AWB-GCN: A Graph Convolutional Network Accelerator with 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. 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 significant performance demands and design challenges.

In this paper, we propose Autotuning-Workload-Balancing GCN (AWB-GCN) to accelerate GCN inference. To address the issue of workload imbalance in processing real-world graphs, three hardware-based autotuning techniques are proposed: dynamic distribution smoothing, remote switching, and row remapping. In particular, AWB-GCN continuously monitors the sparse graph pattern, dynamically adjusts the workload distribution among a large number of processing elements (up to 4K PEs), and, after converging, reuses the ideal configuration. Evaluations are performed using an Intel D5005 FPGA with five commonly-used datasets. Results show that 4K-PE AWB-GCN can significantly elevate the average PE utilization (from 32.5% to 88.6%) and demonstrate considerable performance speedups over CPUs (7569×), GPUs (80.3×), and a prior GCN accelerator (7.4×).

1. INTRODUCTION

Deep learning paradigms such as Convolutional Neural Networks (CNNs) [27] and Recurrent Neural Networks (RNNs) [34] have been applied to a wide range of applications including image classification, video processing, speech recognition, and natural language processing. These paradigms, however, are only able to extract and analyze latent information from euclidean data such as images, video, audio and text [39]. This fact limits the adoption of neural networks. A large (and increasing) number of applications use non-Euclidean data structures that are modeled as graphs. Nodes and edges represent objects and relationships between those objects, respectively, as appropriate for the application. Most of these graphs have a tremendously large numbers of nodes; moreover, the node degree generally varies dramatically, often following a power law distribution [1, 3, 4, 12, 19, 29, 40].

The irregularity of the graph data makes most of the existing Neural Network (NN) algorithms fall short; critical feature extraction operations, such as convolutions, are no longer applicable. To tackle this issue, Graph Neural Networks (GNNs) have been proposed, in various forms, to extend deep learning approaches to graph data [14, 20, 30, 33, 38, 39]. Among various GNNs, 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 and become one of the most important topics in NN-based graph research [9, 15, 26, 46].

With the rapid development of GCNs, designing dedicated hardware accelerators has become an urgent issue [43]. GCNs have already been investigated in a large number of real-world applications [39], including electric grid cascading failure analysis [32], prediction of chemical reactivity [13], prediction of synthesized material property [41], polypharmacy side-effect modeling [50], accurate advertisement in E-commerce [44], and cybersecurity [35]. Many of these applications pose stringent constraints latency and throughput.

Accelerators developed for other domains, such as the sparse-CNN-accelerator (SCNN) [23, 25, 47], are not likely to be optimal as GCN accelerators for several reasons. (i) *GCN applications have highly unbalanced non-zero data distributions:* non-zeros can be clustered or appear in only a few rows. This leads to computing challenges [1, 19, 29] due to workload imbalance [40]. Figure 1 compares the distribution of non-zeros between a typical adjacency matrix in GCN and a typical sparse weight matrix in CNN: the distribution of non-zeros is much more balanced in the SCNN. (ii) *Extremely high sparsity*. The sparsity of a graph adjacency matrix often exceeds 99.9%, while the sparsity of SCNNs generally ranges from 10% to 50%. Therefore, the indices of consecutive non-zero elements are often highly scattered in GCNs, which makes it a major challenge to identify and access sufficient valid non-zero pairs to feed a massive number PEs per cycle at an acceptable hardware cost. Details are in Section 6. (iii) *Large matrix size.* Real-world graphs are usually quite large. For example, the *Reddit* graph has 233K nodes and 12M edges. Its 23K×23K adjacency matrix requires 1.7Tb storage in dense format and 2.3Gb in sparse format, which usually cannot fit into on-chip memory. Although neural networks also have large models, the matrix of

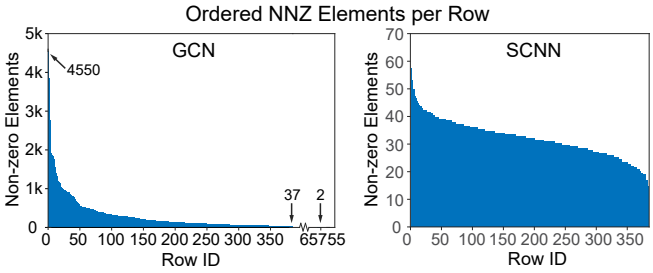

Figure 1: Ordered non-zero density-per-row histograms. Left: Adjacency matrix of the NELL graph (average density: 0.0073%) Most of non-zeros clustered in 70/66k rows. Right: Unstructured compressed AlexNet weight matrix (average density: 27%). Workload is roughly balanced across 384 rows.

a particular layer is often much smaller, e.g. 1k×1k, which means that the working set can often fit easily into on-chip memory.

For these reasons, novel and efficient accelerator designs are urgently required to accelerate GCN workloads. We therefore propose AWB-GCN, a hardware accelerator for GCN inference with workload auto-tuning. It monitors the workload distribution at three levels at runtime and, accordingly, rebalances the distribution per round[1].

Three techniques are proposed: *distribution smoothing*, *remote switching*, and *evil row remapping*. Distribution smoothing balances the workload among neighbors. In matrices following the power-law distribution, non-zero elements are usually clustered, and, in some cases, appear in just a few rows/columns (see Figure 2). Given only distribution smoothing, it would be slow and difficult for an autotuner to converge and achieve good load balance. We solve this problem with *remote switching* and *evil row remapping*. Remote switching shuffles workloads of regions with the most and least clustered non-zero elements, making efficient distribution smoothing possible. If a row is observed to *still* contain too many elements to be smoothed, or to be balanced by remote switching, then it is designated as an *evil row*. AWB-GCN partitions that row and remaps its non-zero elements to multiple regions (with least clustered elements) at runtime. Figure 3 shows the resulting per-round improvement in hardware utilization as these methods are applied (with Nell GCN).

This paper makes the following contributions:

- We propose a novel and efficient architecture for accelerating GCNs and Sparse Matrix Multiplication (SpMM) kernels for matrices with a power-law distribution.
- To handle the extreme workload imbalance, we propose a hardware-based workload distribution autotuning framework, which includes an efficient online workload profiler and three workload rebalancing techniques.
- We evaluate AWB-GCN using an Intel FPGA Acceleration Card D5005 with 5 most widely used GCN datasets.

[1]The calculation of one output matrix column is a round

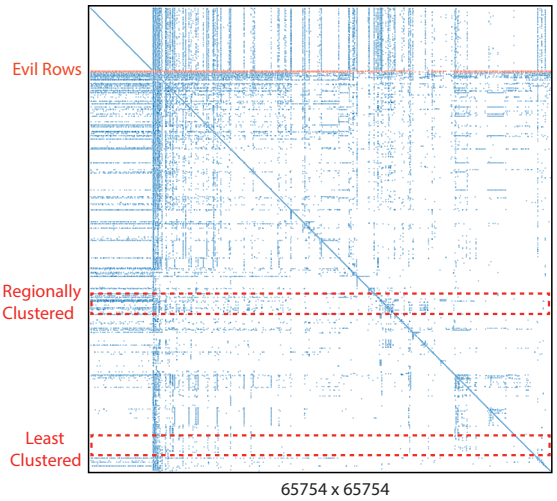

Figure 2: Adjacency matrix of NELL following power-law distribution: elements are clustered regionally and in a few rows/cols. The matrix density is 0.0073%. For better visualization, non-zero dots are enlarged.

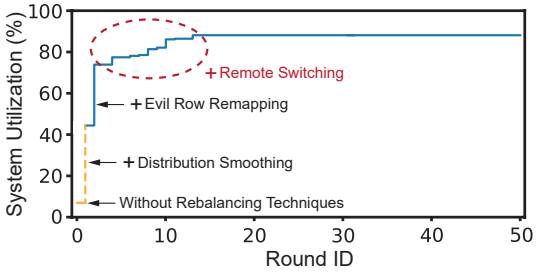

Figure 3: AWB-GCN utilization improvement per round.

