BEAST-GNN: A United Bit Sparsity-Aware Accelerator for Graph Neural Networks

Yunzhen Luo, Yan Ding, *Member, IEEE*, Zhuo Tang, *Member, IEEE*, Keqin Li, *Fellow, IEEE*, Kenli Li, *Senior Member, IEEE*, and Chubo Liu, *Member, IEEE*

***Abstract*—Graph Neural Networks (GNNs) excel in processing graph-structured data, making them attractive and promising for tasks such as recommender systems and traffic forecasting. However, GNNs' irregular computational patterns limit their ability to achieve low latency and high energy efficiency, particularly in edge computing environments. Current GNN accelerators predominantly focus on value sparsity, underutilizing the potential performance gains from bit-level sparsity. However, applying existing bit-serial accelerators to GNNs presents several challenges. These challenges arise from GNNs' more complex data flow compared to conventional neural networks, as well as difficulties in data localization and load balancing with irregular graph data. To address these challenges, we propose BEAST-GNN, a bit-serial GNN accelerator that fully exploits bit-level sparsity. BEAST-GNN introduces streamlined sparse-dense bit matrix multiplication for optimized data flow, a column-overlapped graph partitioning method to enhance data locality by reducing memory access inefficiencies, and a sparse bit-counting strategy to ensure balanced workload distribution across processing elements (PEs). Compared to state-of-the-art accelerators, including HyGCN, GCNAX, Laconic, GROW, I-GCN, SGCN, and MEGA, BEAST-GNN achieves speedups of 21.7×, 6.4×, 10.5×, 3.7×, 4.0×, 3.3×, and 1.4× respectively, while also reducing DRAM access by 36.3×, 7.9×, 6.6×, 3.9×, 5.38×, 3.37×, and 1.44×. Additionally, BEAST-GNN consumes only 4.8%, 12.4%, 19.6%, 27.7%, 17.0%, 26.5%, and 82.8% of the energy required by these architectures.**

***Index Terms*—Graph neural network, GNN accelerator, hardware accelerator, bit-serial computation, bit-level sparsity.**

Received 17 October 2024; revised 3 March 2025; accepted 28 March 2025. Date of publication 8 April 2025; date of current version 11 June 2025. This work was supported in part by NSFC under Grant 62172151, Grant 62202154, and Grant 62441234, in part by the Major Applied Basic Research Program of Hunan Province under Grant 2023GK2003, in part by the Science and Technology Innovation Program of Hunan Province under Grant 2024RC3076, and in part by the Fundamental Research Funds for the Central Universities. Recommended for acceptance by M. Rhu. *(Corresponding authors: Yan Ding; Chubo Liu.)*

Yunzhen Luo, Yan Ding, Zhuo Tang, Kenli Li, and Chubo Liu are with the College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China, and also with the National Supercomputing Center in Changsha, Hunan 410082, China (e-mail: luoyunzhen@hnu.edu.cn; ding@hnu.edu.cn; ztang@hnu.edu.cn; lkl@hnu.edu.cn; liuchubo@hnu.edu.cn).

Keqin Li is with the College of Computer Science and Electronic Engineering, Hunan University, Hunan 410082, China, also with the National Supercomputing Center in Changsha, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

Digital Object Identifier 10.1109/TC.2025.3558587

I. Introduction

Graph Neural Networks (GNNs) have gained significant attention due to their great potential to address graph-structured data. By iteratively aggregating information from neighboring nodes, GNNs capture intricate correlations, which facilitates the application of GNNs in various fields such as recommender systems [40], [25], point cloud segmentation [30], traffic forecasting [5], and autonomous driving [45].

However, as shown in Fig. 1(a), the irregular computational patterns in GNNs lead to low inference efficiency [39], [26]. This inefficiency prevents GNNs from meeting the demands for low latency and energy efficiency in edge computing applications, such as autonomous driving [45] and point cloud segmentation [30].

In response, there has been a growing body of research focused on developing hardware accelerators tailored for GNNs. State-of-the-art GNN accelerators, such as GCNAX [24], I-GCN [14], and GROW [21], primarily leverage value sparsity in adjacency and feature matrices, optimizing performance through sparse-dense matrix multiplication (SpDeGEMM).

While these GNN accelerators optimize performance through value sparsity, they introduce significant memory access overheads when applied to dense feature datasets [41]. To address this issue, we focus on exploiting the finer-grained bit-level sparsity inherent in GNNs, which allows for more efficient memory management and improved computational performance by reducing redundant memory accesses and unnecessary bit-level computations. First, Fig. 1(b) illustrates that the adjacency matrix, which represents edge connections using a single bit, exhibits over 99.8% bit sparsity. Second, recent studies have shown that quantizing GNN features and weights to lower, flexible bit-widths not only maintains accuracy but also significantly enhances efficiency by reducing memory and computational overhead [8], [35], [37]. As shown in Fig. 1(f), flexible bit-widths reduce memory accesses and computational costs by an average of 71%. Furthermore, as demonstrated in Fig. 1(c) and 1(d), the bit-level sparsity of quantized features and weights increases by an average of 29.4% and 53.8%, respectively, compared to their value-level sparsity.

Nonetheless, existing sparse bit-serial accelerators face limitations that make them unsuitable for GNNs, prompting us to identify and analyze three key challenges that hinder the effective adaptation of current acceleration techniques. First, the different computational phases in GNN layers complicates data

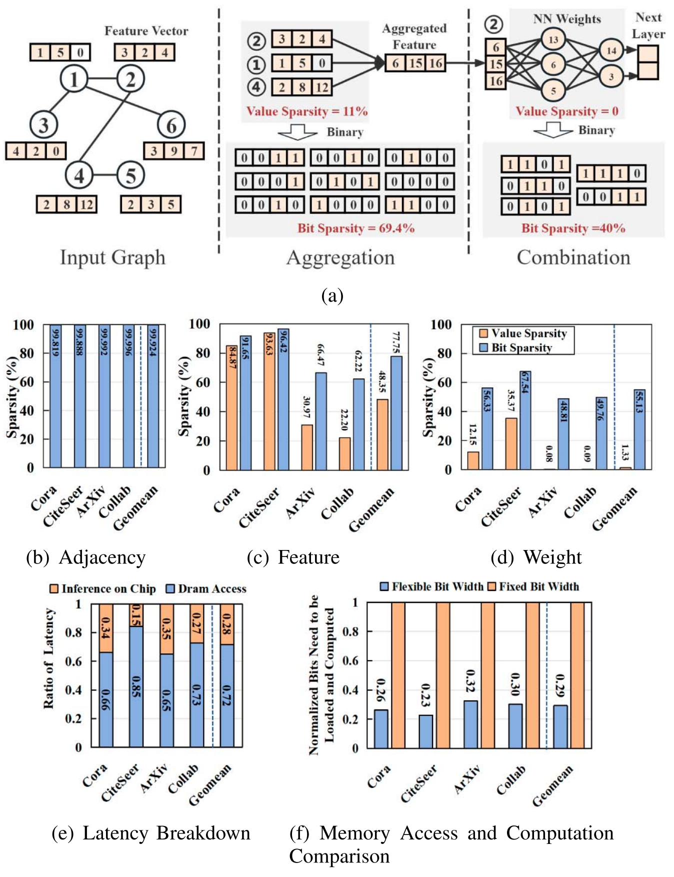


Fig. 1. (a) Illustration of a GNN layer highlighting value and bit-level sparsity. (b)–(d) Comparison of value-level sparsity and bit-level sparsity in the adjacency, feature and weight matrices, as applied in the GCN-SGQuant model [8]. (e) Latency breakdown of GCN-SGQuant on NVIDIA Tesla V100. (f) Comparison of memory access and computation for GCN-SGQuant, using a fixed bit width of 16 bits and flexible bit widths varying between 2, 4, 8, and 16 bits.

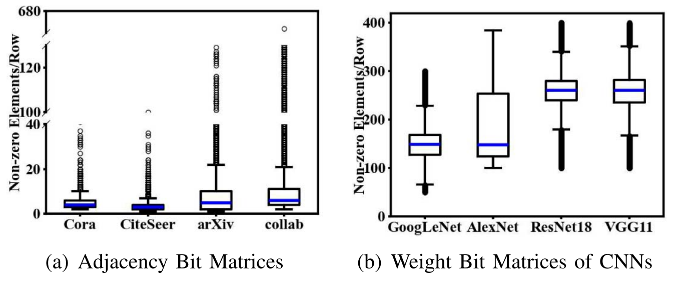


Fig. 2. Comparison of row-wise non-zero elements in (a) the adjacency bit matrix for graph datasets, and (b) the quantized CNN weight bit matrix processed through the Im2col transformation, In (a), the horizontal axis denotes the graph datasets, whereas in (b), it denotes the CNN models [43].

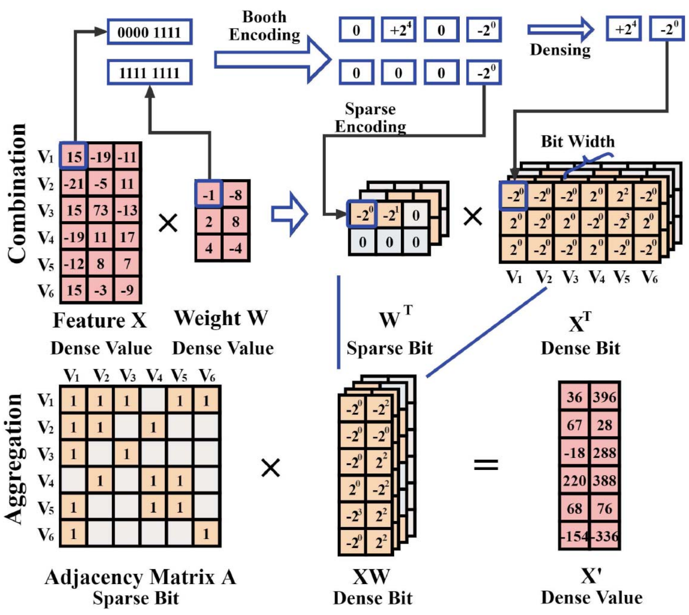


Fig. 3. The *Combination* XW and *Aggregation* $A(XW)$ phases can be unified into a matrix multiplication process of sparse and dense bit matrices by transforming the value matrices into bit matrices[0]. The booth encoding is to further reduce the number of bits that need to be computed [12].

flow, necessitating thorough optimization for efficient bit-serial computation [24]. Second, the irregular structure of graph data makes it challenging for existing sparse bit-serial accelerators to address data locality, resulting in redundant memory accesses [21]. Third, existing sparse bit-serial accelerators are primarily optimized for convolutional neural networks (CNNs), which exhibit a more uniform data distribution, allowing for more predictable processing patterns compared to GNNs, as shown in Fig. 2. In contrast, adjacency bit matrices in GNNs follow power-law distributions, resulting in highly imbalanced data workloads, creating severe load-balancing issues [13].

