Dancer: Dynamic Compression and Quantization Architecture for Deep Graph Convolutional Network

Yunhao Dong [1], Zhaoyu Zhong [1], Yi Wang [1,2], Chenlin Ma [1], Tianyu Wang [1]
1. College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
2. State Key Laboratory of Radio Frequency Heterogeneous Integration
{*dongyunhao2019, 2210275018*}*@email.szu.edu.cn*, {*yiwang, chenlin.ma, tywang*}*@szu.edu.cn*

***Abstract*—Graph Convolutional Networks (GCNs) have been widely applied in fields such as social network analysis and recommendation systems. Recently, deep GCNs have emerged, enabling the exploration of deeper hidden information. Compared to traditional shallow GCNs, deep GCNs feature significantly more layers, leading to considerable computational and data movement challenges. Processing-In-Memory (PIM) offers a promising solution for efficiently handling GCNs by enabling near-data computation, thus reducing data transfer between processing units and memory. However, previous work mainly focused on shallow GCNs and has shown limited performance with deep GCNs.**

In this paper, we present Dancer, an innovative PIM-based GCN accelerator. Dancer optimizes data movement during the inference process, significantly improving efficiency and reducing energy consumption. Specifically, we introduce a novel compressed graph storage architecture and a dynamic quantization technique to minimize data transfers at each layer of the GCN. Additionally, through a detailed analysis of weight dynamics changes, we propose a sparsity propagation strategy to further alleviate the computational and data transfer burden between layers. Experimental results demonstrate that, compared to current state-of-the-art methods, Dancer achieves 3.7× speedup, 7.6× energy efficiency, and reduces of 9.6× DRAM access on average.

***Index Terms*—Deep GCN, Processing-in-Memory, Sparsity, Quantization**

I. Introduction

Graph Convolutional Networks (GCNs) [1] are specialized deep learning models designed for processing graph-structured data, and they have been widely applied in areas such as social network analysis [2] and molecular property prediction [3]. Through aggregation and combination operations, GCNs can capture intricate relationships between graph vertices, which traditional CNNs and DNNs struggle to address directly.

However, traditional GCNs are often limited to shallow structures (e.g., two or three layers) due to challenges such as vanishing gradients and over-smoothing [4]. This limitation hinders their ability to capture complex graph information. With the emergence of techniques like residual connections [5, 6], deep GCNs have been developed. Increasing network layers enables deep GCNs to capture information beyond shallow GCNs' reach. Nevertheless, as the network's depth increases, deep GCNs face both computational and memory challenges. The inference process of deep GCNs requires frequent and substantial data migration between the computing unit and storage unit. Insufficient solutions to these challenges may result in a waste of computational and storage resources, thereby impeding the scalability and practicality of applications.

Various algorithms have addressed the challenges of GCNs, including GPU-based approaches such as PruneGNN [7], AccelGCN [8] and MaxK-GNN [9]. These methods achieve higher efficiency through optimized GPU scheduling. In addition, FPGA-based approaches [10–12] use extra FPGA units, achieving lower energy consumption. Despite GPUs having high-bandwidth memory, GCNs involve frequent random access [13], leading to non-negligible energy overhead and memory access costs. FPGA-based methods offer lower energy consumption and can optimize memory access patterns, but they still require reading data from external storage, which leads to significant bandwidth and latency issues.

As a representative method for GCNs, processing-in-memory (PIM) offers high bandwidth and fast speed with relatively low energy consumption. However, PIM faces challenges during computation [14, 15], leading to increased data movement. To address this issue, efficiently filtering out unnecessary data traffic has become a key problem. Inspired by recent research [7, 16] on sparsity propagation, our goal is to leverage its lower data migration and reduced computational load to simplify and accelerate the GCN process.

This paper introduces Dancer, a PIM accelerator for deep GCNs. Our primary objective is to reduce unnecessary data movement, minimize data migration, lower energy consumption, and enhance overall efficiency. Dancer comprises two key components: intra-layer and inter-layer processing. In the intra-layer component, we propose a novel Compressed Sparse Bitmap (CSB) format to store graph structures using less space and achieving higher parallelism. For dense weight and feature matrices, we employ a dynamic quantization algorithm that quantizes values into different modules based on vertex degrees and weight magnitudes. To further reduce data movement, we design a sparsity propagation algorithm between layers, creating sparsity by utilizing outlier values. We compare Dancer with representative baselines in terms of inference efficiency, DRAM access, and energy consumption. The results demonstrate that average inference efficiency is significantly improved by 3.7×, DRAM access is reduced by 9.6×, and energy consumption is decreased by 7.6× on average.