Results show that 4K-PE AWB-GCN improves the average PE utilization from 32.5% to 88.6% as compared with the baseline without workload rebalancing. Compared with CPUs (Intel Xeon E5-2680v3 + PyTorch Geometric (PyG)), GPUs (NVIDIA Quadro RTX 8000 + PyG), and prior art, AWB-GCN achieves average speedups of 7569×, 80.3×, and 7.4×, respectively.

2. MOTIVATION

In this section we briefly introduce the GCN algorithm and discuss data characteristics of power-law graphs.

2.1 Graph Convolutional Network Structure

Equation 1 shows the layer-wise forward propagation of a multi-layer spectral GCN [26, 39]:

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

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* [27]. In general,

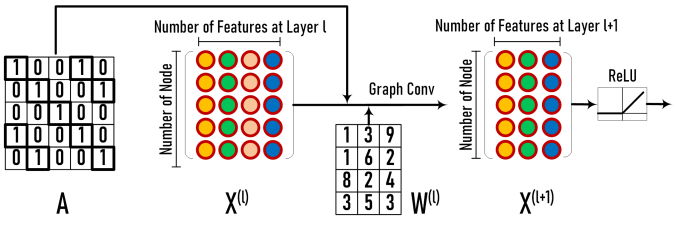

Figure 4: Illustration of a GCONV layer in GCNs.

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 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 $\mathscr{L} = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} = U \Lambda U$. The diagonal matrix Λ comprises the eigenvalues. If a frequency domain filter $g_W = diag(W)$ is defined, then Equation 2 can be simplified [9] as:

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

Equation 3 can be further simplified by defining the filter as the Chebyshev polynomials of the diagonal matrix Λ [15, 26] to obtain Equation 1.

Figure 4 illustrates the structure of a Graph Convolutional layer (GCONV). Each GCONV layer encapsulates the hidden features of nodes by aggregating information from neighbors of nodes. By multiplying A and $X^{(l)}$, information from 1-hop connected neighboring nodes are aggregated. By multiplying $AX^{(l)}$ with $W^{(l)}$, and going through the non-linear activation function $\sigma(\cdot)$, we obtain the output of this layer, which is also the feature matrix for the next layer $X^{(l+1)}$. The matrix A will normally be the same in different layers. After multiple layers, the GCN is able to extract very high-level abstracted features for various learning purposes.

2.2 Characteristics of Power-Law Graphs

Real-world graphs typically follow the *power-law* distribution [3, 4, 12, 40], which states that the number of nodes y of a given degree x is proportional to $x^{-\beta}$ for a constant $\beta > 0$. This implies that in the adjacency matrix A, a small number of rows (or columns) include the majority of non-zeros whereas the majority of the rows (or columns) contain only a few non-zeros but are not empty. Figure 5 shows the distribution of *non-zero* elements for the five publicly available datasets that are widely used for GCN evaluation [26]. The *power-law* effect is prominent for `Cora`, `Citeseer`, `Pubmed` and `Nell`.

Table 1: Matrix density and dimensions of 5 most commonly-used GCN datasets.

		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
	Feature	1433	3703	500	61278	602

Table 2: Operations required under different exec orders

Layer	Order	CORA	CITESEER	PUBMED	NELL	REDDIT
Operations	$(A \times X) \times W$	62.8M	198.0M	165.5M	258G	17.1G
	$A \times (X \times W)$	1.33M	2.23M	18.6M	782M	6.6G

Table 1 lists the density and dimension of matrices in the five GCN datasets used in this paper. As can be seen, Adjacency matrix A is always very sparse ($\geq 99\%$). Matrix X is also sparse. For the first layer, the sparsity ($X1$) is usually larger than 90%. As the weight matrix W is usually dense, the output of AXW is also dense. However, because of the *ReLU* activation function, the final output $X2$ (also the input of the next layer) becomes 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. A can be extremely large and is stored in a sparse format.

3. GCN BASELINE ARCHITECTURE

In this section we introduce our multi-core baseline architecture for GCN acceleration. This "baseline" supports efficient processing of power-law graphs with ultra-high sparsity and large sizes. The proposed baseline design cannot address the workload imbalance issue of power-law graphs, but build a good foundation for its further augmentation. In the next section, we augment this design to achieve near-optimal workload balancing.

3.1 Matrix Computation Order

To compute AXW at each GCONV layer, 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. For $(A \times X) \times W$, since multiplying A and X requires complex sparse-sparse-matrix-multiplication and produces a very large dense matrix, multiplying their product 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 2 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 multiply with A.

3.2 SpMM Execution Order and Mapping

We perform column-wise-product-based SpMM as described as follows. Given $S \times B = C$, if S is $(m \times n)$, B is

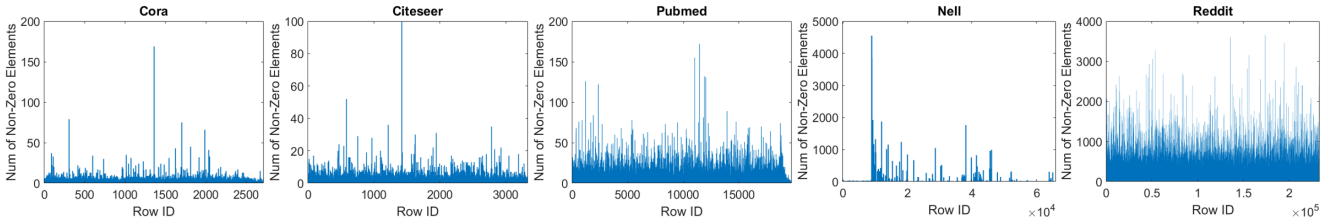

Figure 5: Non-zero distribution imbalance of Adjacency matrices in Cora, Citeseer, Pubmed, Nell and Reddit datasets

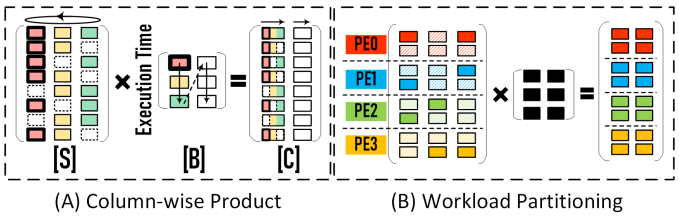

Figure 6: (A) SpMM computation order: Column-wise-product; (B) Matrix partitioning & mapping among PEs.

$(n \times k)$, and C is $(m \times k)$, then we can reformulate C as:

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

where S_j is the jth column of S and $b_{j,k}$ is an element of B at row-j and column-k. In other words, by broadcasting the jth element from column-k of B to the entire column-j of S, 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 is then evicted. In this way, we reuse the entire sparse matrix S for each column of C (k times in total). To reduce off-chip memory access for matrix S, we apply inter-layer data forwarding and matrix blocking techniques in our design (discussed in Section 3.4).

Such a design brings additional advantages when S and C are stored in *Compressed-Sparse-Column* (CSC) format. Furthermore, it provides opportunities to pipeline multiple SpMM operations, as will be discussed in Section 3.4. Moreover, column-wise-product brings massive opportunities of workload distribution autotuning which is key to the achievement of high performance. Figure 6(A) shows the column-wise order for calculating C. The columns of S and elements of B in the same color are multiplied and stored as partial results in C with the same color.

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 to avoid expensive parallel reduction in hardware as illustrated in Figure 6(B).