Therefore, we propose BEAST-GNN, a sparse bit-serial accelerator specialized for GNNs. First, as shown in Fig. 3, BEAST-GNN employs sparse coding for the weight bit-matrix similar to the adjacency matrix, consolidates *Aggregation* and *Combination* into a streamlined sparse-dense bit matrix multiplication with optimized data flow. Our insight is that since GNN weights are shared among nodes, they account for a small percentage of storage [20]. Thus, their bit-sparse matrices can be encoded and retained entirely on-chip, avoiding repeated memory accesses required for feature matrices. Second, to address irregular data access in the graph structure, BEAST-GNN performs a column overlap-based reorder operation on the adjacency matrix, effectively improving data locality. Third, to handle the power-law distribution of graph-structured data, BEAST-GNN uses a sparse bit counting strategy to achieve optimal load balance among PEs.

Compared to the existing state-of-the-art GNN and bit-serial accelerators, including HyGCN, GCNAX, Laconic, GROW, I-GCN, SGCN, and MEGA, BEAST-GNN achieves 21.7×, 6.4×, 10.5×, 3.7×, 4.0×, 3.3×, and 1.4× speedup, respectively. Moreover, it reduces DRAM access by 36.3×, 7.9×, 6.6×, 3.9×, 5.38×, 3.37×, and 1.44×, respectively, while consuming only 4.8%, 12.4%, 19.6%, 27.7%, 17.0%, 26.5%, and 82.8% of the energy by the aforementioned architectures. To

[0]Sparse bit matrices for the Laplacian matrix $\tilde{A}$ in GCN can be generated by applying the same processing used for W.

the best of our knowledge, BEAST-GNN is the first GNN accelerator that fully utilizes bit-level sparsity. Our contributions are summarized as follows:

- We conduct an in-depth analysis of the acceleration potential of state-of-the-art GNN models, considering value and bit sparsity across various datasets and data bit widths.
- We present BEAST-GNN, a GNN accelerator leveraging bit-serial computation and bit sparsity to enhance computational efficiency.
- We implement our architecture design in RTL and evaluate it using detailed microarchitectural simulations on four real-world graph datasets and three GNN models.

II. Background and Related Work

A. Graph Neural Network Structure

The GNN model consists of multiple layers, each updating node representations by aggregating features from neighboring nodes. GNNs efficiently capture complex graph structures, enabling them to outperform conventional methods that struggle with modeling the intricate relationships present in tasks such as recommendation systems [29], [25] and traffic forecasting [3], [5]. The computation of a GNN layer has two phases: *Aggregation* and *Combination*. Similar to conventional graph algorithms [9], [38], the structure of the graph is considered in the *Aggregation* phase, where each node gathers information from its neighbors to compute a message vector. This vector is then used by the target node to update its representation. The *Combination* phase involves a feedforward neural network that takes the aggregated message vector and the current node representation to produce a new node representation. These phases repeat across multiple layers, capturing complex node interactions within the graph. The forward propagation from layer l to $l+1$ is expressed in Eq. (1).

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

A represents the adjacency matrix of the graph. In this matrix, each row corresponds to a vertex and represents its connections with all other vertices in the graph. $X^{(l)}$ is a matrix containing input feature vectors of all vertices in layer l, where each column represents a feature, and each row denotes a vertex's feature vector. W is the GNN's model parameters, which are subject to training. σ is the activation function. It is worth noting that the weights W are shared among nodes in a GNN, so they account for a very small percentage of storage compared to the features X [8].

B. Graph Neural Network Accelerators

Due to the non-Euclidean structure of graphs, GNNs exhibit irregular memory access patterns and complex data dependencies compared to conventional neural networks [39], [13], [14], [16], [24], [23], [21], [44]. Consequently, many researchers are focused on designing specialized hardware accelerators for GNNs. To address the *Aggregation* and *Combination* heterogeneity in GNNs, HyGCN [39] introduces a hybrid architecture accelerator that considers the sparse and dense computation

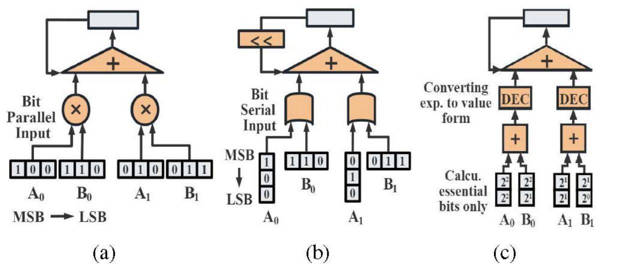


Fig. 4. Implementing a 3-bit Multiply-Accumulate operation in hardware using the inner product form. (a) Bit-parallel computation. (b) Bit-serial computation for A. (c) Bit-serial computation for A and B.

patterns in GCN operations. However, due to varying computational requirements of different datasets and models, this hybrid architecture may lead to load imbalances between the aggregation and combination engines. AWB-GCN [13] addresses the power-law distribution in graphs by implementing an online load-balancing mechanism for GNN accelerators. I-GCN [14] improves data locality by reordering node processing sequences on-the-fly, but real-time scheduling adds significant time and energy overhead. GCNAX [24] and EGCN [16] optimizes GCN computation by restructuring matrix multiplications as for loops and enhancing loop efficiency through reordering, expansion, and fusion. However, for large graphs, optimizing loop structures alone cannot overcome the challenges of irregular data access patterns. GROW [21] improves memory access and overall performance for high-degree nodes by using row-wise products and Metis-based graph partitioning. However, sparse connections between subgraphs can lead to increased DRAM accesses, hindering performance and energy efficiency. MEGA [44] enables flexible quantization of bit-width for graph node features through a hardware-software co-design. However, MEGA can only support its own quantization scheme and does not exploit bit sparsity.

C. Bit-Serial Computation

Bit-serial computation breaks operands into individual bits, performing operations sequentially on each bit [7], unlike bit-parallel computation, where all bits are processed simultaneously. Fig. 4 shows bit-parallel and two bit-serial computations using a 3-bit multiply-accumulate operation. In Fig. 4(b), the multiplier is input in bit-serial form, converting multiplication into a series of shift and add operations, adaptable to multipliers of any bit-width. The execution time depends on the bit-serial input length. In Fig. 4(c), both multipliers are input serially, turning multiplication into summation of exponents of essential bits, with the product decoded by the *DEC* after exponential addition. BEAST-GNN's PE design is optimized from the configuration in Fig. 4(c) to utilize bit sparsity. Bit-serial computation allows faster calculations and lower power consumption with varying bit widths compared to conventional fixed-width arithmetic circuits. It leverages bit sparsity by computing only non-zero bits, reducing energy requirements [32]. This approach has gained attention in deep learning acceleration for reducing memory bandwidth and computation needs in CNNs [2], [31], [34].

TABLE I
ACCURACY OF GCN, GIN AND SMP-GNN(SMP) ON CORA, CITESEER, ARXIV AND COLLAB DATASETS WITH VARIOUS QUANTIZATION METHODS AND BIT WIDTHS. GCN-SGQUANT QUANTIZES FEATURES FOR EACH LAYER AND NODE SEPARATELY WITHOUT QUANTIZING WEIGHTS. WE QUANTIFIED THE WEIGHTS AS INT16. ‘‘A.B.’’ REPRESENTS THE AVERAGE BIT-WIDTH OF THE FEATURE AND HIDDEN LAYER FEATURE

Dataset	GCN [8]		GIN [35]		SMP [37]		
	FP32	SGQuant	FP32	INT8	FP32	INT4	INT2
Cora (Accuracy %)	84.50 ± 1.83	83.37 ± 1.45 (A. B.=2.82)	73.60 ± 1.43	72.30 ± 2.73	82.80 ± 1.45	81.40 ± 1.52	74.60 ± 3.73
CiteSeer (Accuracy %)	74.39 ± 1.32	73.27 ± 2.30 (A. B.=2.49)	56.00 ± 1.67	56.20 ± 1.98	70.00 ± 0.89	68.63 ± 1.23	64.36 ± 1.44
ArXiv (Accuracy %)	70.29 ± 0.78	69.69 ± 1.22 (A. B.=3.028)	53.83 ± 1.16	46.38 ± 1.20	72.58 ± 0.98	68.58 ± 1.26	65.22 ± 2.65
Collab (Hit@100 %)	51.65 ± 1.80	50.65 ± 2.32 (A. B.=3.02)	32.81 ± 1.33	31.24 ± 1.24	53.33 ± 1.23	51.25 ± 1.82	47.32 ± 2.24

However, existing bit-serial neural network accelerators are designed to handle euclidean data structures, such as pictures or videos, without taking into account the irregularity of graph structures. As a result, conventional bit-serial architectures are ill-equipped to handle the high degree of sparsity present in adjacency matrices in GNNs, which poses a significant challenge for data locality and load balance.

III. OPPORTUNITIES AND CHALLENGES

In this section, we analyze the impact of bit width and bit sparsity on GNN computation across different bit widths. Then, we explore how the *Aggregation* and *Combination* phases can be unified at the bit level. Finally, we analyze the challenges of designing a bit-serial GNN accelerator that exploits bit-level sparsity.

A. Bit Width and Sparsity Analysis on GNNs

1) GNNs have flexible requirements for bit widths. First, during the GNN computation, the adjacency matrix A must be represented using only a single bit, i.e., either 0 or 1, to indicate the presence or absence of edge connections. Second, the low bit-width can also be advantageous for the weights W and feature X. As GNNs employ an iterative neighbor aggregation process for node information fusion, the slight loss of quantization can be largely mitigated [8], [35], [37]. Table I presents the accuracy performance of multiple GNNs with varying bit-widths and datasets, corresponding to different tasks. The specifics of the datasets and models utilized in the experiments can be found in Table II. We select two representative datasets with sparse features, Cora and CiteSeer, and two representative datasets with dense features, OGBN-ArXiv(ArXiv) and OGBL-Collab(Collab).

2) GNNs' matrix multiplication computation process exhibits significant bit sparsity. Fig. 5 demonstrates that, when using float32 and value sparsity as the baseline, the GCN-SGQuant can achieve a maximum speedup potential of 74× on the ogbn-collab dataset. For other datasets, the speedup potential of quantified GNN models ranges from 11× to 71× with the consideration of bit sparsity. This performance improvement can be attributed to three key factors. First, the adjacency matrix has a high degree of bit sparsity, reducing the number of computations required. Second, the quantized weights and features have reduced bit-widths, decreasing the computational load in bit-serial processing. Third, the quantization process further increases the bit sparsity of weights and features.

