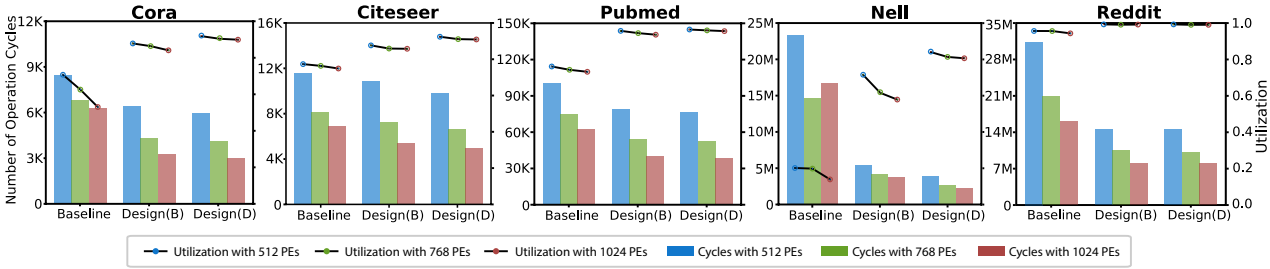

Fig. 18. PE Utilization and overall performance of Baseline, Design(B) and Design(D) of UWB-GCNs with different number of PEs.

TABLE III
CROSS-PLATFORM EVALUATIONS. LATENCY UNIT: *ms*; ENERGY EFFICIENCY UNIT: *graph_inference/kJ*

	Network	Cora	Citeseer	Pubmed	Nell	Reddit
Xeon E5-2698V4	Freq	2.2-3.6 GHz				
	Latency	3.90	4.33	34.15	1.61E3	1.08E4
	Energy	1.90E3	1.71E3	216.9	4.61	0.69
NVIDIA Tesla-P100	Freq	1328-1481 MHz				
	Latency	1.78	2.09	7.71	130.65	2.43E3
	Energy	1.87E3	1.59E3	432.3	25.51	1.37
EIE-like: VCU118 FPGA	Freq(MHz)	285 MHz				
	Latency	0.022	0.024	0.22	59.1	56.3
	Energy	1.19E6	1.11E6	1.20E5	438.2	452.1
Baseline: VCU118 FPGA	Freq(MHz)	275 MHz				
	Latency	0.023	0.025	0.23	61.0	58.9
	Energy	1.21E6	1.09E6	1.16E5	433.3	447.0
UWB-GCN: VCU118 FPGA	Freq(MHz)	275 MHz				
	Latency	0.011	0.018	0.14	8.1	53.2
	Energy	2.38E6	1.43E6	1.86E5	3.17E3	497.3

TABLE IV
SPMM-KERNEL-LEVEL UTILIZATION COMPARISON BETWEEN UWB-GCN DESIGN (D) AND CUSPARSE-BASED P100 GPU

		Cora	Citeseer	Pubmed	Nell	Reddit
UWB-GCN	XW1	0.93	0.90	0.98	0.72	0.99
	A(XW1)	0.87	0.88	0.93	0.92	0.99
	XW2	0.92	0.94	0.94	0.93	0.98
	A(XW2)	0.88	0.91	0.99	0.73	0.99
P100	XW1	0.10	0.11	0.18	0.19	0.18
	A(XW1)	0.03	0.03	0.18	0.39	0.50
	XW2	0.09	0.12	0.17	0.19	0.18
	A(XW2)	0.03	0.03	0.19	0.40	0.49

P100 GPU implementation with respect to the five datasets. UWB-GCN achieves much higher utilization than the GPU. The low utilization of GPU is mainly due to: (i) For small datasets such as *Cora* and *Citeseer*, the total workload is too small that they cannot saturate all the available SMs; (ii) For large datasets, the utilization is better, but still suffers from workload imbalance addressed by UWB-GCN.

E. Generalization of UWB-GCN

The arithmetic primitives in GNN can be abstracted and categorized as ***aggregation*** and ***embedding***, where the former multiplies the adjacency matrix by the feature matrix, and the latter multiplies the ***aggregated*** feature matrix with the weight matrix to shrink the feature vector length of each node. Various dataflow combinations of the two primitives compose multiple GNN designs, such as GraphSage [17], GINConv [39], and the latest GTN [44]. Nevertheless, the kernel of both primitives are SpMM. Given the fact that all these GNNs work on power-law based real-world graphs, and requires SpMMs for embedding and aggregation operations, they can all benefit from the design practice of UWB-GCN. With that, we claim that UWB-GCN is not for a single algorithm but a unified and versatile architecture for various GNN algorithms.