3.3 Design of Baseline Architecture

Figure 7 illustrates the baseline design for SpMM calculation with efficient support of skipping zeros. The architecture comprises of the modules *sparse-matrix-memory* (SpMMeM), *dense-column-memory* (DCM), *task-distributor & Queue* (TDQ), *PE-array*, and *accumulation-buffers-array* (ACC Buffer). SpMMeM buffers the input sparse matrix S (from off-chip) and feeds non-zeros and their indices to TDQ. DCM buffers the input dense matrix B and broadcasts its elements to TDQ. TDQ distributes tasks to the PEs. The PE-array performs concurrent multiplication of non-zero pairs, partial result accumulation, and data exchange with the ACC Buffers. Finally, the ACC Buffers cache the partial results of the resulting matrix C for accumulation and send them to the next SpMM engine at the completion of a whole column calculation. Depending on the sparsity and storage format of S, i.e. CSC, we have two alternative designs for TDQ:

TDQ-1 (Figure 7-left) is used when S is generally sparse (sparsity $< 25\%$) and stored in dense format. We perform the direct row partition as discussed and map non-zeros to the input buffer of corresponding PEs (see Figure 6(B)). At each cycle, $NPE/(1-Sparsity)$ elements are forwarded to PE array. Only non-zeros are kept in the queues. Here NPE denotes the number of parallel PEs. Given evenly distributed non-zeros, each PE receives one non-zero to calculate per cycle probabilistically. However, in practice, one PE has the chance to receive at most $1/(1-Sparsity)$ at one cycle. Therefore, each PE is equipped with multiple Task Queues (TQs) guaranteeing enough concurrency to cache all valid data. As shown in Figure 7-(left), in each cycle a PE can receive up to 4 non-zero elements. Four task queues are equipped per PE to buffer them. In each cycle, an arbitrator selects a non-empty queue, pops an element, checks for a Read-after-Write (RaW) hazard, and forwards it to the PE for processing. Since the computations are all floating-point, the latency of pipelined MAC unit is relatively longer, making *Read-after-Write*(RaW) hazard a critical problem. To address this problem, each PE is equipped with a *RaW-check-unit* and a stall buffer. If RAW hazard occurs, the new job is cached in the stall buffer until the hazard is resolved.

TDQ-2 (Figure 7-right) is used when S 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 of neighboring elements are highly scattered. We use a multi-stage Omega-network for routing the non-zeros to the correct PE according to their row indices. 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 by sending NPE non-zeros per cycle. This is achieved when non-zero elements are distributed evenly among rows. Compared with a global crossbar network, the Omega-network design scales better and incurs lower hardware complexity.

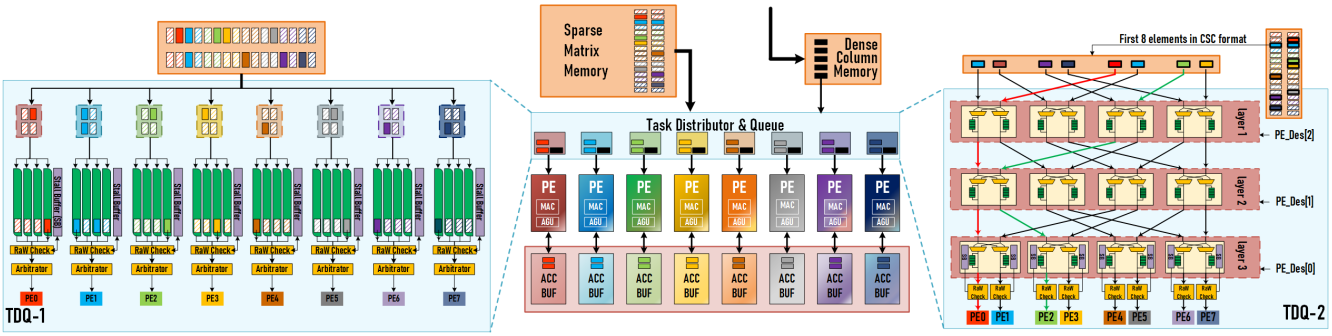

Figure 7: Architecture design of the proposed baseline SpMM engine.

When PEs receive new non-zeros from TDQ, they perform the new multiplication task, fetch corresponding partial results of output matrix C from ACC buffers according to the newly received row index, perform accumulation, and update the ACC buffers. Each PE is coupled with a bank of ACC buffer 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 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.

Overall, for each layer of GCN, we first execute SpMM on $X \times W$. Since X is generally sparse (except the first layer) 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.

3.4 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 through fine-grained pipelining. This is based on the observation that A is constant for the inference of a certain graph. Once a column of (XW) is calculated, we can start the multiplication of this column with A immediately without waiting for the entire XW (see Figure 8). This design has two major benefits: (i) we gain extra parallelism and reduce the overall latency through this fine-grained pipelining, and (ii) instead of requiring off-chip storage to cache the big 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 GCONV layer if (AXW) is left-multiplied by any 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 and processed in parallel.

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 layer (see Figure 8). In this way, the output generation of the previous layer matches the data consumption of the current layer, given optimal workload balance and PE utilization. Pipelining SpMMs from different layers has two benefits. First, it exploits inter-layer parallelism. Second, since A is shared for all GCONV layers in the inference of a particular graph, it can be reused by SpMM engines across the layers, so off-chip accesses of A are only required by the first layer. This is done by forwarding elements of A through the layers.

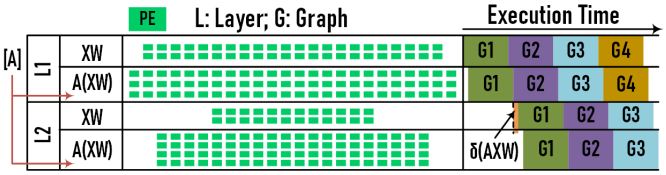

Figure 8: Pipelined SpMMs: data production and consumption rates match across consecutive SpMMs by allocating PEs in proportion to workload sizes.

Bandwidth Analysis: Off-chip data access of the big Adjacency matrix A can be a concern. However, as AWB-GCN always requests and consumes data with continuous addresses, the off-chip memory bandwidth and the burst mode access can be efficiently utilized. Also, we use three extra methods to reduce the off-chip bandwidth requirement: (1) as mentioned above, A is reused across layers; (2) matrix blocking is used to improve the data locality and reuse of matrix A. Figure 9 illustrates how the proposed matrix blocking works without affecting the efficiency of the rebalancing techniques which are discussed in the next section. The numbers in the figure are execution orders. A is partitioned into multiple blocks. Instead of calculating each column of $A(XW)$ by multiplying all blocks of A and the corresponding column of (XW), we calculate t columns of $A(XW)$ in parallel. The calculation of a certain block of A will not start until the previous block is reused t times and finishes calculating its intermediate results of all t columns of the resulting matrix. By doing so, the data reuse of matrix A is improved by t times. Note that this optimization will not hurt the efficiency of the autotuning rebalancing of AWB-GCN, as the sub-SpMM of each block of A is still following column-wise product order. (3) AWB-GCN is equipped with a scratchpad memory to cache parts of A on-chip as much as possible. For example, the A and $X1$ of Cora can be entirely stored on-chip.

Based on our evaluation, without the proposed matrix

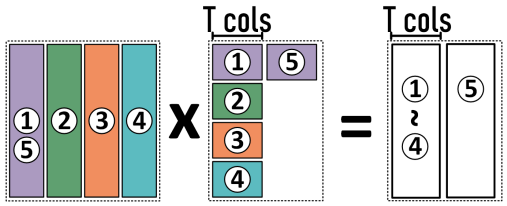

Figure 9: Matrix Blocking Optimization to reduce the off-chip bandwidth requirement. The sub-SpMM of each pair of blocks is performed in column-wise-product order. The numbers represent execution orders.

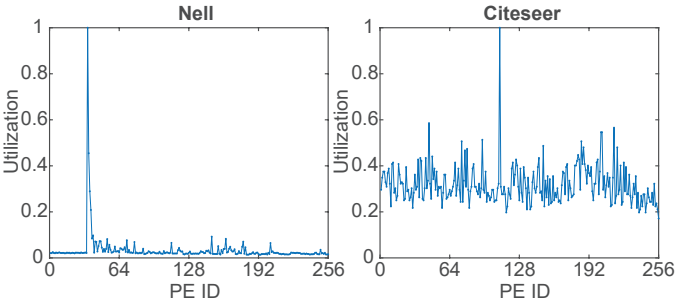

Figure 10: PE utilization waves of 256-PE Baseline SpMM engine processing $A \times (XW)$ of Nell and Citeseer.

blocking optimization, the AWB-GCN accelerator requires at most 459 Gbps off-chip bandwidth to keep the hardware busy with 1024 PEs for the 5 datasets evaluated. This bandwidth demand can be generally satisfied by current platforms (e.g., Intel D5005 FPGA board provides 614 Gbps DDR bandwidth; VCU-128 FPGA provides 3680 Gbps HBM bandwidth; NVIDIA V100 provides 7176 Gbps HBM bandwidth).