The contributions of this paper are as follows:

- A PIM architecture is proposed to reduce data migration in the deep GCN inference process, thereby effectively reducing energy consumption.
- A compressed sparse graph structure (CSB) and a quanti-

978-3-9826741-0-0/DATE25/© 2025 EDAA

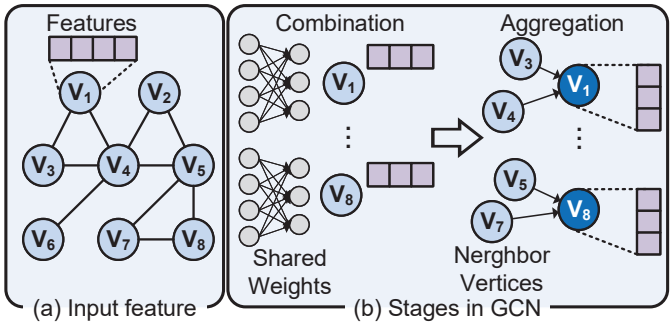


Fig. 1: Illustration of a GCN model.

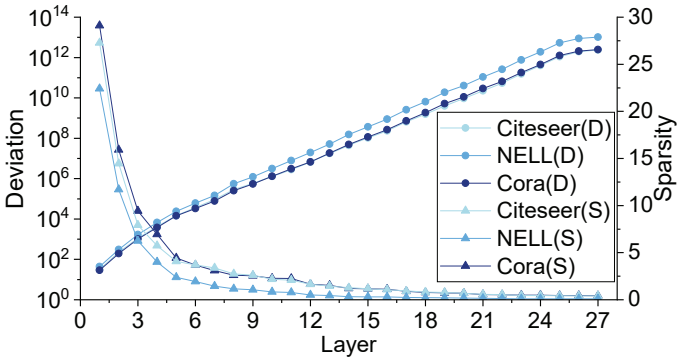


Fig. 2: Feature statistics in different layers. "D" for deviation and "S" for Sparsity.

zation method are proposed to reduce the graph's storage space. Additionally, a sparsity propagation strategy is adopted to minimize data migrations between GCN layers.
- Comprehensive experiments of the proposed method are conducted using widely recognized datasets and a general simulator in the field through comparative analysis with representative solutions.

The rest of this paper is organized as follows: Section II provides the background and discusses the motivation of this work. Section III presents our proposed Dancer in detail. Section IV introduces the experimental results. Finally, Section V concludes the paper and discusses future work.

II. Background and Motivation

A. Graph Convolutional Network

As shown in Figure 1, given the topology of a graph and a set of input features, a graph convolutional network (GCN) can produce the output features through the operations in the layers on the input features. In a common GCN, the output features of the l-th layer, X^{l+1}, can be computed from the previous layer, and this process can be expressed as:

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

where σ is a non-linear activation function (e.g., Relu). $\hat{A}$ is a normalized adjacency matrix derived from the input graph, and X^l and W^l represent the input features and weights of the l-th layer, respectively. As shown in Figure 1(b), the process of GCN can be divided into two stages: *combination* and *aggregation*. In each stage, the features of each vertex are updated. During the combination stage, a combination function updates the feature vector into a new one, which is then passed to the next stage. In the aggregation stage, each vertex collects information from its neighbour and generates a new feature vector using an aggregation function. For example, in Figure 1(b), vertex V_1 aggregates the features of vertex V_3 and V_4, while vertex V_8 aggregates the features of V_5 and V_7. The output of each layer serves as the input for the next layer. After multiple layers of iterations, the output features represent high-level information extracted from the input graph.

Due to issues like over-smoothing, most traditional GCNs are shallow, typically with fewer than five layers, making it challenging to uncover deep information. However, with the recent development of residual connections [4], deep GCNs have become feasible. Our proposed method, *Dancer*, is designed specifically for deep GCNs. Additionally, since A, W, and X are all matrices, the order of matrix multiplications can be rearranged, meaning the execution order of the combination and aggregation stages can also change. Previous research has shown that performing the combination first and then aggregation ($\hat{A} \cdot (X^l \cdot W^l)$) can significantly reduce computational cost. Therefore, our *Dancer* method adopts this "combination first" approach.

B. Processing in Memory

Processing-in-memory (PIM) is an emerging computational architecture that performs logic operations within memory arrays, thus avoiding frequent data transfers between memory and the processor. This approach enhances computational performance and reduces power consumption. To support PIM operations, processing units are embedded into selected memory banks, referred to as PIM banks. These banks are equipped with lightweight processing elements (PE) and small SRAM buffers, enabling local data processing without the need for constant data movement. By integrating processing units directly into the memory structure, the PIM architecture minimizes data movement, reducing latency and energy consumption during computation.