VI. RELATED WORK

Graph neural networks or GNNs use neural network-based approaches to address problems in graph domain. The first GNN model is proposed by Gori et al. [15]. In Gori's GNN, target nodes collect information of their neighbor nodes through recurrent neural structure until a converged status is achieved. In the past decade, researchers never stop optimizing the algorithms of GNNs and exploring new neural network approaches that can be useful for the tasks in graph domain [10], [30], [34], [37].

More recently, influenced by convolutional neural networks (CNN) that achieve great success in extracting local features from euclidean data such as images or videos, graph convolutional networks (GCNs) is proposed to address the feature extraction of non-euclidean data such as graphs by re-defining the convolution operators for graph calculation. In 2013, Bruna et al. [7] proposed a design of graph convolutional networks based on spectral graph theory; this was followed and further developed by a number of variants [11], [19], [22]. Meanwhile, other types of GNNs ,that are not based on spectral graph theory, have also been explored and proposed, including spatial-based GCNs [10], [13], graph attention networks [1], and graph generative networks [43]. Among different types of GNNs, spectral-based GCNs are at the center of the researches with regards to the neural network-based graph approaches.

To the best of our knowledge, the present work is the first accelerator design focusing on GCNs. There have been

many efforts on accelerating sparse CNNs [2], [18], [21], [24], [33], [45]. We briefly summarize these studies and explain why these solutions fall short when applied to GCNs. Kung et al. condense the sparse parameter matrix through column grouping [24]. In case of conflict, only the most significant parameters are kept, others are discarded. Essentially, some accuracy is traded-off for performance. Kim et al. [21] mitigate the workload imbalance problem of sparse CNNs, but use information from design-time profiling and pre-scanning. Han et al. [18] propose EIE, an SpMV accelerator that forwards non-zeros to PEs in column-major order; this is similar to our baseline design with TDQ-1. However, they only focus on SpMV and do not address the workload imbalance issue among the rows of the sparse matrix. Zhang et al. [45] rely on different indexing methods to identify and select non-zeros. However, these techniques do not function well when the matrix becomes ultra-sparse, as in GCNs. The reason these studies do not touch on the workload imbalance issue is partially because, compared with GCNs that process graphs, the impact of workload imbalance for sparse CNNs is much less significant. Chen et al. [8] propose Eyeriss. Rather than skipping zeros, Eyeriss saves energy by power-gating computations with zeros involved.

Besides a bunch of acceleration works on sparse CNNs, researchers also propose some architectures for general SpMM. Zhuo and Prasanna [48] present an SPMV design for FPGAs. They use the CSR format, which can be applied to various sparse matrices. However, this design still suffers from irregular sparse structures and the workload imbalance problem. Pal [32] proposes an outer-product-based SpMM architecture. This work focuses on reducing redundant memory access to non-zeros and does not essentially address the ultra-workload-imbalanced issue we are facing in GCNs. In their experiment results, load-imbalances during the merge phase and the uneven data sharing patterns during the multiply phase lead to degraded speedup for the dataset with highly-unbalanced non-zero element distribution.

Another active area of research about SpMM is software optimizations on general-purpose multicores and GPUs [3]–[5], [16], [28]. However, there are 3 reasons that solutions on general-purpose processors do not meet the strict timing requirement of GCN on latency. (1) significant overheads of pre-scanning: many recent new approaches compute the entire sparse matrix by first scanning the entire matrix, collecting the shape characteristics and classifying the sparse rows with different numbers of non-zero elements into different bins, and then, for different bins, different optimized kernels are launched [3], [16], [28]. UWB-GCN does not require any pre-scanning and profiling. (2) poor performance and low utilization with ultra-workload imbalance: all GPU software implementations (PyTorch, TensorFlow) rely on SpMM in cuSPARSE. However, cuSPARSE itself shows poor performance regarding such ultra-imbalanced conditions (as shown in Table IV). (3) some sparse matrices (feature matrices) in GCN are generated during processing and some matrices (adjacency matrices) are evolving at runtime, making any profiling and pre-scanning even less efficient.

VII. Conclusion

In this paper, we propose an architecture called Ultra Workload Balanced GCN to accelerate graph convolutional network inference. To tackle the major performance issues derived from workload imbalance, we propose dynamic local workload sharing and remote workload switching techniques. These rely on hardware flexibility to realize performance autotuning with negligible area and delay overhead. This is the first accelerator design for GCNs that relies on hardware autotuning to achieve workload rebalancing for sparse matrix computations. We create RTL designs and run experiments on a Xilinx VCU-118 FPGA with five widely used GCN graph datasets. These show that UWB-GCN can achieve, on average, 248.2×, 79.0×, and 2.8× speedups, respectively, over high-end CPUs and GPUs, and the baseline design without workload rebalancing.