3.5 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. Figures 10(A) and (B) illustrate the utilization of 256 PEs processing SpMMs with Adjacency matrices of the Citeseer and NELL datasets. As mentioned in Section 1, evil rows and regionally clustered non-zeros in power-law graph matrices bring the inefficiency. The existence of evil rows keeps only a few PEs busy while all others idle most of the time, resulting in significant major crests in the utilization waves; the regionally clustered non-zero elements result in the minor crests; the differences in the numbers of non-zeros in neighboring rows result in other fluctuations.

A common software approach for dealing with sparse data structures is to profile the structure, e.g. with symbolic analysis, and then use that information to guide the "real" processing. For GCNs, however, it has been demonstrated that the preprocessing stage can take 10× more time than the inference itself [43]. In this work, we *dynamically* adjust hardware configurations for workload rebalancing. This design can be applied to a variety of specialized accelerators for processing sparse data structures.

4. AWB-GCN ARCHITECTURE

In this section, we describe the AWB-GCN architecture. The core is the handling of load balancing at three levels of granularity: *distribution smoothing* for local utilization fluctuations among PEs, *remote switching* for the minor crests, and *row remapping* for the major crests.

Figure 11 illustrates autotuning with 24 PEs performing SpMM on a power-law matrix. The gray bars at the top show the execution time of parallel PEs; the length changes dynamically through the process. The narrower bars at the bottom show the static density-per-row of the matrix. Ideally, at the end of autotuning, all bars on the top becomes short and have the same length. Each round of autotuning includes two phases: First, data processing and distribution smoothing in phase 1; then remote switching and row remapping.

Figure 11 (a)&(b) illustrate the first round of autotuning. The progression from Figure (a) to (b) shows the first phase. Figure (a) gives estimated execution time without distribution smoothing; Figure (b) shows the actual execution time with distribution smoothing applied. During phase 1, PEs keep offloading workloads to their less busier neighbors, resulting in a more flat and smooth execution time wave (shown in (b)). Meanwhile, the execution time of PEs at the wave crests and troughs is recorded by the Autotuner.

After all the PEs are done, phase 2 starts. The Autotuner partitions and remaps evil rows to PEs at troughs and switches workloads of the PEs at the minor crests with the ones at the troughs. The green and blue arrows in (b) show evil row remapping and remote switching decisions, respectively. After these decisions are made, the second round of autotuning starts (Figures (c)&(d)). With remote switching and row remapping determined in the first round, the initial workload distribution among PEs at the start of the second round (shown in (c)) can be more efficiently balanced by distribution smoothing (shown in (d)). The blue arrows in (d) show that remote balancing not only finds new pairs of PEs to switch workloads, but also adjusts the switch fractions determined in the previous round. After several rounds, the system converges to optimal balanced status; this is then used for the remainder of the computation.

All profiling and adjustment are performed at runtime. We now present design details.

4.1 Distribution Smoothing

At the start of processing, rows are evenly distributed among PEs as introduced in Section 3.2 (as shown in Figure 6(B)). During the calculation of each round, we employ distribution smoothing by averaging out the workloads among neighbors. The architecture is able to monitor the runtime PE utilization information by tracking the number of pending tasks in TQs and keep offloading the work of PEs with more pending tasks to their less busy neighbors. However, the offloaded work needs to be returned for aggregation after processing. Due to chip area and design complexity restrictions, we may offload workloads among direct neighbors, 2-hop neighbors, or even 3-hop neighbors, but not farther ones.

Figure 12 illustrates the hardware design of 1-hop distribu-

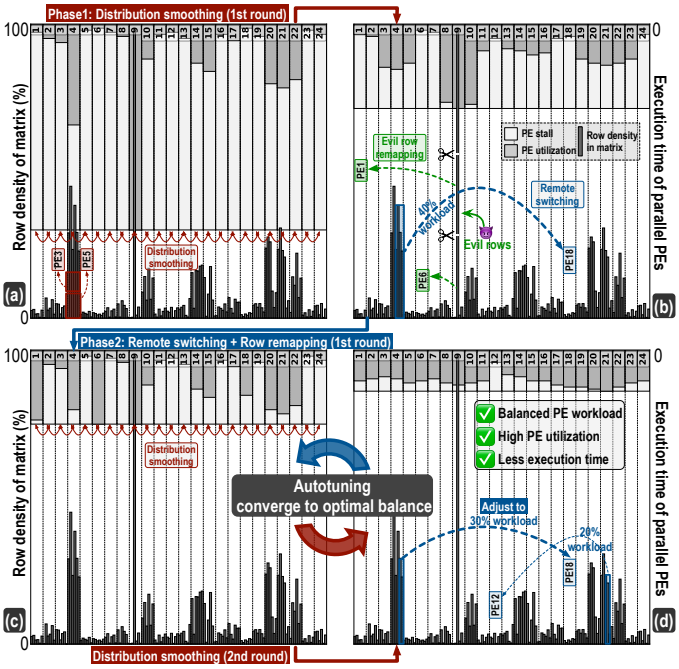

Figure 11: Rebalancing process: distribution smoothing, remote switching and row remapping per round. (a)&(b): 1st round; (c)&(d): 2nd round.

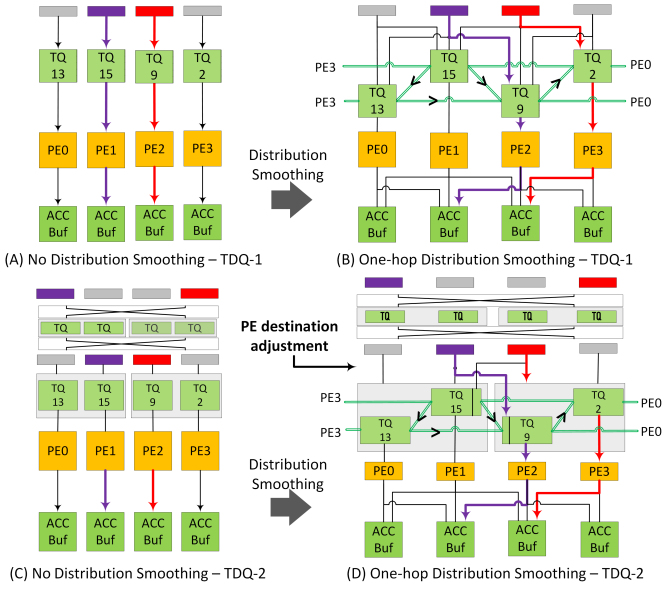

Figure 12: Architecture of distribution smoothing.

tion smoothing for TDQ-1 and TDQ-2.

TDQ-1: Before a new task is pushed into the TQ of a PE, 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 after accumulation (see Figure 12-(B)). The calculation of valid return address and accumulation of partial results are done in the neighbor PE.

TDQ-2: The final layer of the multi-stage Omega network handles neighbor task forwarding. As shown in Figure 12-(C) (also in Figure 13), multiple PEs share the same final-layer switch; we refer to these PEs as a *group*. AWB-GCN keeps tracking the usage of TQs of the final layer. Once a new task is forwarded to the final-layer switch, the TQ usages among neighbors are compared and then the task is routed to the PE with the lowest TQ usage. To enable PEs on the group edge (i.e. the leftmost or rightmost PEs per group) to communicate with their out-of-group neighbors, we augment the Omega-network by adding 2 extra links per switch in the final layer, as shown in Figure 12-(D). Note 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.

Distribution smoothing helps remove local utilization fluctuations (Figures 11(a) to (b)), but is not sufficient when (1) non-zeros are clustered in a region across many PEs, so that neighbors are mostly busy and have no chance to help each other, resulting in a minor utilization crests (PE20,21,22 in Figure 11(b)); or (2) most non-zeros are clustered in only a few rows so that the major crests cannot be eliminated even if all neighboring PEs help (PE9 in Figure 11(b)).

4.2 Remote Switching

To address regional clustering, we propose remote switching. This process partially or completely exchanges the workloads between under- and overloaded PEs, i.e., at centers of utilization wave troughs and crests, respectively. The switch fraction is determined at runtime by an autotuner and is based on per-round PE utilization. As the sparse matrix A is reused during the processing per round, the switch strategy generated in prior rounds is very valuable in the processing of the later rounds. The accelerator remembers the switch strategies being used in the current round and incrementally optimizes it based on the utilization information obtained in the next round. In this way, distribution smoothing is able to flatten the crests and troughs; after several rounds of autotuning, the switch strategy best matching the sparse structure of A is obtained, and we use it for the remaining rounds obtaining almost perfect PE utilization.