Leveraging the inherent storage characteristics of DRAM, the PIM architecture enables finer-grained parallel computation. However, PIM can only handle relatively simple, repetitive operations and is not capable of processing complex tasks like a central processing unit. These features make PIM particularly suitable for compute-intensive, memory-bound applications such as GCN [17] and Approximate Nearest Neighbor Search (ANNS) [18]. In this paper, we exploit the near-memory characteristics of PIM to significantly reduce data movement and lower energy consumption.

C. Motivation

Redundant data migration has led to severe energy consumption issues and reduced the efficiency of GCNs. We select three representative datasets, Citeseer [19], Cora [19], and NELL [20], and train a 27-layer deep GCN model [4]. As shown in Figure 2, we focus on the changes in the feature

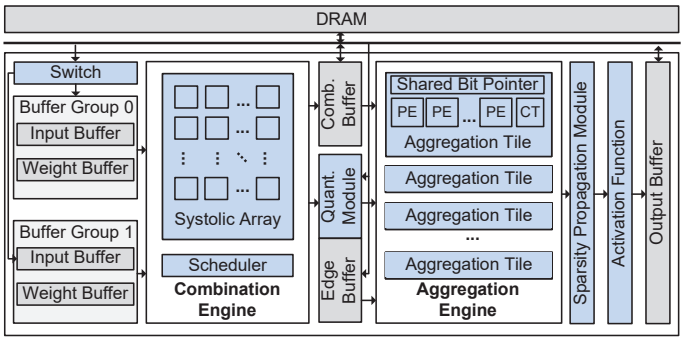


Fig. 3: The architecture of Dancer.

matrix at each layer during GCN computation. Figure 2 illustrates the changes in the standard deviation of the absolute values in the feature matrix during the GCN inference process. The standard deviation of the feature matrix is initially quite low, indicating a concentrated distribution. As the number of layers increases, the standard deviation grows exponentially. This indicates that the data in the GCN process becomes much more dispersed, with large values becoming larger and small values becoming smaller.

Figure 2 illustrates the exponential decline in sparsity with increasing GCN layers, nearing zero after the 10th layer. Combining this with the conclusions about the standard deviation, we find that the GCN process generates a large amount of marginal data. This marginal data is frequently transmitted and computed, resulting in significant overhead. Motivated by impact of marginal data on GCN computation and storage and to address the issue of low sparsity, we propose a PIM-based GCN accelerator.

III. Dancer Architecture

A. Overview

We provide a detailed explanation of Dancer, a software-hardware co-designed Deep GCN accelerator based on PIM architecture. At the software level, we designed a denser graph storage structure and employed dynamic quantization and sparsity propagation methods to reduce data transmission. At the hardware level, we developed buffer groups and aggregation tiles to offer higher parallelism.

B. Architecture Design

As shown in Figure 3, our proposed Dancer adopts a hybrid architecture, where the combination and aggregation phases are completed in the combination engine and aggregation engine, respectively. To distinguish between the functional modules and buffer modules, we colour the buffer modules in grey and the functional modules in blue. As mentioned in the previous section, to reduce the computational load, Dancer follows a strategy of combination first and then aggregation.

Initially, we designed a buffer rotation mechanism. To improve the spatial locality and parallelism of reading DRAM, we implemented a rotating buffer. When the combination engine reads inputs and weights from one buffer group, another buffer group performs data prefetching from DRAM. The data reading from the buffer group and the data prefetching from DRAM will operate alternately. Once computation begins, the dense inputs and weights in the buffer group are streamed into the systolic array of the combination engine as dense row data under the scheduler's control. After partial fragment multiplication is completed, accumulation takes place to generate partial sums or final results. Once a vertex's computation is complete, it enters the quantization unit, where data is quantized based on edge information and output weight information. This significantly reduces the size of the transmitted data.

Next, during the aggregation process, we perform aggregation calculations in the form of inner products. For each vertex that has completed the combination phase, the corresponding adjacency matrix A is prefetched into the edge buffer as a Compressed Sparse Bitmap (CSB). The combination results and the corresponding graph topology are sent to the aggregation tiles within the aggregation engine. To enhance parallelism within the tiles, we designed a shared bit pointer mechanism. Each Processing Element (PE) can retrieve the graph topology in constant time to complete the calculations. Unlike the combination phase, each tile computes a separate portion of the data. Each tile has a counter (CT) to accumulate the number of weights below the sparsity propagation threshold in the next stage. A sparsity propagation module creates sparsity after the aggregation phase to increase the weight of sparsity between layers. The results of the aggregation phase are stored in the output buffer after passing through the activation function.

C. Compressed Sparse Bitmap

Extensive data migration is particularly severe in deep GCNs. This not only introduces additional overhead but also degrades system performance. The previously used compression methods, such as Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC), are not dense enough and still result in some spatial inefficiency.