TABLE II
OVERVIEW OF DATASETS AND GNN MODEL INFORMATION UTILIZED IN THE EXPERIMENT. THE ACTIVATION FUNCTION IS RELU

Datasets		**Cora**	**CiteSeer**	**ArXiv**	**Collab**
# of Nodes		2,708	3,327	169,343	235,868
# of Edges		13,264	12,431	1,166,243	1,285,465
Feature Length		1,433	3,703	128	128
Feature Value Density		1.27%	0.85%	100%	100%
Hidden Feature Length	GCN	256	256	256	256
	GIN	128	128	256	128
	SMP	64	64	256	256
# of Layer	GCN	2	2	3	3
	GIN	5	3	2	2
	SMP	10	10	10	10
Hidden Feature Density(Avg.)	GCN	89.82%	90.39%	78.36%	68.86%
	GIN	52.70%	52.12%	51.29%	65.88%
	SMP	53.93%	51.30%	72.36%	69.23%

Although the weight and feature matrix exhibits high value density, a significant number of weights and features have values close to 0, resulting in a substantial amount of bit sparsity, particularly for integer and fixed-point number [2].

B. Unifying Aggregation and Combination at Bit Level

GNN layers involve two-phase matrix multiplication, AXW, where the computation order affects the number of Multiply-Accumulate (MAC) operations. Research on AWB-GCN [13] shows AXW can be computed as $(AX) \times W$ or $A \times (XW)$. The $A \times (XW)$ order offers two benefits: the XW matrix is generally smaller than X, reducing MAC operations. Additionally, sparse X matrices enable more efficient computations with dense W matrices via two consecutive SpDeGEMM operations, avoiding the underutilization and load imbalance issues found in hybrid architectures like HyGCN [39]. However, many datasets, such as Reddit [15], Yelp [42], Collab [19], and ArXiv [19], have dense X matrices. In these cases, as shown in Fig. 6, computing XW with SpDeGEMM requires sparse coding of dense X, leading to increased memory accesses.

We find that the weights exhibit high bit sparsity, as shown in Fig. 7. We can use the same sparse storage format for the weight bit matrix and the adjacency matrix, unifying

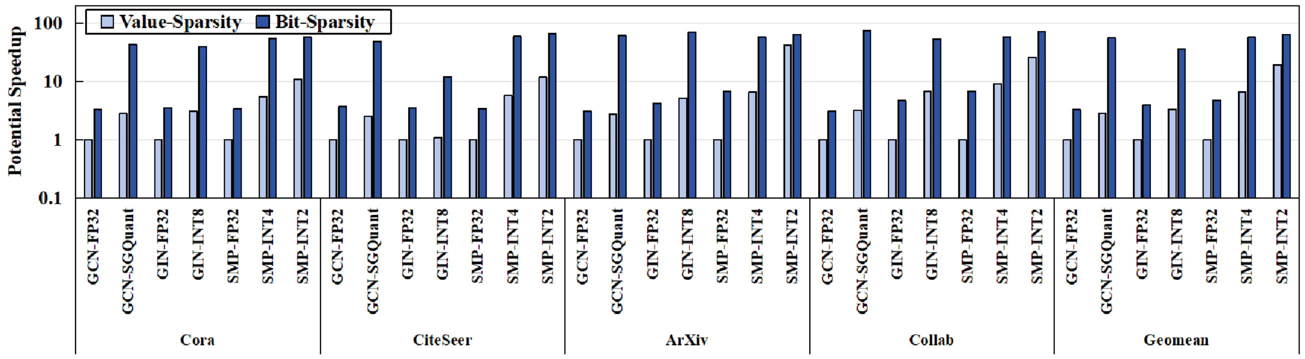


Fig. 5. Performance improvement potential gains from different bit widths, value sparsity, and bit sparsity. Value-Sparsity under Float32 (FP32) is used as the baseline. GCN-SGQuant quantizes features for each layer and node separately without quantizing weights. We quantified the weights as INT16.

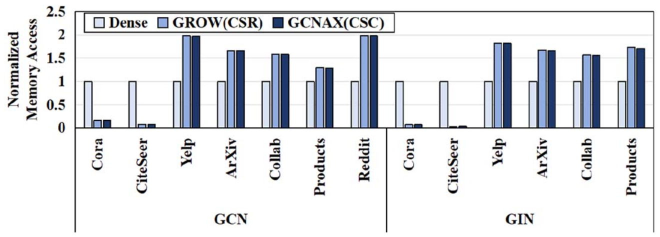


Fig. 6. Comparison of memory accesses for GCN and GIN models in dense, CSR, and CSC formats.

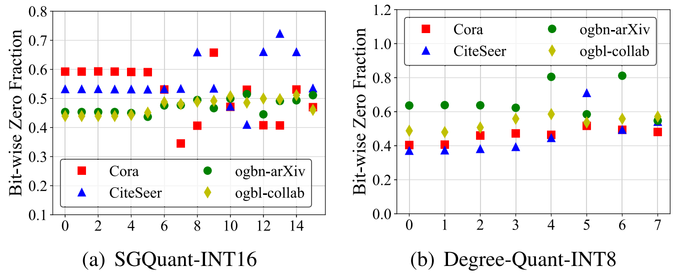


Fig. 7. The bit-wise zero fraction of weights. The horizontal axis represents the bit position within the binary representation of the weights.

Aggregation and *Combination* as bit-level SpDeGEMM. Additionally, the weights W are shared among nodes in a GNN, making each weight matrix small enough to be stored on-chip [20]. Thus, sparse coding of the weight bit matrix can be done on-chip, avoiding repeated memory accesses as required for feature matrices.

C. Challenges on Accelerator Design

While sparse bit-serial computation can enhance efficiency and reduce power consumption in GNNs, existing accelerators are not optimized for GNNs' unique computation processes, necessitating the design of specialized accelerators. However, designing such accelerators presents three key challenges.

First, GNN layers involve two heterogeneous computational phases: *Aggregation* and *Combination*, making their data flow more complex than conventional neural networks. Bit-serial computation introduces an additional bit-width dimension, complicating matrix multiplication for both phases [2], [31]. Therefore, the data flow must be carefully analyzed to optimize computational efficiency in accelerator design. Second, the irregular nature of graph data leads to non-contiguous memory access patterns during the *Aggregation* phase, causing frequent cache misses and increased DRAM accesses, which raise computational latency and energy consumption [39]. This irregularity hinders the effectiveness of sparse bit-serial accelerators designed for regular data structures. Third, the uneven distribution of connections in graph data, characteristic of power-law distribution, complicates load balancing among PEs [13]. In power-law distributed graphs, a small number of nodes have many connections, resulting in a few rows or columns in the adjacency matrix holding the majority of non-zero values. Without specific load-balancing strategies, this uneven workload distribution leads to overloading of some PEs, increasing computational latency and inefficiency.

IV. BEAST-GNN Architecture

In this section, we first present the fundamental design of BEAST-GNN. Second, we conduct a detailed analysis of BEAST-GNN's data flow and propose an optimized data flow design. Third, we introduce a graph partitioning algorithm to address the irregular memory access and sparsity challenges inherent in graph data. Finally, we propose a load balancing strategy to manage the computational challenges posed by the power-law distribution of graph data.

A. Architecture Overview

Fig. 8(a) presents an overview of the BEAST-GNN architecture, which comprises a controller, an encoder module, a on-chip SRAM module, a fetcher and dispatcher, a PE module, and an activator.

Within the encoders module, the booth encoder employs Radix-4 booth encoding [4] to encode weights and features following the value encoding scheme, which transforms values into exponential input form (for example, representing the number 1000 as 3, denoting 2^3). Despite this transformation into encoded exponential input form, we continue to refer to it simply as "Bit". Subsequently, the Sparse Encoder performs sparse encoding on the bit matrix derived from each set of

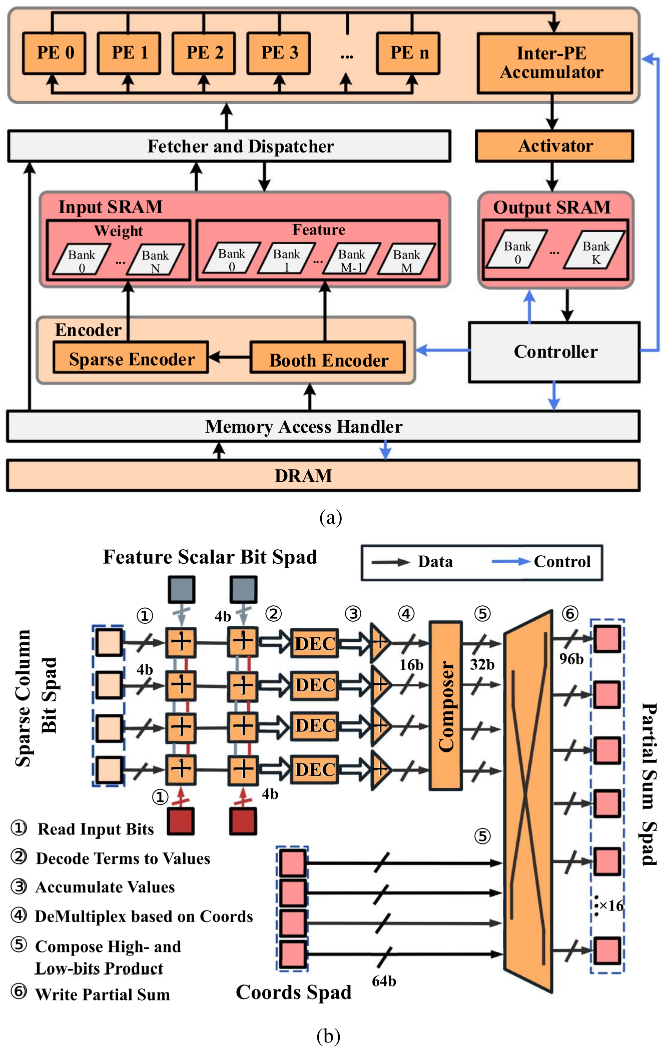


Fig. 8. (a) Architecture Overview. (b) Processing Element.

weight bits. In Section IV-C, we elucidate the rationale behind encoding the sparse bit matrix in CSC format through dataflow analysis.

The on-chip SRAM module comprises an input SRAM and an output SRAM. The input SRAM contains two areas: a weight area and a feature area. The fetcher and dispatcher retrieve computational data from the on-chip SRAM to perform the bit-serial SpDeGEMM, which is shown in Fig. 3. The fetcher and dispatcher assign columns of the adjacency matrix or weight bit sparse matrix, along with their corresponding features, to the PE module for column-wise product at bit level. Thus, the sparse matrices are saved in the CSC format. Since the column-wise product of PEs is executed in parallel and a long column may dispatch to more than one PE, the Inter-PE Accumulator is necessary to accumulate the results of the column-wise product. Finally, the Activator applies the activation function to the output.