The hardware design is shown in Figure 13. The most over-loaded and under-loaded PEs are identified by using the *PE Status Monitor* (PESM) of Autotuner during Phase one of autotuning. Recall that each TQ has a counter to track the number of pending tasks; these can trigger an *empty* signal when reaching zero. These empty signals are connected to the PESM. At each cycle, the updated empty signals are *xor*ed with their values recorded on the previous cycle. The XOR results indicate which PEs are newly finished; this information is stored in the *Switch Candidate Buffer*.

At the start of each round (after A is totally sent to TDQs), the arbiter keeps scanning the buffer and record IDs of newly done PEs until enough under-loaded PEs have been found. The number of PE tuples for switching at each round can be customized. In Figure 13, four tuples of the most over- and under-loaded PEs need to be selected for remote switching.

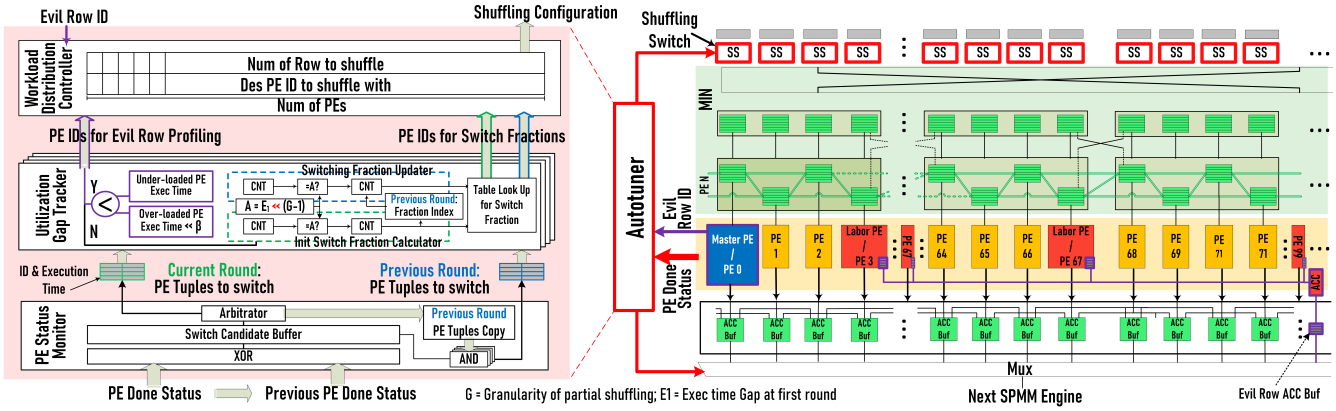

Figure 13: Overall architecture of SpMM engine in AWB-GCN with three rebalancing techniques: distribution smoothing, remote switching (red bordered) and evil row remapping (purple bordered).

After the arbiter finds the first 4 idle PEs, it stops scanning the buffer and instead starts to wait for the completion signal ($bit-AND$ all empty signals) from the system, which implies all PEs have become idle. Meanwhile, the Switch Candidate Buffer keeps caching the newly idle info of the most recent cycles. Whenever the arbiter receives the completion signal it starts to scan the buffer and keeps scanning until the four most over-loaded PEs have been found. Note that the arbiter does not select neighbor PEs continuously; this guarantees that PE tuples selected by PESM are at different crests and troughs of the utilization wave.

To avoid thrashing, we only exchange a portion of the workload between PEs. We use the following equation 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_init:

$$N_i\_init = G_i/G_1 \times (R/2) \tag{5}$$

where G_i is the workload gap of the selected PE tuple at the i-th round, and R is the number of rows per PE under equal mapping. Here, workload gap is approximated as the difference of execution cycles to finish all tasks.

In the $i+1$-th round, new PE-tuples are selected and their switch fractions are calculated. Meanwhile, the autotuner also tracks the post-switching utilization gaps of PE-tuples selected in the prior rounds and uses them as feedback to adjust the switch fraction N_i_init; this minimizes the utilization gaps further. The workload switching fraction for each tracked PE-tuple is adjusted for two or more rounds and is highly likely to converge to the optimal distribution. Equation 5 can now be rewritten as follows:

$$N_{i,j} = \begin{cases} G_i/G_1 \times (R/2) & if\ j = 0 \\ N_{(i-1),(j-1)} + \alpha \times G_i/G_1 \times (R/2) & if\ j > 0 \end{cases} \tag{6}$$

where j denotes the number of rounds of fraction update. $N_{i,j}$ indicates that the current PE-tuple is in its j-th update and its initial fraction to switch was calculated in the $i-j$-th round. The number of rounds tracked simultaneously can be customized and depends on the size of the tracking window in the PESM; this is an area/performance tradeoff. In Figure 13 two consecutive rounds are tracked, which is found to be sufficient for GCN datasets.

Calculation of Equation 6 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-friendly approximation with threshold-based counting and table lookup; details are omitted due to space limitations. Once the number of rows to be switched is known, it is forwarded to the *Workload Distribution Controller* (WDC) together with the corresponding PE IDs. At the start of the next round, the destination PE of these rows is updated in the *Shuffle Switches* (SS). By doing so, the non-zeros in these rows will be forwarded to the post-switching PEs in the coming rounds.

Remote switching followed by distribution smoothing is efficient on getting rids of most of crests of utilization waves. However, for the major crests resulted from evil rows which have too many non-zeros to be shared only by neighbors, extra effort is required.

4.3 Evil Row Remapping

We address evil-row clustering by building row remapping support into the remote switching hardware. With row remapping, the evil row is distributed to the most under-loaded PEs in troughs; in this way the neighbors of these PEs can help. Row remapping is triggered based on demand at the end of each round. The autotuner calculates the utilization gaps between the most over- and under-loaded PEs and determines whether their gaps are too big for remote switching to handle. If yes, row remapping is performed. The workloads of the PE overloaded in the current round are switched (temporarily) with a *Super-PE* in the next round. During processing of the next round, the Super-PE counts the numbers of non-zeros per row and finds the evil rows containing the most non-zeros. In the round after, the workloads of each evil row are distributed to a set of *Labor-PEs* controlled by the Super-PE. In our implementation, every 128 PEs has one Super-PEs four Labor-PEs. These numbers can be customized.

When evil rows are remapped to Labor-PEs, the original workloads of the Labor-PEs are swapped with the most under-loaded PEs via remote switching; this ensures that the Labor-PEs do not become new crests after row remapping. By remapping evil rows statically to certain PEs instead of

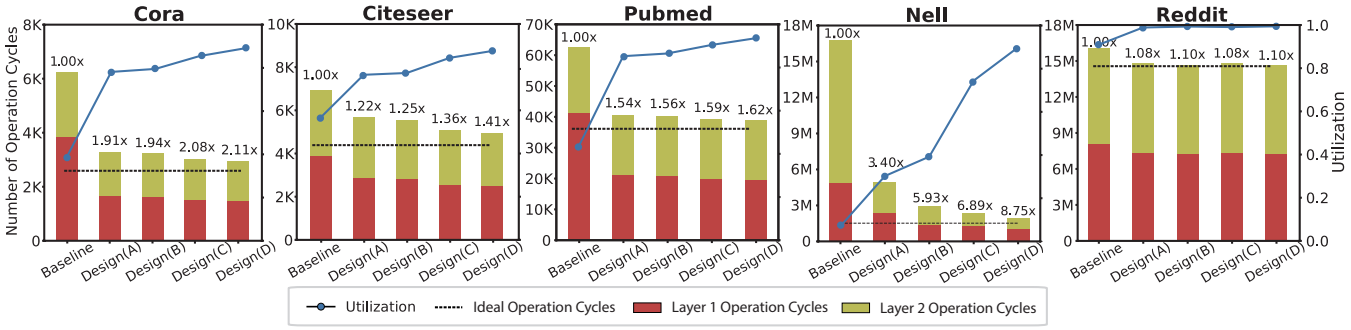

Figure 14: Overall performance and PE utilization of 1K-PE AWB-GCN with five design choices.