To address this issue, as shown in Figure 4(c), we propose a Compressed Sparse Bitmap (CSB) data structure to store the graph more compactly and reduce the size of data migration. In this method, the sparse matrix is stored in a row-first scheme. Rows are independent of each other, which allows for parallelism across rows. For each row in the sparse matrix, the CSB format consists of a row index, bitmap index pointers, bitmaps,

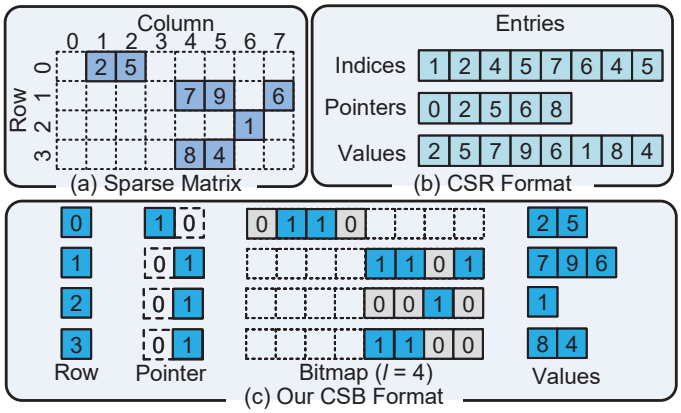


Fig. 4: Comparison between CSR format and CSB format.

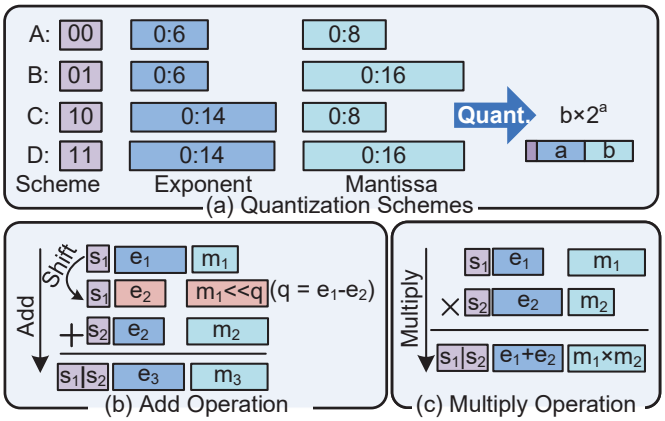


Fig. 5: Scheme and operations of quantization.

and the corresponding values. The row index represents the current row number of specific CSB. When data does not need to be stored in fragments, the uniqueness of the row index is maintained, meaning that each row of the sparse matrix corresponds to a single CSB. When sharding is necessary, CSBs from the same row are stored neighbouring to each other and are computed in parallel by aggregation tiles during the aggregation process. The bitmap index pointer and bitmap are used to identify the corresponding columns. Each bitmap has the same length l and is sequentially arranged in the row. The bitmap index pointer indicates the position of the current bitmap in that row. Since the matrix is sparse, most bitmaps are all-zero bitmaps. The bitmap index pointers corresponding to these all-zero bitmaps are omitted, and only the pointers to non-zero bitmaps are kept. Finally, the corresponding values are stored as an array at the end of the CSB. For example, consider the sparse matrix of size 4×8 shown in Figure 4(a). When using the CSR storage format, the data storage is illustrated in Figure 4(b), where a total of 13 integers are used to store the graph topology.

In contrast, when using the CSB structure with a bitmap length of 4 ($l = 4$), as shown in Figure 4(c). Values circled in dashed lines will be compressed and omitted. In this case, only 8 integers and 16 bits are used to store the graph structure. Compared to the original CSR format, this results in a 35% reduction in storage space. In addition, this row-independent compressed storage format works in conjunction with the aggregation tiles to maximize parallelism during the aggregation phase.

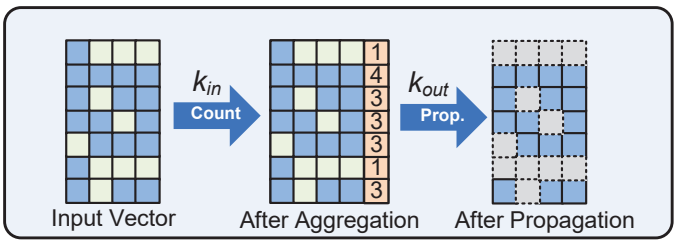


Fig. 6: Sparsity propagation illustration.

D. Dynamic Quantization

Traditional fixed-length storage for input and weight matrices is not suitable for deep GCNs. As analyzed earlier, deep GCNs contain a large number of boundary values, and storing these values in fixed length results in additional overhead. We propose dynamically compressing these values in a reasonable method, which can further reduce the size of the data stream while maintaining a certain level of accuracy. To address this issue, we develop a dynamic quantization strategy.