B. Processing Element Design

Fig. 8(b) depicts the architecture of the Processing Element(PE), which comprises a *Sparse Column Bit (Scratch Pad)Spad*, a *Feature Scalar Bit Spad*, a *Coordinates(Coords) Spad*, 8 adders, a set of decoders, a de-multiplexer, and a *Partial Sum Spad*, where Bit Spads are implemented by registers, and other Spads are implemented by SRAM. For each round of calculation, the adder reads in 4 sparse column bits and 4 scalar bits①, with the sparse column bits shared among the 4 column adders and the scalar bits shared among the 4 row adders. Each adder is linked to two *Feature Scalar Bits*, guaranteeing that a group of *Sparse Column Bits* can calculate with two distinct *Feature Scalar Bits* during a calculation cycle, thus preventing the adder from idling. Once the adder has completed its calculations, the results of each row are passed to the decoder (*DEC*) to decode the bits as values②. The decoder is designed with reference to Laconic [31]. Following decoding, the results of one row of adder calculations need to be summed③. The term *Composer*④ and its nomenclature originate from Locanic [31]. Its primary function is to expand the exponent of Radix-4 Booth encoding, which inherently represents only 8-bit numbers, to accommodate higher bit representations. BEAST-GNN has opted to extend this capability to 16 bits. For example, assuming we have two 16-bit numbers a and b, they can be represented as:

$$\begin{aligned} a &= a_{hi} \times 2^8 + a_{lo}, \\ b &= b_{hi} \times 2^8 + b_{lo}. \end{aligned} \tag{2}$$

In Eq. (2), a_{hi} and a_{lo} respectively represent each 8 bits from the most significant to least significant in a. Similarly, b follows the same pattern. We then perform multiplication:

$$a \times b = (a_{hi} \times 2^8 + a_{lo}) \times (b_{hi} \times 2^8 + b_{lo}). \tag{3}$$

Expanding this yields a similar form:

$$\begin{aligned} a \times b = a_{hi} \times b_{hi} \times 2^{16} + (a_{hi} \times b_{lo} + a_{lo} \times b_{hi}) \times 2^8 \\ + a_{lo} \times b_{lo}. \end{aligned} \tag{4}$$

Composer's inputs and outputs are 16-bit and 32-bit respectively to prevent overflow during calculations.

Finally, summing up these five parts gives us the final product. Throughout the computation process, BEAST-GNN guarantees that the intermediate results do not exceed the specified limits, thereby preventing overflow. Moreover, for signed numbers, BEAST-GNN initially isolates the sign bit and subsequently establishes the sign based on the computed results. Thus, if required, the PE of BEAST-GNN can be extended to accommodate 32-bit in a similar manner.

The result of this summation in *Composer*④ is written to the *Partial Sum Spad*⑥ via the de-multiplexer⑤, according to the coordinates of sparse matrix. The coordinates of the sparse matrix are additionally stored in the *Partial Sum Spad*, a detail omitted from the figure for clarity.

C. Dataflow Analysis

As BEAST-GNN considers the *Combination* and *Aggregation* process of GNNs at the bit-level, similar to SpDeGEMM, it introduces an additional bit-width dimension compared to conventional SpDeGEMM. To analyze the data flow for

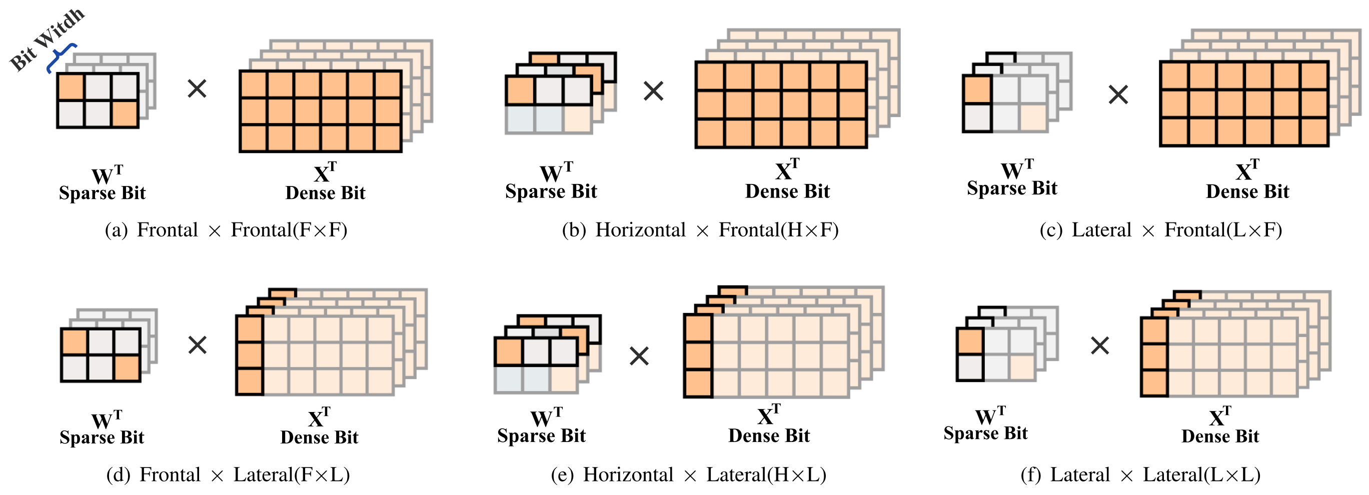

Bit Width
W^T Sparse Bit
X^T Dense Bit
(a) Frontal × Frontal(F×F)
(b) Horizontal × Frontal(H×F)
(c) Lateral × Frontal(L×F)
(d) Frontal × Lateral(F×L)
(e) Horizontal × Lateral(H×L)
(f) Lateral × Lateral(L×L)

Fig. 9. Data flow analysis of BEAST-GNN using tensor slicing: the *Combination* as an example. We did not include the Horizontal slice of X^T in the figure because it suffers from the same issue as the Frontal slice of X^T, which requires access to all nodes. Additionally, the Outer Product, for which the Horizontal slice of X^T is suited, has low output reuse and consumes a large amount of on-chip memory.

BEAST-GNN, we used a tensor slicing approach, as depicted in Fig. 9. This approach is based on the multiplication of each Booth-coded and sparsely coded weighted sparse bit matrix with the feature bit matrix.

Fig. 9(a)–9(c) depict the three data flow for frontal slices of X^T, where W^T is sliced in frontal, horizontal, and lateral slices, respectively. This slicing approach requires loading and encoding all node features to obtain a bit matrix of X, which leads to a significant amount of memory swapping in large graphs. The horizontal slice of X^T suffers from the same issue as the frontal slice of X^T, which requires access to all nodes. Additionally, the outer product, for which the horizontal slice of X^T is suited, has low output reuse and consumes a large amount of on-chip memory. The form of slicing W^T for horizontal slicing and X^T for lateral slicing, as shown in Fig. 9(e), is similar to the calculation process of an inner product. However, this method generates more index matching operations and suffers from the poor data reusability [33]. Additionally, in the aggregation phase, inner product either needs to aggregate all neighbor information of a node at once, which results in redundant memory access. The two forms depicted in Fig. 9(d) and 9(f) can utilize the column-wise product process, where the features of one node are loaded at a time and the columns of W^T can be calculated in parallel. Furthermore, some datasets' feature matrices such as Cora and CiteSeer are inherently sparse, and the column-wise product approach can skip some columns of the weight matrix depending on the feature matrix columns [33].

According to the above analysis, we give the specific data flow based on column-wise product of *Combination* and *Aggregation* in BEAST-GNN in Fig. 10. The arrows in Fig. 10 indicate the data flow, where the circled numbers indicate the order of the data flow. Arrows with the same number indicate that the process is parallel, while an arrow with multiple numbers indicates that the completion of the data flow requires waiting for the completion of multiple data flows. The highlighted portion indicates the smallest part of the computation process that needs to be loaded onto the chip. As shown in

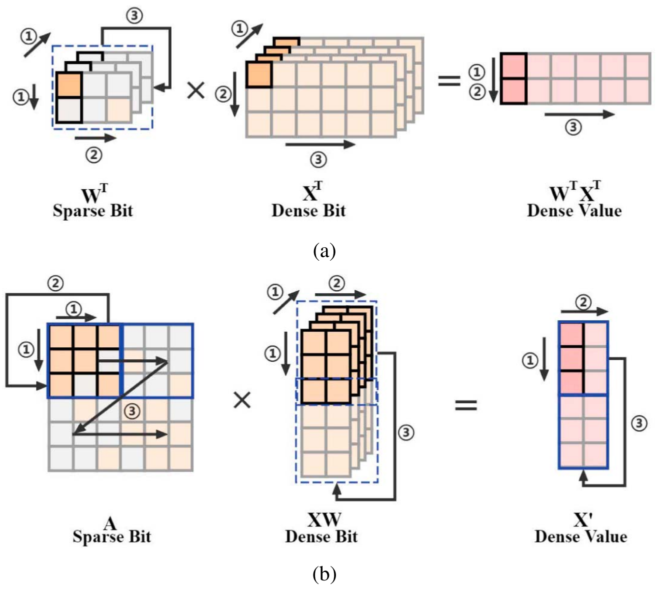


Fig. 10. (a) Data flow of *Combination*. (b) Data flow of *Aggregation*. The arrows in the figure indicate the data flow, where the circled numbers indicate the order of the data flow.

Fig. 9, during the *Combination* phase, denoted as $W^T \times X^T$, only the features of a single node or a subset thereof are required to be loaded onto the chip. Besides, leveraging the column-wise product property, computations for each feature dimension are executed concurrently across multiple PEs. Once the portion of XW resulting from the combination operation reaches the size of A or a predefined block size, the *Aggregation* process initiates computation, establishing a pipelined workflow.

D. Enhancing Locality by Column-Overlapped Reordering based Graph Partition

The order of matrix multiplication in BEAST-GNN is to perform W^TX^T first, followed by $A(XW)$. However, this leads to the inability to form a good pipeline when the matrix

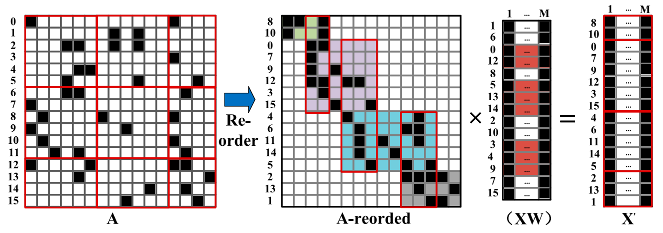


Fig. 11. The data localization of the adjacency matrix significantly improves after applying column-overlapped reordering, allowing for the reuse of features within the overlapped portion (highlighted in red in the figure) between adjacent column-overlapped blocks, where M denotes the dimension of a node's features. Based on the graph partitioning approach depicted in the figure for matrices A and A-reordered, the memory traffic for the column-wise product is measured to be $45M$ and $31M$, respectively.

multiplication of the two phases employs the same data flows. This is because the results obtained from the W^TX^T computation need to be transposed and then entered into the $A(XW)$ computation phase, which makes it impossible to immediately consume the columns obtained from the W^TX^T computation, as shown in Fig. 10. Furthermore, in the case of large graphs, the adjacency matrix must be partitioned during *Aggregation* to ensure proper loading of the corresponding node features in DRAM and on-chip memory, as depicted in Fig. 10(b). However, the data distribution obtained from direct partitioning is irregular, resulting in poor data localization and repeated loading of features, which leads to an increase in memory access.