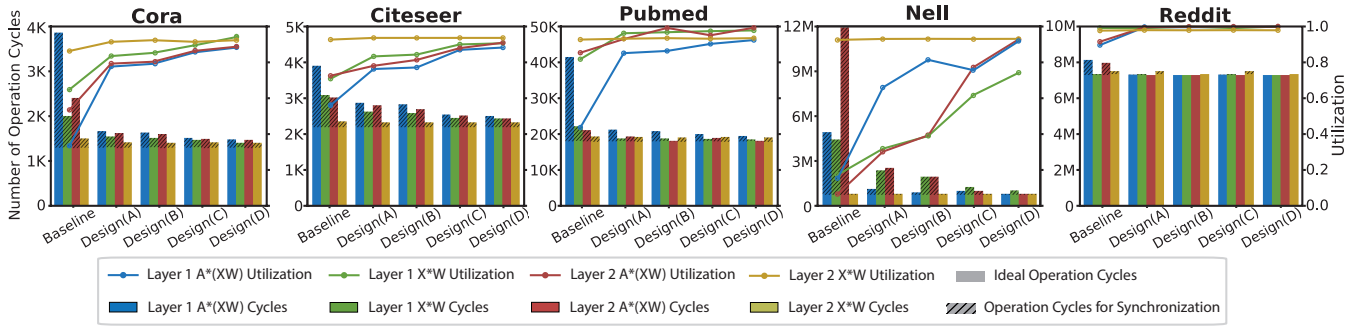

Figure 15: Per-SpMM performance and PE utilization of 1K-PE AWB-GCN with five design choices.

dynamically to random ones, the aggregation of partial results becomes hardware efficient. If row remapping is not triggered, Super- and Labor-PEs serve as normal PEs.

The existence of evil rows is generally the most critical bottleneck, especially when utilization is lower than 50%. The proposed row remapping technique makes it possible for the autotuner to find the optimal workload distributions and achieve high utilization. As evil row remapping is normally triggered during the first few rounds, the utilization of the system increases rapidly right at the start and the autotuner generally converges quickly.

Figure 13 illustrates the hardware support of row remapping. For clarity, only one Super-PE and its four Labor-PEs are shown. The Labor-PE has an architecture similar to the normal PE, but they are connected to an adder tree for result aggregation. The aggregated results of evil rows are cached in a small separate ACC buffer. The super-PE is much bigger than other PEs, as it serves as a profiler to find the evil rows. It is equipped with two extra modules: a parallel sorting circuit that tracks the rows with the most non-zeros (eight in our implementation); and a non-zero counter (including a local buffer) that records the number of non-zeros per row. Workload remapping between Super-PE & Labor-PEs and workload switching between Super-PE & the PE-with-evil-rows are handled by augmenting the Autotuner as follows. First, a Utilization Gap Tracker (UGT) module is equipped with a comparator to identify whether evil rows exist; if they do, then the UGT will send the information to WDC. The WDC knows the IDs of the Super-PE and Labor-PEs. If row remapping is triggered or an evil row is found, then the entries of the Super- and Labor-PE at Distribution Switch Table in the WDC will be updated. This enables workload switching and remapping to be in the coming round.

5. EVALUATION

In this section, we evaluate AWB-GCNs with different design choices and compare them with other platforms processing the same networks.

5.1 Evaluation Configuration

We implement AWB-GCNs in Verilog HDL and measure PE utilization, performance, energy efficiency, and hardware resource consumption on Intel acceleration card D5005 which is equipped with a Stratix 10 SX FPGA. Note that the FPGA is only used as an evaluation platform to demonstrate the performance of AWB-GCN. The design is a general architecture that does not leverage any FPGA-specific features.

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 also measured with a cycle-accurate hardware counter. The hardware consumption and operating frequency are reported by Quartus Pro 19.4 after synthesis and implementation. To perform fair cross-platform comparisons, we implement GCNs with the state-of-the-art and famous software framework, PyG [17], and run them on Intel Xeon E5-2680-V3 CPU and NVIDIA RTX 8000 GPU. We also compare AWB-GCN with prior work on GCNs such as HyGCN [43].

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.

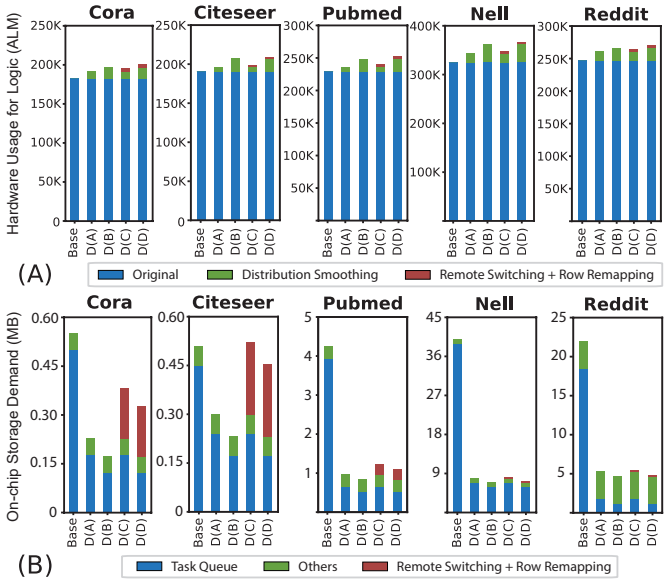

Figure 16: (A) Hardware resource consumption normalized to the number of ALMs and (B) On-chip storage demand of 1K-PE AWB-GCN with five design choices.

5.2 AWB-GCN Evaluation

Design efficiency is evaluated by comparing the performance, hardware resource consumption, and PE utilization of the 1K-PE baseline design without any rebalancing techniques (i.e., ***Baseline***) with the four different design choices of 1K-PE AWB-GCNs: (i) 1-hop distribution smoothing (i.e., ***Design(A)***), (ii) 2-hop distribution smoothing (i.e., ***Design(B)***), (iii) 1-hop distribution smoothing plus remote switching (i.e., ***Design(C)***), and (iv) 2-hop distribution smoothing plus remote switching (i.e., ***Design(D)***). The only exception is for *Nell* where we use 2-hop and 3-hop distribution smoothing (rather than 1-hop and 2-hop) due to its extremely clustered distribution.

Figure 14 compares the end-to-end GCN inference latency and average utilization of PEs for the five designs over the five datasets. The lines show the overall PE utilization. The bars show the breakdown of execution cycles of different GCN layers. The latency of ReLU is too low to show in the figure. The off-chip memory access latency is overlapped with computation. We also mark the latency lower bound assuming theoretically ideal PE utilization. For *Cora*, *Citeseer*, *Pubmed*, *Nell* and *Reddit*, comparing to Baseline, Design(B) can improve PE utilization from 38%, 56%, 44%, 7.1% and 90%, to 79%, 77%, 86%, 39%, and 99%, respectively, leading to 1.94×, 1.25×, 1.56×, 5.93×, and 1.10× performance improvement. Enabling remote switching can further improve PE utilization to 90%, 88%, 94%, 90%, and 99%, bringing performance gain to 2.11×, 1.41×, 1.62×, 8.75×, and 1.10×. The results show that AWB-GCN always provides high utilization and close to theoretical peak performance for datasets with various levels of power-law distribution.

The remaining 1%-12% gaps between the utilization of Design(D) and theoretically ideal ones are due to PE underutilization in the autotuning phase. In AWB-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 lengths at Design(D), while their lengths vary significantly at Baseline.

Figure 15 further breaks down the numbers of execution cycles and shows results for every SpMM kernel; this demonstrates the benefits of AWB-GCN on kernels with various sparsity, size and distributions. 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 execution cycles of the four SpMM kernels in the two-layer GCNs [26, 39]: $A \times (XW)$ *and* $X \times W$ *at Layer 1 and 2*. The lines show the corresponding PE utilizations. As shown in Figure 15, Design(D) significantly minimizes the synchronization overheads for all kernels of the 5 GCN models.