As shown in Figure 5(a), each value is represented by a triplet consisting of a scheme ID s, mantissa m, and exponent e. The two-bits scheme ID indicates the precision of current quantization. Each value is expressed as the product of the mantissa and the power of exponent base 2, which is $m \times 2^e$. The mantissa has lengths of 8 or 16 bits, while the exponent has lengths of 6 or 14 bits. This configuration meets almost all quantization requirements without causing overflow. For the multiply and add operations in GCNs, the computation rules after quantization are different. As shown in Figure 5(b), the add operation involves aligning the exponents of the values by shifting the smaller exponent to match the larger one and then accumulating the mantissa. The multiply operation is simpler, as shown in Figure 5(c); the exponents are summed, and the mantissa is multiplied. The resulting quantization precision is determined by taking the bitwise OR of the scheme IDs.

To minimize storage, we allocate precision based on the magnitude of the values and the degree of the vertices. For data with relatively small values, it is clear that we do not need large exponents for storage. Therefore, we use scheme A for values smaller than its maximum threshold and scheme B for those exceeding scheme A's limit but within scheme B's range. For other data, we calculate precision value Q using formula 2:

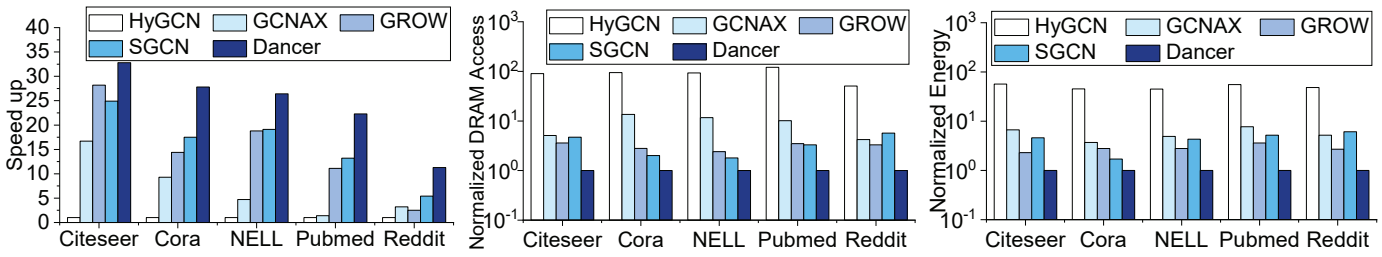
$$Q = \lambda \frac{d_i}{d_{max}} + (1 - \lambda) \frac{\log v_i}{\log v_{max}} \tag{2}$$

where d represents the degree of the corresponding vertex, d_{max} is the maximum vertex degree in the current layer, v is the current value to be quantized, $v_{max}$is the maximum value to be quantized, and λ is the degree parameter. For different workloads, we set different thresholds θ to distinguish quantization schemes. For values whose Q is less than θ, we apply scheme C for quantization, other values use scheme D.

E. Sparsity Propagation

Although we apply quantization to both the input and weight matrices, all of this data still needs to be fully processed during computation. Unstructured sparsity can only slightly reduce storage requirements, but it does not alleviate computational load or provide benefits for the computations in the next layer. Thus, more than our quantization techniques are required. Based on our earlier analysis of sparsity, we find that as the number of layers increases, the sparsity decreases exponentially. To address the issue of low sparsity and the resulting large computational workload, we propose a sparsity propagation method.

As shown in Figure 6, our goal is to enhance intra-layer feature sparsity and effectively propagate this sparsity to the



(a) Speed up (Normalized to HyGCN). (b) DRAM Access (Normalized to Dancer). (c) Energy Consumption (Normalized to Dancer).

Fig. 7: Evaluation on varying datasets.

next layer. To achieve this, we implement a sparsity propagation mechanism with threshold control for each row of features. The core of this mechanism lies in defining an intra-row sparsity threshold ratio k_{in} and inter-row sparsity threshold ratio k_{out}. We dynamically determine the sparsity filtering threshold based on this ratio. Considering the continuity of data distribution between adjacent layers , we adopt a forward-looking strategy. Specifically, we use the statistical data from the previous layer, where the k_{in}-th percentile of the current row serves as the sparsity filtering threshold. This computation is efficiently completed in memory and runs in parallel with the feature integration process in the current layer, ensuring non-blocking support for the aggregation phase.

During the aggregation phase, each aggregation tile maintains a counter to track the number of retained elements. As the aggregation proceeds, the counter is updated in real time, and once the specific tile task is completed, its state is passed to the next sparsity propagation stage. Then, in the sparsity propagation module, based on the inter-row sparsity retention standard, k_{out}, the system automatically evaluates and removes rows with a retention ratio below k_{out}. This step significantly enhances inter-layer sparsity. Finally, after completing this process of sparsity enhancement and propagation, all data undergoes activation function processing and is reloaded into DRAM in preparation for subsequent computation.