References

[1] S. Abu-El-Haija, B. Perozzi, R. Al-Rfou, and A. A. Alemi, "Watch your step: Learning node embeddings via graph attention," in *Advances in Neural Information Processing Systems*, 2018, pp. 9180–9190.

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-neuron-free deep neural network computing," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 1–13, 2016.

[3] A. Ashari, N. Sedaghati, J. Eisenlohr, S. Parthasarathy, and P. Sadayappan, "Fast sparse matrix-vector multiplication on gpus for graph applications," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*. IEEE Press, 2014, pp. 781–792.

[4] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[5] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 2009, p. 18.

[6] R. v. d. Berg, T. N. Kipf, and M. Welling, "Graph convolutional matrix completion," *arXiv preprint arXiv:1706.02263*, 2017.

[7] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[8] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[9] C. W. Coley, W. Jin, L. Rogers, T. F. Jamison, T. S. Jaakkola, W. H. Green, R. Barzilay, and K. F. Jensen, "A graph-convolutional neural network model for the prediction of chemical reactivity," *Chemical science*, vol. 10, no. 2, pp. 370–377, 2019.

[10] H. Dai, Z. Kozareva, B. Dai, A. Smola, and L. Song, "Learning steady-states of iterative algorithms over graphs," in *International Conference on Machine Learning*, 2018, pp. 1114–1122.

[11] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Advances in neural information processing systems*, 2016, pp. 3844–3852.

[12] C. Ding, S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan *et al.*, "Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 395–408.

[13] H. Gao, Z. Wang, and S. Ji, "Large-scale learnable graph convolutional networks," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 1416–1424.

[14] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1263–1272.

[15] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, vol. 2. IEEE, 2005, pp. 729–734.

[16] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the csr storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, 2014, pp. 769–780.

[17] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in neural information processing systems*, 2017, pp. 1024–1034.

[18] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2016, pp. 243–254.

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

[20] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[21] D. Kim, J. Ahn, and S. Yoo, "A novel zero weight/activation-aware hardware architecture of convolutional neural network," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1462–1467.

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

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[24] H. Kung, B. McDanel, and S. Q. Zhang, "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 821–834.

[25] C.-E. Lee, Y. S. Shao, J.-F. Zhang, A. Parashar, J. Emer, S. W. Keckler, and Z. Zhang, "Stitch-x: An accelerator architecture for exploiting unstructured sparsity in deep neural networks," in *SysML Conference*, 2018.

[26] Q. Li, Z. Han, and X.-M. Wu, "Deeper insights into graph convolutional networks for semi-supervised learning," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[27] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[28] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. IEEE, 2014, pp. 370–381.

[29] Y. Liu, N. Zhang, D. Wu, A. Botterud, R. Yao, and C. Kang, "Guiding cascading failure search with interpretable graph convolutional network," *arXiv preprint arXiv:2001.11553*, 2020.

[30] A. Micheli, "Neural network for graphs: A contextual constructive approach," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 498–511, 2009.

[31] H.-T. Nguyen, Q.-D. Ngo, and V.-H. Le, "Iot botnet detection approach based on psi graph and dgcnn classifier," in *2018 IEEE International Conference on Information Communication and Signal Processing (ICICSP)*. IEEE, 2018, pp. 118–122.

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

[33] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 27–40.

[34] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[35] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[36] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[37] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *arXiv preprint arXiv:1901.00596*, 2019.

[38] T. Xie and J. C. Grossman, "Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties," *Physical review letters*, vol. 120, no. 14, p. 145301, 2018.

[39] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[40] C. Yang, A. Buluc, and J. D. Owens, "Graphblast: A high-performance linear algebra-based graph framework on the gpu," *arXiv preprint arXiv:1908.01407*, 2019.

[41] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2019, pp. 3165–3166.

[42] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 2018, pp. 974–983.

[43] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, "Graphrnn: A deep generative model for graphs," *arXiv preprint arXiv:1802.08773*, 2018.

[44] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 11 960–11 970.

[45] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 20.

[46] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.

[47] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen, "Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 15–28.

[48] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. ACM, 2005, pp. 63–74.

[49] M. Zitnik, M. Agrawal, and J. Leskovec, "Modeling polypharmacy side effects with graph convolutional networks," *Bioinformatics*, vol. 34, no. 13, pp. i457–i466, 2018.