To mitigate these issues, we employ a preprocessing method called HP_{BDCO} based graph partition to enhance data locality, where HP_{BDCO} denotes block-diagonal column-overlapped hypergraph partition [1]. As shown in Fig. 11, this method of matrix reordering has the following advantages: First, the reordering of the adjacency matrix results in a significant increase in the overall density over the original adjacency matrix. Second, since the neighbors of the target node are all in the closest block possible after the reordering, the times of feature loading required for aggregation are greatly reduced, which reduces the on-chip memory footprint and access to memory. Finally, the overlap of features can be utilized multiple times, which improves data reusability. Although the algorithm in [1] exhibits the above advantages, it is limited by the graph's diameter, which restricts partitioning efficiency. To overcome this, BEAST-GNN extends the approach by further partitioning large subgraphs after applying the algorithm in [1], enabling better scalability and performance on accelerator architectures. Additionally, as shown in Fig. 11, the row and column coordinates of the adjacency matrix from [1] are asymmetric post-partitioning, which introduces additional complexity. BEAST-GNN efficiently addresses this issue by ensuring that the accelerator can restore the hidden layers' order after aggregation, facilitating seamless computation for subsequent network layers.

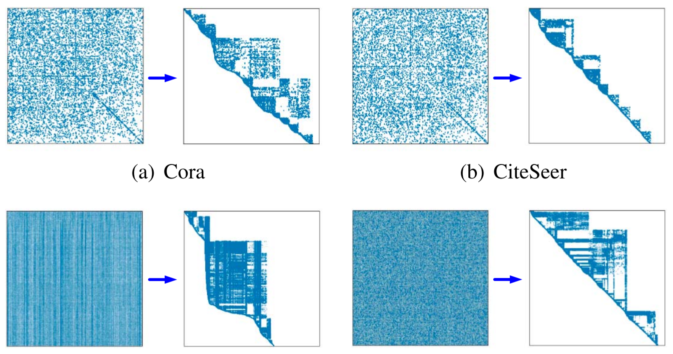
The graph partition process based on HP_{BDCO} is detailed in Algorithm 1. The inputs to Algorithm 1 include the original graph G, the graph's diameter K, and the target block size P. The diameter K is crucial as the HP_{BDCO} algorithm generates a number of blocks equal to K(For multiple connected subgraphs select the largest diameter of them). Initially, the algorithm converts the graph into a column-net hypergraph (line 1), then forms a column-overlapped graph (line 2) to produce K blocks. Subsequently, each block is further subdivided into smaller segments based on the target chunk size P to create the final partitioned graph G^* (lines 4-7). KyHyPar [17] is a graph partitioning tool specialized for hypergraphs, which can balance the graph partitioning efficiently.

Algorithm 1: HP_{BDCO} based Graph Partition.

```
Input : Original graph $G$, target block size $P$, graph
        diameter $K$.
Output: Column-overlapped-reordering graph $G^*$.
// The $HP_{BDCO}$ [1] employs hypergraph-based partitioning
and converts the graph into a column-net hypergraph.
1  $HyperG \leftarrow$ ColumnNet($G$);
2  $HyperG' \leftarrow HP_{BDCO}(HyperG, K)$;
3  initialize $G^*$;
4  for each $HyperG'_{sub} \in HyperG'$ do
       // $K_{sub}$ denotes the number of partitions for the
       subdivided subgraph $HyperG'_{sub}$. PARTITION method
       is KaHyPar [17]
5      $K_{sub} \leftarrow HyperG'_{sub}.num\_nodes/P$;
6      $G^*_{sub} \leftarrow$ PARTITION($HyperG'_{sub}, K_{sub}$);
7      $G^* \leftarrow G^* \cup G^*_{sub}$;
8  return $G^*$;
9  Function ColumnNet($G$: Graph):
10     initialize $HyperG$;
11     for each $node \in G$ do
           // A node's target nodes in a directed graph are
           represented as a column in the adjacency matrix.
12         $Hyperedge \leftarrow node.target\_nodes$;
13         $HyperG \leftarrow HyperG \cup Hyperedge$;
14 return $HyperG$;
```

The graph partitioning algorithm used in BEAST-GNN stands apart from the methods employed in I-GCN and GROW. I-GCN divides the graph into "islands" and "hubs", where each node is either part of an island or a hub. While this structure allows for the reuse of aggregation results within islands, it requires repeated searches of islands during inference, making it inefficient for large graphs. GROW, on the other hand, utilizes the METIS [22] algorithm to partition the graph based on densely connected regions, which increases intra-subgraph density but leads to greater sparsity between subgraphs. In contrast, BEAST-GNN's method optimizes subgraph connections by ensuring that overlaps in the adjacency matrix are minimized, concentrating effective elements along the diagonal. As shown in Fig. 12, BEAST-GNN significantly enhances data locality, reducing computation cost during the aggregation phase and improving overall processing efficiency.

The graph partitioning algorithm is performed offline with a complexity of $O(logP(V+E))$ [1], where P represents the number of subgraphs, V is the number of nodes, and E is



(a) Cora (b) CiteSeer

(c) ArXiv (d) Collab

Fig. 12. The effect of HP_{BDCO} based graph partitioning on the adjacency matrix of (a) Cora, (b) CiteSeer, (c) ArXiv, and (d) Collab. For better visualization, non-zero dots are enlarged.

the number of edges in the hypergraph, which is negligible compared to the overall computational overhead of the GNN. Specifically, the latency measurements for the graph partitioning method in a typical CPU configuration are compelling. On an Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, the method processes the datasets in 19ms, 22ms, 813ms, and 829ms, respectively. In contrast, the corresponding inference times for the GCN-SGQuant model on the same hardware are significantly higher: 301ms, 367ms, 6282ms, and 9528ms (all averages over 10 runs). Additionally, the graph reordering is performed just once and can be applied across all models, ensuring that it provides an efficient, reusable preprocessing step without adding repetitive overhead.

E. Load Balancing by Sparse Bit Counting Strategy

As illustrated in Fig. 13, although the power-law distribution of subgraphs diminishes following the partition based on HP_{BDCO}, an observable imbalance in neighbor distribution remains. Most subgraph nodes have less than 4 neighbors, i.e., the maximum number of sparse elements that can be accommodated by the PE of BEAST-GNN, while some subgraph nodes have a much larger number of neighbors than 4. To mitigate this and boost the operational efficiency of BEAST-GNN by reducing PE idleness, we have incorporated a *Sparse Bit Counting Strategy* into the fetcher and dispatcher module.

As shown in Fig. 14, the fetcher and dispatcher implement a Counter for each PE, tasked with tracking both the quantity of sparse bits and the extent of columns they cover within each PE. Each Counter fetches a column of sparse bits each cycle. When the PE dispatch cache contains fewer than four sparse bits and span less than two columns subsequent to dispatch, the sparse bit will be allocated to the PE Spads. Following this allocation, the respective feature will be retrieved based on the column associated with the sparse bit. Conversely, if the PE contains sparse bits surpassing these thresholds, they will be forwarded to the next PE Counter. Simultaneously, the content will be dispatched to the relevant PE for computation. Thus, each PE is capable of obtaining a maximum of 4 sparse bits from two distinct columns of the adjacency matrix in a round, owing to

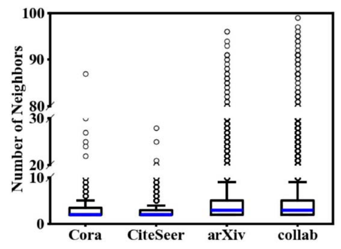


Fig. 13. The number of neighbors distribution in 4 datasets after HP_{BDCO} based graph partition. The target block size of HP_{BDCO} based graph partition is 100.

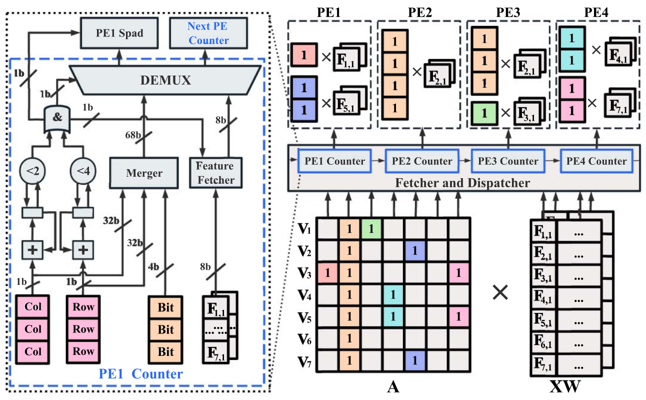


Fig. 14. Load Balancing by Sparse Bit Counting Strategy.

the PE design enabling concurrent feature computations from two different rows. For instance, the PE1 receive $F_{1,1}$ and $F_{5,1}$ simultaneously.

Moreover, BEAST-GNN incorporates an Inter-PE Accumulator, allowing for the subdivision of a column's elements within a block across multiple PEs for computation when the count exceeds 4. Subsequently, the results are accumulated across the PEs based on their respective coordinates.

F. Booth Encoder and Sparse Encoder

In Fig. 8(a), once the *Memory Access Handler* retrieves weights or features from memory to the chip, the subsequent step involves passing through the *Booth Encoder* to obtain encoded bit-serial computation elements. BEAST-GNN leverages the Radix-4 Booth Encoding technique [4], a method that significantly diminishes the quantity of displacement operations and partial products during bit-serial multiplication computations [31].

Upon obtaining the Booth-encoded matrix, BEAST-GNN proceeds to partition the matrix based on the relevant bit slices, routing it to the *Sparse Encoder* for sparse encoding. Adhering to the principles of Radix-4 Booth Encoding, each value element in the sparse matrix comprises 4 bits, encompassing 3 value bits and 1 sign bit. For instance, consider the 8-bit integer 27, represented in binary as 00011011, which, through Radix-4 Booth encoding, transforms into (none, 5, -2, 0), i.e., (none, 0101, 1010, 1000). Here, 5, -2, and 0 are allocated within the sparse encoding of the corresponding matrix based on their

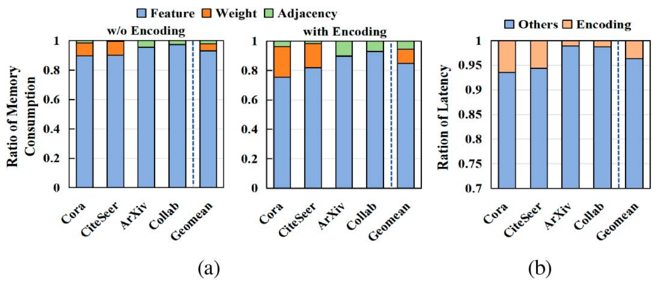


Fig. 15. (a) Ratio of memory consumption with and without encoding adjacency and weight matrix. (b) Latency overhead of encoding adjacency, weight and feature Matrix.

positions. Furthermore, owing to BEAST-GNN's implementation of column-wise product in its data flow, the sparse coding format aligns with the CSC format.