IV. Evaluation

A. Experimental setup

We selected five real-world graph datasets from different sources, with details provided in Table I. Cora, CiteSeer, and PubMed are citation network graphs from [19]. In these datasets, each vertex represents a document connected to other vertices, with a feature vector composed of a bag of words. NELL [20] is a knowledge graph from the Never Ending Language Learning (NELL) project, where each vertex has a one-hot encoded vector of length 61,278. Finally, Reddit [21] is a graph commonly used for GCN kernel evaluation. Each of these GCN applications adheres to a 27-layer architectural design.

To evaluate the performance of our method, we modified and extended the commonly used memory simulator DRAMSim3 [22] to support PIM operations for GCNs. We measured overall system energy consumption, latency, and the number of DRAM accesses through simulation. The computational logic was implemented in Verilog and synthesized using Synopsys Design Compiler with a 90nm library.

Dancer is compared with HyGCN [17], GCNAX [23], GROW [24] and SGCN [25]. HyGCN is a representative GCN accelerator that employs two hybrid processing engines to enhance performance. GCNAX is notable for its "combination-first" strategy, which optimizes the execution of graph computations. Similarly, GROW adopts the combination-first strategy but further improves upon it by introducing a tailored pipeline mechanism for GCNs. SGCN is an accelerator specifically designed for the inference of deep GCNs.

Considering the significant configuration differences between the accelerators being compared, we made the following adjustments to ensure a fair comparison. For all accelerators, we set the same size for the on-chip buffers. For HyGCN and SGCN, which also adopt heterogeneous architectures, we converted arithmetic operations into bitwise operations to maintain consistency. For GCNAX and GROW, which utilize unified compute units, we ensured during simulation that the area of their compute units is equivalent to the total area of the processing units in our Dancer architecture.

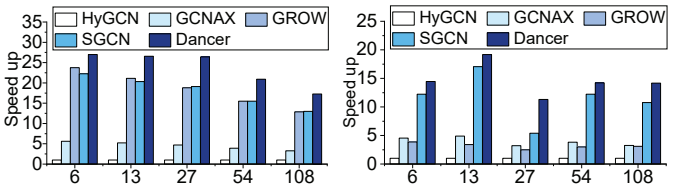
B. Results and Discussion

1) Speedup: Figure 7 (a) illustrates the speedup achieved across various benchmarks. The results clearly demonstrate that Dancer significantly outperforms the baseline models, achieving an average speedup of 24.3×, 3.4×, 1.6×, and 1.5× compared to HyGCN, GCNAX, GROW, and SGCN, respectively. This is due to the use of a compressed format for storing the graph and aggregation tile units, which helps improve parallelism and computational speed. For the relatively dense Reddit dataset, the computation time within the aggregation tiles is longer, resulting in a less significant performance improvement compared to the other four datasets.

2) Normalized DRAM Access: Figure 7 (b) presents the normalized DRAM access across the benchmarks. Dancer

TABLE I: Graph datasets.

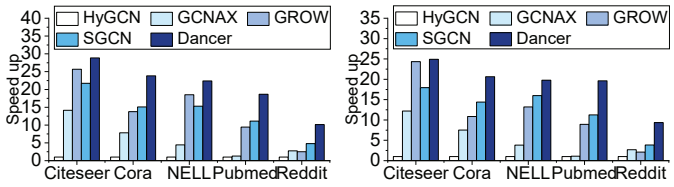
Dataset	# of vertices	# of edges	# of features	Average Degree
Citeseer	3,327	9,104	3,703	2.74
Cora	2,708	10,556	1,433	3.90
NELL	65,755	251,550	61,278	3.83
Pubmed	19,717	88,648	500	4.50
Reddit	232,965	114,615,892	602	491.99



(a) Speed up with various layers on NELL (Normalized to HyGCN).

(b) Speed up with various layers on Reddit (Normalized to HyGCN).

Fig. 8: Speed up with varying depths.



(a) Speed up on GIN model (Normalized to HyGCN).

(b) Speed up on GraphSAGE model (Normalized to HyGCN).

Fig. 9: Speed up on varying models.

shows substantial reductions in DRAM access, with an average reduction of 89.8×, 8.9×, 3.1×, and 3.5× relative to HyGCN, GCNAX, GROW, and SGCN, respectively. This is due to the use of a PIM-based compressed graph structure and sparsity propagation methods. These approaches reduce the amount of data that needs to be transferred, significantly decreasing the number of DRAM accesses.