Comparing SpMM kernels, utilization improves significantly for $A \times (XW)$ at both layers and $X \times W$ at Layer-1. As for $X \times W$ at Layer-2, although X is also sparse after activation is performed, its sparsity is much lower than that of the X at Layer-1 and its non-zero distribution does not follow the power-law (similar to that of the sparse matrices in SCNNs); utilization is thus high even with the baseline design.

Figure 16(A) compares the hardware resource usage of the five designs over the five datasets. Results are normalized to the number of *Adaptive Logic Modules* (ALMs), which is the basic component of Intel FPGAs. In an ASIC design analogue would be the number of transistors. The blue segments represent the resource usage for the modules of the baseline design; the green and red segments refer to the hardware overheads for the support of *distribution smoothing* and *remote switching + row remapping*. As shown in figure (A), the overheads of 1-hop and 2-hop distribution smoothing are on average 3.5% and 6.7%, respectively, which is acceptable; the overhead of remote switching and row remapping is, on average 0.9%, which is negligible.

Figure 16(B) compares on-chip storage demand. That of Task Queues in the Omega-Network is in blue; the buffers for remote switching + row remapping are in red; the others are in green. As shown in Figure (B), the overall storage demands of AWB-GCN with Design(D) are even lower than the baseline. This is largely due to dramatically reduced per-PE Task Queue size under more balanced workloads. With much more balanced workload distributions in Design(D), the congestion and backpressure in Omega-Network are significantly relieved, making the TQs narrower and shallower.

Finally, Figure 17 shows the utilization improvement due to iterative workload autotuning. Rebalancing can be accomplished within 10 iterations. This means that most of the iterations can benefit from operating under the converged optimal strategy. As shown in the figure, the utilization of *Nell* has a sharp improvement at round 3 which is the result of effective evil row remapping.

5.3 Scalability of AWB-GCN

We evaluate the scalability of AWB-GCN by running GCN inference of the five datasets on the baseline as well as Designs (B) and (D) of AWB-GCN and varying the number of

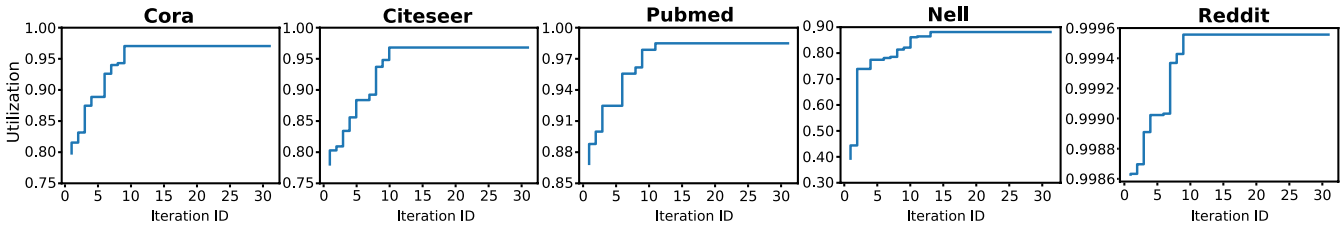

Figure 17: AWB-GCN PE (1K) average utilization per round of workload autotuning.

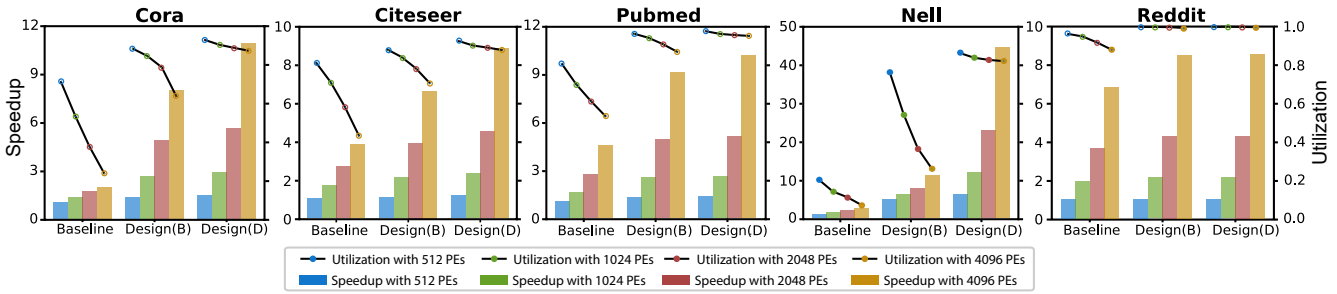

Figure 18: Scalability evaluation: PE utilization and overall performance of Baseline, Design(B) and Design(D) of AWB-GCNs with 512, 1K, 2K and 4K PEs.

PEs from 512, 1024, 2048 to 4096. In Figure 18, the bars represent the performance speedup comparing with the baseline design with 512 PEs. The lines represent average PE utilizations.

As shown in Figure 18, the PE utilization of the baseline design drops dramatically with increasing number of PEs. This is because more PEs means fewer rows per PE, highlighting the imbalance among PEs: they have fewer opportunities to absorb inter-row imbalance. Due to the dropping PE utilization, the performance speedup shows poor scalability. For AWB-GCN with only distribution smoothing, PE utilization also drops but more slowly than baseline. *Nell* is an outlier as the utilization of baseline with 512 PEs is too low to drop. In contrast, the PE utilization of the complete version of AWB-GCN, Design(D), is high and stable. The performance scales linearly with increasing number of PEs.

5.4 Cross-platform Comparison

We evaluate four scenarios: (i) AWB-GCN Design-(D) with 4096 PEs, (ii) PyG-based implementation on Intel Xeon E5-2680v3 CPU (PyG-CPU), (iii) PyG-based implementation on a NVIDIA RTX 8000 GPU (PyG-GPU), and (iv) 4096-PE baseline AWB-GCN without workload rebalancing. We use the five datasets: Cora, Citeseer, Pubmed, Nell and Reddit for the evaluation. The GCN model configuration follows the original GCN algorithm papers [10, 26, 48]. We label these GCNs "*Standard_networks*".

As shown in Table 3, despite running at a relatively low frequency, AWB-GCN achieves, on average, speedups of 6017× and 135.5× over the well-optimized PyG implementations on high-end CPUs and GPUs. It achieves a speedup of 6.0× over the baseline design without workload rebalancing. For the Nell dataset, the speedup over the baseline is 18.8×, demonstrating in particular the impact of workload balancing. Reddit fails on GPU due to out-of-memory.

The tremendous speedups over PyG-CPU and PyG-GPU originate from AWB-GCN's dedicated architecture which uses features not available on general-purpose CPUs and GPUs: (a) the dynamic autotuning techniques ensure balanced workload and high PE utilization (Section 4); (b) all the SpMM kernels of the GCN layers are deeply pipelined, leading to reduced on-chip storage demand (Figure 8); (c) inter-layer data forwarding and matrix blocking (with column-wise-product sub-SpMM execution Figure 9) improve data reuse and guarantee that off-chip memory accesses are to consecutive addresses.

We also compare AWB-GCN with existing GCN accelerators. Prior to this work, to the best of our knowledge, the design proposed by Yan et al., HyGCN [43], is the only reported accelerator of GCN inference. However, HyGCN customizes the hidden layer of all GCN models to 128 channels, which is distinct from the original settings [10, 26, 48]. We refer to the HyGCN-customized models as *HyGCN_networks*. Also, the HyGCN report does not give absolute performance but, rather, relative speedups over a E5-2680v3 CPU. To compare AWB-GCN with HyGCN, we realize the *HyGCN_networks* on the same E5-2680v3 CPU, adopting the same software framework (PyG [17] – HyGCN also uses PyG for the testing on CPU). The PyG-CPU result is thus a common baseline for comparing the relative speedups. Table 4 shows the results.

With the HyGCN_networks, AWB-GCN achieves on average 9120×, 25.3×, and 7.4× speedups over PyG-CPU, PyG-GPU, and the HyGCN design, respectively. The performance improvement is attributable, in part, to the features of AWB-GCN as discussed, and one additional reason: that HyGCN scheduling is coarse-grained block-wise, while that of AWB-GCN is fine-grained element-wise. This optimization avoids redundant operations and results in more benefit from a balanced workload.