As shown in Fig. 15(a), applying Radix-4 encoding followed by bit-sliced CSC encoding to the weight and non-binary adjacency matrices introduces additional on-chip memory consumption. However, because the adjacency matrix has very high sparsity and the weight matrix is typically small, the overall increase in memory overhead, including the features, remains relatively minor. Based on these characteristics, BEAST-GNN applies Radix-4 and CSC encoding to the adjacency and weight matrices, while the feature matrix is encoded only with Radix-4, as shown in Fig. 15(b). The optimized data flow and pipeline design ensure that this process introduces minimal latency overhead.

V. Evaluation

A. Experimental Setup

Benchmark Graph Datasets and Models. The datasets and models utilized in our evaluation adhere to the configuration outlined in Table II. In addition, since BEAST-GNN can efficiently handle models quantized with arbitrary bit-widths, three quantized GNN models, GCN-SGQuant [8], Degree-Quant GIN(GIN-INT8) [35], and SMP [37], are selected for our evaluation experiments. BEAST-GNN can be applied to the mentioned GNN models due to its ability to efficiently map their computational processes to matrix multiplication. This flexibility not only allows BEAST-GNN to meet the computational demands of the above models but also enables seamless integration with other GNN variants, such as GraphSage [15] by simply adding a sampler. Furthermore, for more complex models like GAT [36], the computation of attention values $e_{ij} = \sigma(\tilde{W}[Wh_i||Wh_j])$ can also be mapped to matrix multiplication, where σ represents the activation function, W is the weight matrix, and h is the feature matrix.

Baseline Architectures. To benchmark BEAST-GNN against current state-of-the-art accelerators in terms of performance, energy efficiency and area, we selected HyGCN [39], GCNAX [24], GROW [21], Laconic [31], I-GCN [14], SGCN [41], MEGA [44] for comparison. HyGCN is a hybrid-architecture GNN accelerator containing an aggregation engine and a combination engine. GCNAX is a outer product

TABLE III
The Configurations of Compared Architectures

Accelerator	Computing Unit@1GHz	Area(mm^2)	Precision	Tiling or Graph Partition/Reorder
HyGCN*	16 MACs 4 SIMD16	2.085	16 bits	Tiling
GCNAX	32 MACs	2.264	16 bits	Tiling
Laconic⋆	8 × 4 LPEs	1.838	1-16 bits	–
GROW	32 MACs	1.959	16 bits	Graph Patition/Reorder
I-GCN	32 MACs	2.597	16 bits	Graph Patition/Reorder
SGCN*	16 MACs 4 SIMD16	2.332	16 bits	Tilling
MEGA†	4x8x32 BSEs 256 Aggre Units	2.136	1-16 bits	Graph Patition/Reorder
BEAST-GNN	64 PEs	1.759	1-16 bits	Graph Patition/Reorder

* 16 MACs for combination and 4 SIMD16 for aggregation.
⋆ The COO format encoding was applied to the adjacency and feature matrices [28].
† BSEs for combination and Aggre Units for aggregation.

based SpDeGEMM accelerator tailored for GNNs, which proposes a dynamically reconfigurable loop unrolling and tiling mechanism for *Aggregation*. GROW is a row-wise product based SpDeGEMM accelerator tailored for GNNs. Laconic is a complete bit-serial accelerator that accounts for bit sparsity, conducting multiplication computations when both multipliers are in bit-serial format. Laconic is designed for conventional neural networks and do not readily adapt to the inherent irregularity of graph structures. I-GCN is a GNN accelerator specifically designed to reduce redundant aggregation computations by utilizing online reordering. SGCN is a deep GNN accelerator that introduces a specialized encoding method for handling deep, dense node features. MEGA is a GNN accelerator that flexibly quantizes graph node feature bit-widths through a hardware-software co-design approach.

Methodology. The performance, energy efficiency and area of BEAST-GNN and baseline architecture are measured by using the following methods.

Performance. We have developed and implemented a cycle-accurate and execution-driven simulator to precisely measure execution time in terms of the number of cycles. The simulator operates based on the analyzed GNN models, graph datasets and the reordered adjacency matrices (refer to Table II), extracted utilizing PyTorch Geometric [10], KaHyPar [17], Metis [22], and OGB (Open Graph Benchmark) [19]. To ensure a fair comparison with HyGCN, GCNAX, GROW and Laconic, I-GCN, SGCN, MEGA, BEAST-GNN has been set up to deliver equivalent levels of computational throughput and off-chip memory bandwidth(128GB/s), alongside similar on-chip SRAM capacity(378KB). The Weight, Feature and Output SRAM of BEAST-GNN are divided into 16 banks. We initially set the maximum bit-width to 16 bits for all architectures is that the weight matrix of GCN-SGQuant and hyperparameters of SMP are 16 bits. Table III presents a concise overview of the crucial architectural parameters within BEAST-GNN's baseline configuration. Furthermore, to ensure a fair comparison, we evaluate the performance of BEAST-GNN, GROW, and SGCN on the GIN-INT8 and GIN-INT4 models with 8-bit and 4-bit precision, respectively. Among all fixed 16-bit baselines, GROW and SGCN outperformed the others.

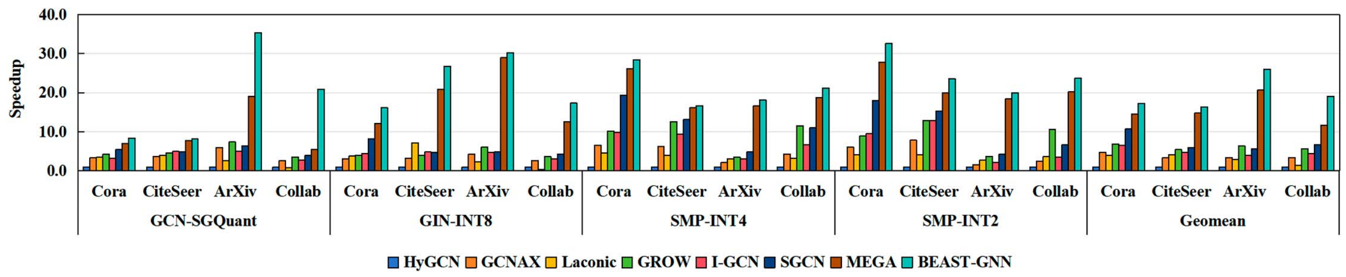


Fig. 16. Speedup over baseline architectures, normalized to HyGCN (higher is better).

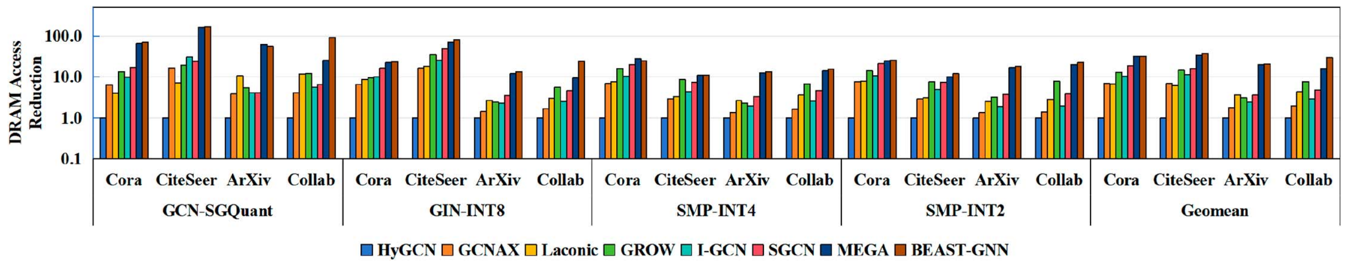


Fig. 17. DRAM access reduction, normalized to HyGCN (higher is better).

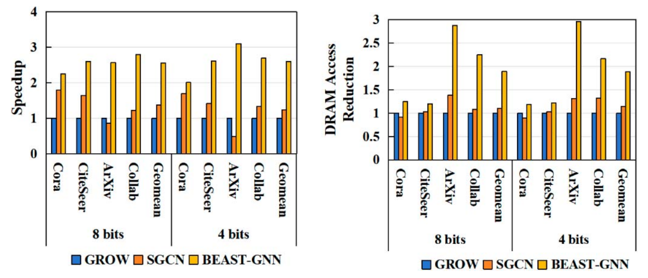
Area. We measure the modules' area of architectures by implementing it in RTL using Verilog. The RTL model is synthesized with Synopsys Design Compiler [6] targeting 1 GHz of operating frequency using the FreePDK 45nm standard-cell library [11].

Energy Efficiency. In assessing energy efficiency, we utilize a computing's energy model [18] to quantify energy per operation for arithmetic operations and off-chip DRAM accesses. To estimate the power and energy consumption related to on-chip SRAM usage, we employ CACTI [27] designed for the FreePDK 45nm process [11]. Given that the on-chip SRAM space significantly influences the area of chips, we leverage CACTI's leakage power to approximate static energy consumption.

B. Overall Results

Speedup. Fig. 16 shows that BEAST-GNN achieves an average speedup of 21.7×, 6.4×, 10.5×, 3.7×, 4.0×, 3.3×, and 1.4× compared to HyGCN, GCNAX, Laconic, GROW, I-GCN, SGCN, and MEGA respectively. This performance improvement can be attributed to BEAST-GNN's effective utilization of flexible bit-width and bit sparsity in the GNN computation process, as well as its enhanced data locality strategies in the aggregation process. Firstly, BEAST-GNN employs a bit-serial computation approach, which can selectively access and compute only the effective bit-width required in the GNN computation process, thereby improving efficiency. Secondly, BEAST-GNN makes full use of bit sparsity by sparse coding the bit matrices of the adjacency and weight matrices, and densing the essential bits of the feature matrices, effectively reducing the computational workload. Thirdly, BEAST-GNN enhances data locality through a column-overlapped reordering based graph partition strategy, which helps to pipeline the *Aggregation* and *Combination* process, ultimately reducing DRAM access. Finally, BEAST-GNN improves load balancing among PEs through the Sparse Bit Counting Strategy, further enhancing the overall computational efficiency. Furthermore, as shown in Fig. 18(a), BEAST-GNN maintains a significant advantage even under the same maximum bit-width, achieving 2.58× and 1.98× speedup over GROW and SGCN, respectively. This improvement is primarily due to the higher bit sparsity enabled by low-precision quantization and, as illustrated in Fig. 18(b), the effective reduction of DRAM access through BEAST-GNN's reordering algorithm. This evaluation highlights BEAST-GNN's ability to efficiently adapt across different bit-widths, reinforcing its advantage in lower-fixed-precision GNN processing.