3) Normalized Energy Consumption: Figure 7 (c) shows the normalized energy consumption for the evaluated benchmarks. Dancer achieves significant energy savings, reducing consumption by an average factor of 50.1×, 5.6×, 2.8×, and 4.4× compared to HyGCN, GCNAX, GROW, and SGCN, respectively. The improvement primarily stems from the reduced data movement achieved through Dancer's sparsity propagation and the corresponding reduction in computational load due to this sparsity.

4) Accuracy: As shown in Table II, we present a comparison of accuracy and evaluation length with FT32 and FT16. On the five datasets—CiteSeer, Cora, NELL, PubMed, and Reddit—we achieved space savings of 43.1%, 44.3%, 40.8%, 38.9%, and 37.3%, respectively, while incurring only 0.4%, 0.6%, 1.4%, 2.9%, and 2.7% accuracy losses. These improvements are attributed to the reasonable dynamic quantization methods implemented in our PIM-based architecture.

C. Sensitive Study

1) Number of layers: As an accelerator supporting deep GCNs, the depth of the GCN significantly affects the acceleration performance. We conducted a detailed experiment on the acceleration effects from 6 layers to 108 layers using the NELL and Reddit datasets. The results, shown in Figure 8, indicate that Dancer achieves significant acceleration, with average speedups of 19.1×, 4.5×, 1.8×, and 1.3× compared to HyGCN, GCNAX, GROW, and SGCN, respectively. Dancer demonstrates good scalability across GCNs of different depths.

2) Scalability: The scalability of an accelerator determines whether it can be widely applied across different domains. We also explored the general applicability of Dancer to Graph Isomorphism Network (GIN) [26] and GraphSAGE [21] models. As shown in Figure 9, Dancer demonstrated the best performance across five datasets for both the GIN and GraphSAGE models, achieving a speedup of 7.5× on GIN and 7.2× on GraphSAGE, respectively. Dancer demonstrates satisfying scalability on similar GIN and GraphSAGE models.

TABLE II: Accuracy and average length compared with FT32 and FT16

Dataset	Quantization	Accuracy	Average Length
Citeseer	FT32	71.32%	32
	Dancer	70.93%	18.21
	FT16	68.95%	16
Cora	FT32	80.06%	32
	Dancer	79.48%	17.83
	FT16	76.15%	16
NELL	FT32	73.84%	32
	Dancer	72.48%	18.93
	FT16	69.84%	16
Pubmed	FT32	76.13%	32
	Dancer	73.19%	19.54
	FT16	66.94%	16
Reddit	FT32	89.61%	32
	Dancer	88.91%	20.08
	FT16	83.45%	16

V. Conclusion

This paper presents Dancer, a PIM-based deep GCN accelerator that enhances inference efficiency by compressing graph structures and minimizing data immigration within and between layers. Its primary contributions include a novel compressed graph storage structure and a dynamic quantization method, allowing graph information to be stored in a smaller space. By designing sparsity propagation method, we significantly reduce data migration and computational workload. Experimental results demonstrate that Dancer can significantly reduce redundant memory accesses and achieve higher efficiency with lower energy consumption compared with representative schemes. Future work will consider the PIM heterogeneous design to enhance GCN training performance on optimized storage and computing architecture.

Acknowledgment

The work described in this paper is supported in part by NSFC (62122056, 62102263, and U23B2040), in part by Guangdong Basic and Applied Basic Research Foundation (2022A1515010180), in part by Shenzhen Science and Technology Program (RCJC20221008092725019, JCYJ20210324094208024, and 20220810144025001), in part by Guangdong Province Key Laboratory of Popular High Performance Computers (2017B030314073), in part by Guangdong Province Engineering Center of China-made High Performance Data Computing System and Shenzhen Key Laboratory of Service Computing and Applications. Yi Wang is the corresponding author.

References

[1] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks," in *ICLR*, 2017, pp. 1–14.

[2] J. Zhao, J. He, Y. Du, M. Tang, and X. Xu, "GNEA: A Novel GCN-based Network Embedding Algorithm for Semantic Social Network," in *ISSSR*, 2024, pp. 258–266.

[3] Z. Hao, C. Lu, Z. Huang, H. Wang, Z. Hu, Q. Liu, E. Chen, and C. Lee, "ASGN: An Active Semi-Supervised Graph Neural Network for Molecular Property Prediction," in *KDD*, 2020, pp. 731–752.

[4] G. Li, M. Muller, A. Thabet, and B. Ghanem, "DeepGCNs: Can GCNs Go as Deep as CNNs?" in *ICCV*, 2019, pp. 9267–9276.

[5] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks," in *KDD*, 2019, pp. 257–266.