Table 3: Comparison with CPU, GPU and Baseline processing Standard_networks. OoM: Out of Memory.

Platform	Standard_networks	Cora	CiteSeer	Pubmed	Nell	Reddit
Intel Xeon E5-2680 (PyG)	Latency (ms) [speedup]	2.51 [1×]	3.66 [1×]	13.97 [1×]	2.28E3 [1×]	2.94E5 [1×]
Freq: 2.5GHz	Energy efficiency (graph/kJ)	6.68E3	3.88E3	1.03E3	6.99	5.43E-2
NVIDIA RTX8000 (PyG)	Latency (ms) [speedup]	0.69 [3.6×]	0.68 [5.4×]	0.69 [20.2×]	90.50 [25.2×]	OoM
Freq: 1395MHz	Energy efficiency (graph/kJ)	1.06E4	1.29E4	1.11E4	89.06	OoM
Baseline Intel D5005 FPGA	Latency (ms) [speedup]	1.3E-2 [191.8×]	9.0E-3 [406.0×]	6.7E-2 [207.6×]	30.09 [75.8×]	13.96 [21036.7×]
Freq: 330MHz	Energy efficiency (graph/kJ)	6.86E5	9.75E5	1.22E5	3.28E2	6.19E2
AWB-GCN Intel D5005 FPGA	Latency (ms) [speedup]	2.3E-3 [1062.6×]	4.0E-3 [913.2×]	3E-2 [465.8×]	1.6 [1425.4×]	11.20 [26220×]
Freq: 330MHz	Energy efficiency (graph/kJ)	3.08E6	1.93E6	2.48E5	4.12E3	5.98E2

Table 4: Comparison with the prior art, HyGCN, processing HyGCN_networks customized in HyGCN paper [43].

Platform	HyGCN_networks	Cora	CiteSeer	Pubmed	Nell	Reddit
Intel Xeon E5-2680 (PyG)	Latency (ms) [speedup]	13.07 [1×]	15.73 [1×]	2.19E2 [1×]	3.17E3 [1×]	8.05E5 [1×]
Freq: 2.5GHz	Energy efficiency (graph/kJ)	1.23E3	9.36E2	70.61	4.59	0.02
NVIDIA RTX8000 (PyG)	Latency (ms) [speedup]	0.69 [18.9×]	0.69 [22.8×]	1.31 [166.8×]	100.18 [31.7×]	OoM
Freq: 1395MHz	Energy efficiency (graph/kJ)	1.16E4	1.09E4	6.46E3	79.81	OoM
HyGCN TSMC 12 nm	Latency (ms) [speedup]	2.1E-2 [627.2×]	0.30 [52.1×]	0.64 [341.5×]	NA	2.89E2 [2786.9×]
Freq: 1GHz	Energy efficiency (graph/kJ)	7.16E6	4.94E5	2.33E5	NA	5.17E2
AWB-GCN Intel D5005 FPGA	Latency (ms) [speedup]	1.7E-2 [767.5×]	2.9E-2 [548.0×]	0.23 [948.4×]	3.25 [977.89×]	19 [42361.7×]
Freq: 330MHz	Energy efficiency (graph/kJ)	4.39E5	2.71E5	3.17E4	2.28E3	3.75E2

6. RELATED WORK

GNN studies use neural network algorithms to address problems in graph processing. The first GNN model was proposed by Gori et al. [20]. In the past decade, work has continued on optimizing GNN algorithms exploring new neural network approaches [2,14,18,33,38,39,45]. More recently, inspired by CNNs that achieve great success with euclidean data, GCNs are proposed for hidden feature extraction of non-euclidean data. In 2013, Bruna et al. [9] proposed the first GCNs for spectral graph theory; this was developed further in a number of variants [15,24,26]. GCNs are at the center of the research on neural-network-based graph processing [46].

There have been many efforts on accelerating sparse CNNs [5,11,16,23,25,28,37,47]. We 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 [28]. In case of conflict, only the most significant parameters are kept, others are discarded. Essentially, some accuracy is sacrificed for performance. Kim et al. [25] address the workload imbalance problem of sparse CNNs, but use information from design-time profiling and pre-scanning. Han et al. [23] propose EIE, an SpMV accelerator that addresses imbalance with row-direction queuing. The design is not feasible in GCNs due to their large data size and power-law distribution. In EIE, weight matrices of SCNNs are distributively pre-stored on-chip in local buffers of PEs. This avoids off-chip non-zero accesses and online workload distribution, but is not possible for GCNs. Also, single-direction queuing fails to balance the workload of power-law matrices, which have serious imbalance on both directions. Zhang et al. [47] propose Cambricon-S with efficient index matching to identify and multiplex non-zeros and feed them to massive parallel PEs. These proposed architectures are not feasible in GCNs due to the ultra-low sparsity of power-law graphs which lead to highly scattered indices of neighboring elements. Given the adjacency matrix of Nell and a 1024-PE Cambricon-S, multiplexing enough non-zero pairs to feed all PEs per cycle would require 1024× 13699:1 multiplexers for single-precision floating point; this is not viable given likely chip technology.

Besides work on sparse CNNs, researchers also propose architectures for general SpMM. Zhuo and Prasanna [49] present an SPMV design for FPGAs. Pal [36] proposes an outer-product-based SpMM architecture. This work focuses on reducing redundant memory accesses to non-zeros and does not essentially address the ultra-workload-imbalanced issue faced with GCNs. In their 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 GPUs and general-purpose multicore CPUs [6, 7, 8, 21, 31]. These software solutions, however, do not meet the strict timing requirements of GCNs because of significant overhead in pre-scanning [6, 21, 31, 43] which is avoided in AWB-GCN. Also, adjacency matrices evolve at runtime, making offline processing even less useful.

7. CONCLUSION

In this paper, we propose AWB-GCN to accelerate GCN inference. To tackle the major performance issues derived from workload imbalance, we propose a hardware-based autotuning framework including three runtime workload rebalancing techniques: distribution smoothing, remote switching, and row remapping. The proposed rebalancing methods 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 evaluate AWB-GCN using an Intel FPGA D5005 Accelerator Card with 5 widely used GCN datasets. Results show that AWB-GCN can achieve, on average, 7569×, 80.3×, and 7.4× speedups, respectively, over high-end CPUs, GPUs, and prior work on GCNs. Although FPGAs are used as a demonstration in this paper, the proposed architecture does not rely on any FPGA-specific features. And although AWB-GCN is designed for GCNs, it is generally efficient for GNNs which major arithmetic primitives are also SpMMs, such as GraphSage [22], GINConv [42], and GTN [46].

REFERENCES

[1] A. Abou-Rjeili and G. Karypis, "Multilevel algorithms for partitioning power-law graphs," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2006, pp. 10–pp.

[2] 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.

[3] L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman, "Search in power-law networks," *Physical review E*, vol. 64, no. 4, p. 046135, 2001.

[4] W. Aiello, F. Chung, and L. Lu, "A random graph model for power law graphs," *Experimental Mathematics*, vol. 10, no. 1, pp. 53–66, 2001.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] J. Chen, J. Zhu, and L. Song, "Stochastic training of graph convolutional networks with variance reduction," *arXiv preprint arXiv:1710.10568*, 2017.

[11] 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.

[12] F. Chung, L. Lu, and V. Vu, "The spectra of random graphs with given expected degrees," *Internet Mathematics*, vol. 1, no. 3, pp. 257–275, 2004.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[18] 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.

[19] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[20] 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.

[21] 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.

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

[23] 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.

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

[25] 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.

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

[27] 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.

[28] 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.

[29] M. Latapy, "Main-memory triangle computations for very large (sparse (power-law)) graphs," *Theoretical computer science*, vol. 407, no. 1-3, pp. 458–473, 2008.

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

[31] 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.

[32] 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.

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

[34] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh annual conference of the international speech communication association*, 2010.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[40] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," in *Advances in neural information processing systems*, 2014, pp. 1673–1681.

[41] 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.

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

[43] 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," *arXiv preprint arXiv:2001.02514*, 2020.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] C. Zhuang and Q. Ma, "Dual graph convolutional networks for graph-based semi-supervised classification," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 499–508.

[49] 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.

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