BEAST-GNN's speedup is exceptionally impressive relative to HyGCN, given that HyGCN's $(AX)W$ execution order leads to a considerable surge in supplementary MAC operations. In addition, HyGCN employs a hybrid architecture, which can lead to low utilization of a certain part of the GNN when the two phases of the GNN are computationally unbalanced. In comparison, while Laconic is also capable of leveraging the flexible bit-width and bit sparsity inherent to the GNN computation process, it struggles to effectively the inherent irregularities of graph computations. This limitation introduces a significant number of additional DRAM access operations, which ultimately hinders Laconic's performance. For GROW, GCNAX and I-GCN, because of their default feature matrix X as a sparse matrix, it brings a lot of extra DRAM accesses for feature-dense datasets such as arXiv and Collab. As for SGCN, its encoding method for deep, dense feature matrices results in more pronounced acceleration effects for SMP with



(a) Speedup (b) DRAM Access

Fig. 18. Comparison of Speedup and DRAM Access Reduction for GROW, SGCN, and BEAST-GNN on the GIN Model. The 8-bits and 4-bits indicate the maximum bit-width of these architectures, corresponding to the GIN-INT8 and GIN-INT4 models (normalized to GROW).

10 layers, although it is limited by the overhead imposed by fixed bit-widths and heterogeneous architectures. MEGA is the architecture most similar to BEAST-GNN. However, its additional overhead, compared to BEAST-GNN, primarily arises from not fully exploiting bit sparsity.

DRAM Access. As shown in Fig. 17, BEAST-GNN achieves average DRAM access reductions of 36.3×, 7.9×, 6.6×, 3.9×, 5.38×, 3.37×, and 1.44× compared to HyGCN, GCNAX, Laconic, GROW, I-GCN, SGCN, and MEGA, respectively. These significant gains can be attributed to two key aspects of BEAST-GNN's design. Firstly, the bit-serial computation method employed by BEAST-GNN enables flexible access to data of arbitrary bit-widths, thereby reducing unnecessary DRAM accesses. Secondly, BEAST-GNN improves the density of effective blocks in the adjacency matrix through its column-overlapped reordering based graph partition strategy, which enhances data reuse and further reduces DRAM access.

In comparison, HyGCN, GCNAX, and SGCN apply tiling only to the adjacency matrix without sufficiently enhancing data locality, resulting in excessive DRAM accesses. Notably, SGCN introduces an encoding method for dense matrices, which reduces memory overhead. While Laconic can reduce DRAM accesses by leveraging flexible bit-width reads on features and weights, its inability to handle the irregularities inherent to the *Aggregation* process introduces significant overhead. Although GROW performs graph partitioning using Metis, the sparse connections between the partitioned subgraphs still result in irregular access patterns. I-GCN employs a graph reordering strategy designed to identify frequently aggregated node clusters to reduce redundant aggregation. However, its online search approach requires multiple rounds to discover effective clusters, thereby increasing overhead. MEGA, employing a dynamic node scheduling strategy called Condense based on METIS, reduces subgraph overlap but, for larger graphs, struggles with numerous sparse interconnections due to insufficient global information.

Energy Efficiency. As illustrated in Fig. 19, BEAST-GNN's average energy consumption is only 4.8%, 12.4%, 19.6%, 27.7%, 17.0%, 26.5%, and 82.8% compared to HyGCN, GCNAX, Laconic, GROW, I-GCN, SGCN, and MEGA respectively. The energy consumption of all these architectures includes the accessing off-chip memory. This is due to the fact that for the inherent irregularities of the graph during the *Aggregation* process, the majority of the energy is expended on off-chip data movement, rather than on-chip SRAM access or computation. BEAST-GNN's superior energy efficiency can be attributed to two key factors. Firstly, BEAST-GNN's bit-level computation optimizations effectively reduce the overall computational workload. Secondly, its enhanced data localization strategies, enabled by the graph partitioning techniques employed during the *Aggregation* process, contribute to the improved energy efficiency.

Area Analysis. As shown in Table IV, the total area of BEAST-GNN is 1.759 mm^2, synthesized using a 45nm standard-cell library. Compared to HyGCN, GCNAX, Laconic, GROW, I-GCN, SGCN, and MEGA, BEAST-GNN achieves reductions in chip area of 15.6%, 22.3%, 4.3%, 10.2%, 32.3%, 24.6%, and 17.6%, respectively. The majority of the area (78%) is occupied by on-chip SRAM, as the main computational units of BEAST-GNN are the adder and the decoder, which are significantly smaller in size compared to the MAC units used in HyGCN, GCNAX and GROW. In SGCN, an additional encoder designed specifically for dense feature matrices increases the area overhead compared to HyGCN. The online graph reordering module in I-GCN, which is used to reduce redundant aggregation, also occupies a large portion of the on-chip area. Additionally, the heterogeneous aggregation and combination architecture in HyGCN, SGCN, and MEGA introduces extra area overhead.

Ablation Study. To clearly elucidate the sources of BEAST-GNN's acceleration, we conducted an ablation study and quantified the impact of isolating our graph partitioning strategy and load balancing strategy, as shown in Fig. 20. The baseline BEAST-GNN employs the data flow described in Section IV-C and the hardware configurations presented in Table IV, without the application of any other strategies. We average the speedup of the 4 datasets. The experimental results demonstrate that the graph partition strategy alone achieves an average 1.47× speedup, while the load balancing strategy alone delivers an average 1.39× speedup. By applying both the graph partitioning strategy and the load balancing strategy, a combined speedup of 2.21× can be achieved.

Sensitivity Study. Given that BEAST-GNN effectively reduces DRAM accesses through its flexible bit-width access and graph partitioning strategy, we conduct a sensitivity analysis to further explore the impact of off-chip memory bandwidth on the performance of BEAST-GNN and other compared architectures. Fig. 21 illustrates the changes in speedup for BEAST-GNN and the compared architectures as the memory bandwidth is varied from 16GB/s to 256GB/s. All architectures have normalized their own performance with the 128 GB/s memory bandwidth, in order to highlight their robustness at different memory bandwidth levels. As shown in the figure, BEAST-GNN exhibits the least sensitivity to memory bandwidth, demonstrating the smallest change in performance when the DRAM bandwidth is varied. This indicates that

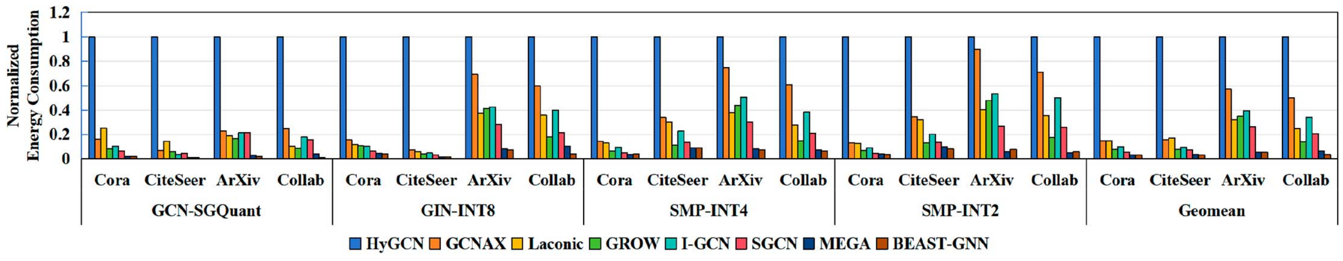


Fig. 19. Energy consumption, normalized to HyGCN (lower is better).

TABLE IV
THE CONFIGURATION AND BREAKDOWN OF BEAST-GNN

	Component	Number or Size	Area(mm^2)	Power(mW)
Processing Element	Bit Spad	4×2×64	0.036	7.014
	Coords Spad	4×2×64 (4KB)	0.028	10.257
	Adder	8×64	0.031	10.240
	Decoder	4×64	0.029	3.840
	Accumulator	8×64	0.048	41.856
	Composer	2×64	0.042	10.880
	DEMUX	64	0.156	7.667
	Partial Sum Spad	8×64 (6KB)	0.022	7.344
On-chip Memory	Weight SRAM	32KB	0.135	50.544
	Feature SRAM	256KB	0.894	356.976
	Output SRAM	80KB	0.299	121.928
Encoder	Booth Encoder	8×64	0.004	1.441
	Sparse Encoder	–	0.022	2.166
Others		–	0.013	5.462
Total		–	1.759	630.272

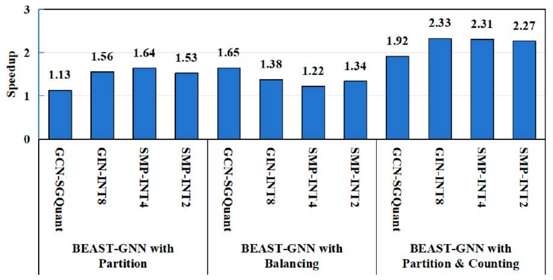


Fig. 20. BEAST-GNN's average speedup when incrementally applying our proposed optimization strategies (from left to right).

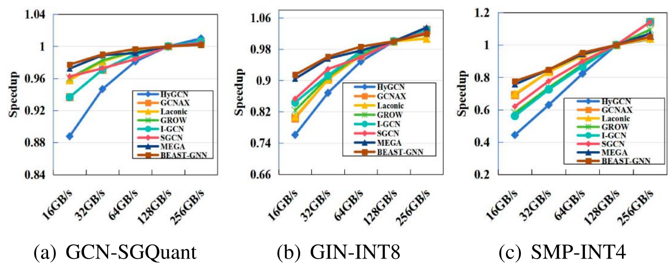


Fig. 21. Sensitivity to DRAM bandwidth (Normalized to 128GB/s).

BEAST-GNN possesses a high degree of robustness in terms of performance, outperforming the other architectures under diverse memory bandwidth conditions.

VI. CONCLUSION

In this paper, we propose an architecture for GNN inference called BEAST-GNN. The architecture is based on bit-serial computation, which enables the full utilization of bit-level sparsity during GNN computation, and is specially designed to optimize the data flow of GNNs. Additionally, the architecture further enhances the GNN computation process by employing a column-overlapped reorder based partition strategy and a sparse bit-oriented load balancing strategy, significantly improving the operational efficiency of GNNs. Compared to the existing state-of-the-art GNN and bit-serial accelerators, including HyGCN, GCNAX, Laconic, GROW, I-GCN, SGCN, and MEGA, BEAST-GNN achieves 21.7×, 6.4×, 10.5×, 3.7×, 4.0×, 3.3×, and 1.4× speedup, respectively.

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their constructive comments.