[6] L. Zhang, X. Yan, J. He, R. Li, and W. Chu, "DRGCN: Dynamic Rvolving Initial Residual for Deep Graph Convolutional Networks," in *AAAI*, 2023, pp. 11 254–11 261.

[7] D. Gurevin, M. Shan, S. Huang, M. A. Hasan, C. Ding, and O. Khan, "PruneGNN: Algorithm-Architecture Pruning Framework for Graph Neural Network Acceleration," in *HPCA*, 2024, pp. 108–123.

[8] X. Xie, H. Peng, A. Hasan, S. Huang, J. Zhao, H. Fang, W. Zhang, T. Geng, O. Khan, and C. Ding, "Accel-GCN: High-Performance GPU Accelerator Design for Graph Convolution Networks," in *ICCAD*, 2023, pp. 01–09.

[9] H. Peng, X. Xie, K. Shivdikar, M. A. Hasan, J. Zhao, S. Huang, O. Khan, D. Kaeli, and C. Ding, "MaxkGNN: Extremely Fast GPU Kernel Fesign for Accelerating Graph Neural Networks Training," in *ASPLOS*, 2024, pp. 683–698.

[10] H. Zeng and V. Prasanna, "GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms," in *FPGA*, 2020, pp. 255–265.

[11] C. Zhao, Z. Dong, Y. Chen, X. Zhang, and R. D. Chamberlain, "GNNHLS: Evaluating Graph Neural Network Inference via High-Level Synthesis," in *ICCD*, 2023, pp. 574–577.

[12] P. Haghi, W. Krska, C. Tan, T. Geng, P. H. Chen, C. Greenwood, A. Guo, T. Hines, C. Wu, A. Li, A. Skjellum, and M. Herbordt, "FLASH: FPGA-Accelerated Smart Switches with GCN Case Study," in *ICS*, 2023, p. 450–462.

[13] H. Jin, D. Chen, L. Zheng, Y. Huang, P. Yao, J. Zhao, X. Liao, and W. Jiang, "Accelerating Graph Convolutional Networks through a PIM-Accelerated Approach," *IEEE Transactions on Computers*, pp. 2628–2640, 2023.

[14] J. Chen, Z. Zhong, K. Sun, C. Ma, R. Mao, and Y. Wang, "Lift: Exploiting Hybrid Stacked Memory for Energy-Efficient Processing of Graph Convolutional Networks," in *DAC*, 2023, pp. 1–6.

[15] J. Chen, Y. Lin, K. Sun, J. Chen, C. Ma, R. Mao, and Y. Wang, "GCIM: Towards Efficient Processing of Graph Convolutional Networks in 3D-Stacked Memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, pp. 3579–3590, 2022.

[16] C. Liu, X. Ma, Y. Zhan, L. Ding, D. Tao, B. Du, W. Hu, and D. P. Mandic, "Comprehensive Graph Gradual Pruning For Sparse Training in Graph Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 14 903–14 917, 2023.

[17] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "HyGCN: A GCN Accelerator with Hybrid Architecture," in *HPCA*, 2020, pp. 15–29.

[18] Z. Zhu, J. Liu, G. Dai, S. Zeng, B. Li, H. Yang, and Y. Wang, "Processing-In-Hierarchical-Memory Architecture for Billion-Scale Approximate Nearest Neighbor Search," in *DAC*, 2023, pp. 1–6.

[19] Z. Yang, W. Cohen, and R. Salakhudinov, "Revisiting Semi-Supervised Learning with Graph Embeddings," in *ICML*, 2016, pp. 40–48.

[20] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. Hruschka, and T. Mitchell, "Toward an Architecture for Never-Ending Language Learning," *AAAI*, pp. 1306–1313, 2022.

[21] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive Representation Learning on Large Graphs," *NIPS*, pp. 1025–1035, 2017.

[22] S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A Cycle-Accurate, Thermal-Capable DRAM Simulator," *IEEE Computer Architecture Letters*, pp. 106–109, 2020.

[23] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A Flexible and Energy-Efficient Accelerator for Graph Convolutional Neural Networks," in *HPCA*, 2021, pp. 775–788.

[24] R. Hwang, M. Kang, J. Lee, D. Kam, Y. Lee, and M. Rhu, "Grow: A Row-Stationary Sparse-Dense Gemm Accelerator for Memory-Efficient Graph Convolutional Neural Networks," in *HPCA*, 2023, pp. 42–55.

[25] M. Yoo, J. Song, J. Lee, N. Kim, Y. Kim, and J. Lee, "SGCN: Exploiting Compressed-Sparse Features in Deep Graph Convolutional Network Accelerators," in *HPCA*, 2023, pp. 1–14.

[26] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How Powerful are Graph Neural Networks?" in *ICLR*, 2019, pp. 1–14.