REFERENCES

[1] S. Acer and C. Aykanat, "Reordering sparse matrices into block-diagonal column-overlapped form," *J. Parallel Distrib. Comput.*, vol. 140, pp. 99–109, Jun. 2020.
[2] J. Albericio et al., "Bit-pragmatic deep neural network computing," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 382–394.
[3] A. Ali, Y. Zhu, and M. Zakarya, "Exploiting dynamic spatio-temporal graph convolutional neural networks for citywide traffic flows prediction," *Neural Netw.*, vol. 145, pp. 233–247, Jan. 2022.
[4] A. D. Booth, "A signed binary multiplication technique," *Quart. J. Mechan. Appl. Math.*, vol. 4, no. 2, pp. 236–240, 1951.
[5] X. Chen, J. Wang, and K. Xie, "Trafficstream: A streaming traffic flow forecasting framework based on graph neural networks and continual learning," 2021, *arXiv:2106.06273*.
[6] Synopsys Design Compiler. Accessed: Apr. 11, 2025. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTL Synthesis/Pages/default.aspx
[7] S. Lawson "VLSI signal processing: A bit-serial approach," *Electron. Power*, vol. 32, no. 2, p. 169, 1986.
[8] B. Feng, Y. Wang, X. Li, S. Yang, X. Peng, and Y. Ding, "SGQuant: Squeezing the last bit on graph neural networks with specialized quantization," in *Proc. IEEE 32nd Int. Conf. Tools Artif. Intell. (ICTAI),* Piscataway, NJ, USA: IEEE Press, 2020, pp. 1044–1052.
[9] G. Feng et al., "Aggregaterank: Bringing order to web sites," in *Proc. 29th Annu. Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2006, pp. 75–82.
[10] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," 2019, *arXiv:1903.02428*.
[11] Free PDK 45 nm open-access based PDK for the 45 nm technology node. Accessed: April. 10, 2025. [Online]. Available: https://eda.ncsu.edu/freepdk/freepdk45/

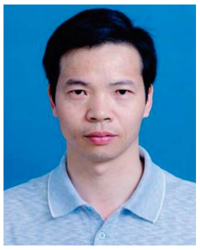
[12] Y. Gao, K. Zhang, and S. Jia, "BEM: Bit-level sparsity-aware deep learning accelerator with efficient booth encoding and weight multiplexing," in *Proc. IEEE 4th Int. Conf. Circuits Syst. (ICCS),* Piscataway, NJ, USA: IEEE Press, 2022, pp. 187–194.
[13] T. Geng et al., "AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO),* Piscataway, NJ, USA: IEEE Press, 2020, pp. 922–936.
[14] T. Geng et al., "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2021, pp. 1051–1063.
[15] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Adv. Neural Inf. Process. Syst.*, vol. 30, pp. 1024–1034, 2017.
[16] Y. Han, K. Park, Y. Jung, and L.-S. Kim, "EGCN: An efficient GCN accelerator for minimizing off-chip memory access," *IEEE Trans. Comput.*, vol. 71, no. 12, pp. 3127–3139, Dec. 2022.
[17] T. Heuer, P. Sanders, and S. Schlag, "Network flow-based refinement for multilevel hypergraph partitioning," *ACM J. Exp. Algorithmics*, vol. 24, pp. 1–36, Sep. 2019.
[18] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers (ISSCC),* Piscataway, NJ, USA: IEEE Press, 2014, pp. 10–14.
[19] W. Hu et al.,"Open graph benchmark: Datasets for machine learning on graphs," in *Proc. Adv. Neural Inf. Process. Systems*, vol. 33, 2020, pp. 22118–22133.
[20] L. Huang et al., "Epquant: A graph neural network compression approach based on product quantization," *Neurocomputing*, vol. 503, pp. 49–61, Sep. 2022.
[21] 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 *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA),* Piscataway, NJ, USA: IEEE Press, 2023, pp. 42–55.
[22] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, 1998.
[23] K. Kiningham, P. Levis, and C. Ré, "GRIP: A graph neural network accelerator architecture," *IEEE Trans. Comput.*, vol. 72, no. 4, pp. 914–925, Apr. 2022.
[24] J. Li, A. Louri, A. Karanth, and R. Bunescu, "GCNAX: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA),* Piscataway, NJ, USA: IEEE Press, 2021, pp. 775–788.
[25] Z. Li, Z. Cui, S. Wu, X. Zhang, and L. Wang, "FI-GNN: Modeling feature interactions via graph neural networks for ctr prediction," in *Proc. 28th ACM Int. Conf. Inf. Knowl. Manage.*, 2019, pp. 539–548.
[26] S. Liang et al., "ENGN: A high-throughput and energy-efficient accelerator for large graph neural networks," *IEEE Trans. Comput.*, vol. 70, no. 9, pp. 1511–1525, Sep. 2021.
[27] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Lab.*, vol. 27, 2009, Art. no. 28.
[28] A. Parashar et al., "SCNN: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Comput. Archit. News*, vol. 45, no. 2, pp. 27–40, 2017.
[29] M. J. Pazzani and D. Billsus, "Content-based recommendation systems," in *The Adaptive Web: Methods and Strategies of Web Personalization*. Springer, 2007, pp. 325–341.
[30] X. Qi, R. Liao, J. Jia, S. Fidler, and R. Urtasun, "3D graph neural networks for RGBD semantic segmentation," in *Proc. IEEE Int. Conf. Comput. Vis.*, 2017, pp. 5199–5208.
[31] S. Sharify et al., "Laconic deep learning inference acceleration," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 304–317.
[32] S. Smith, M. McGregor, and P. Denyer, "Techniques to increase the computational throughput of bit-serial architectures," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP),* vol. 12, Piscataway, NJ, USA: IEEE Press, 1987, pp. 543–546.
[33] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO),* Piscataway, NJ, USA: IEEE Press, 2020, pp. 766–780.
[34] W. Sun, Z. Zou, D. Liu, W. Sun, S. Chen, and Y. Kang, "Bit-balance: Model-hardware co-design for accelerating NNS by exploiting bit-level sparsity," *IEEE Trans. Comput.*, vol. 73, no. 1, pp. 152–163, Jan. 2024.
[35] S. A. Tailor, J. Fernandez-Marques, and N. D. Lane, "Degree-quant: Quantization-aware training for graph neural networks," 2020, *arXiv:2008.05000*.
[36] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2017, *arXiv:1710.10903*.
[37] S. Wang, B. Eravci, R. Guliyev, and H. Ferhatosmanoglu, "Low-bit quantization for deep graph neural networks with smoothness-aware message propagation," in *Proc. 32nd ACM Int. Conf. Inf. Knowl. Manage.*, 2023, pp. 2626–2636.
[38] T. Weng, X. Zhou, K. Li, K.-L. Tan, and K. Li, "Distributed approaches to butterfly analysis on large dynamic bipartite graphs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 2, pp. 431–445, Feb. 2023.
[39] M. Yan et al., "HYGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA),* Piscataway, NJ, USA: IEEE Press, 2020, pp. 15–29.
[40] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. 24th ACM SIGKDD Int. Conf. Knowl. Discovery & Data Mining*, 2018, pp. 974–983.
[41] 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 *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA),* Piscataway, NJ, USA: IEEE Press, 2023, pp. 1–14.
[42] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graphsaint: Graph sampling based inductive learning method," 2019, *arXiv:1907.04931*.
[43] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental network quantization: Towards lossless cnns with low-precision weights," 2017, *arXiv:1702.03044*.
[44] Z. Zhu et al., "Mega: A memory-efficient GNN accelerator exploiting degree-aware mixed-precision quantization," in *Proc. IEEE Int. Symp. High-Perform. Comput. Archit. (HPCA),* Piscataway, NJ, USA: IEEE Press, 2024, pp. 124–138.
[45] X. Zou, K. Li, Y. Li, W. Wei, and C. Chen, "Multi-task y-shaped graph neural network for point cloud learning in autonomous driving," *IEEE Trans. Intell. Transp. Syst.*, vol. 23, no. 7, pp. 9568–9579, Jul. 2022.

Yunzhen Luo received the B.E. degree in software and engineering from Hefei University of Technology, in 2017, and the M.E. degree from the Central South University, in 2021. He is currently working toward the Ph.D. degree in computer science and technology with Hunan University, China. His research interests include edge computing and computer architecture.

Yan Ding (Member, IEEE) received the Ph.D. degree in computer science from Hunan University, China, in 2021. Currently, he is an Assistant Professor with Hunan University. His research interests include parallel computing, mobile edge computing, big data, artificial intelligence, and architecture. He has published eight papers in journals and conferences, including Design Automation Conference, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, *Journal of Parallel and Distributed Computing*, *Computers & Security*, and the 17th IEEE International Symposium on Parallel and Distributed Processing with Applications (IEEE ISPA 2019). He received the Outstanding Paper Award in the 17th IEEE ISPA.

Zhuo Tang (Member, IEEE) received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2008. Currently, he is a Professor with the College of Computer Science and Electronic Engineering, Hunan University. He is also the Chief Engineer with the National Supercomputing Center in Changsha. He has published almost 120 journal articles and book chapters. His research interests include distributed computing system, cloud computing, and parallel processing for big data, including distributed machine learning, security model, parallel algorithms, and resources scheduling and management in these areas. He is a member of ACM and CCF.

Keqin Li (Fellow, IEEE) received the B.S. degree from Tsinghua University, in 1985, and the Ph.D. degree from the University of Houston, in 1990, both in computer science. He is a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University, China. He has authored or co-authored more than 1110 journal articles, book chapters, and refereed conference papers. He holds nearly 75 patents announced or authorized by the Chinese National Intellectual Property Administration. He is listed in ScholarGPS Highly Ranked Scholars (2022–2024) and among the top 0.002% of over 30 million scholars worldwide based on composite scores for research productivity, impact, and quality in the recent five years. He is an AAAS Fellow, an AAIA Fellow, and an ACIS Founding Fellow. He is a member of Academia Europaea.



Kenli Li (Senior Member, IEEE) received the M.S. degree in mathematics from the Central South University, China, in 2000, and the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003. He was a Visiting Scholar with the University of Illinois at Urbana-Champaign, from 2004 to 2005. He is a Full Professor of computer science and technology with Hunan University. He has published more than 300 papers in international conferences and journals. His research interests include parallel and distributed processing, supercomputing and cloud computing, high-performance computing for Big Data and artificial intelligence, etc. He is currently served on the Editorial Boards of IEEE TRANSACTIONS ON COMPUTERS. He is an Outstanding Member of CCF.

Chubo Liu (Member, IEEE) received the B.S. and Ph.D. degrees in computer science and technology from Hunan University, China, in 2011 and 2016, respectively. Currently, he is a Full Professor of computer science and technology with Hunan University. He has published over 40 papers in journals and conferences such as IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, IEEE TRANSACTIONS ON CLOUD COMPUTING, IEEE TRANSACTIONS ON MOBILE COMPUTING, IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, IEEE INTERNET OF THINGS JOURNAL, ACM *Transactions on Modeling and Performance Evaluation of Computing Systems, Theoretical Computer Science*, ISCA, DAC, and NPC. He won the IEEE TCSC Early Career Researcher (ECR) Award in 2019. He is a member of ACM.