OCO-GAT: An Accelerator for Graph Attention Network with Optimized Calculation Order

Qi Liu, Wenjin Huang, WenLu Peng, Yihua Huang
School of Electriconics and Information Technology, Sun Yat-sen University, Guangzhou, Guangdong, China
Email: {liuq295, hwenjin, pengwlu}@mail2.sysu.edu.cn, huangyih@mail.sysu.edu.cn

***Abstract*—The Graph Attention Network (GAT) introduces an attention mechanism to focus on the most significant aspects of the data and utilizes weighted sums for aggregation. As a result, GAT exhibits superior performance in tasks involving graph data compared to previous Graph Neural Networks (GNNs). However, this also introduces a more complex computational process and stronger data dependencies, making previous GNN accelerators inadequate for GAT inference. Therefore, we propose an optimized calculation order for GAT along with the corresponding accelerator architecture, OCO-GAT, specifically tailored for GAT inference, which includes efficient pipeline design. Additionally, we introduce a distributed fine-grained on-chip storage scheme, ensuring computational parallelism while mitigating significant growth in storage resource consumption. We deployed OCO-GAT on Xilinx Alveo U250 FPGA and the experimental results demonstrate that optimized calculation order reduces the workload of division operations by an average of 17.2% across four datasets. OCO-GAT achieves high energy efficiency and improves inference performance from 1.2× to 301.1× compared to CPU, GPU, and three peer works.**

***Index Terms*—GNN, GAT, accelerator, FPGA, Graph**

I. Introduction

Graph Neural Networks (GNNs) have become powerful tools for managing graphs, a typical type of non-Euclidean data [1]. GNNs have shown considerable advantages over traditional neural networks in diverse applications, including social networks [2], [3], recommendation systems [4], [5], transportation networks [6], and the prediction of chemical molecular structures [7], [8]. Unlike earlier GNN models such as Graph Convolutional Networks (GCNs), Graph Attention Networks [9] incorporate an attention mechanism [10] into the original Combination and Aggregation stages. This mechanism evaluates the relevance of various neighbors of a node by employing attention coefficients, enabling the model to concentrate on the most pertinent aspects of the data. By aggregating neighbors using weighted sums, GAT enhances its ability to represent complex relationships within graphs.

The pursuit of improved performance introduces complexity in computational processes. GAT involves hybrid characteristics with operations like division and softmax. During inference, the Aggregation and part of the Attention stages heavily rely on graph adjacency, requiring irregular and sparse memory access. In contrast, the Combination stage and another part of the Attention stage need more regular but denser memory access. These characteristics pose challenges for CPUs and GPUs, the predominant computing architectures. CPUs lack computational power for dense calculations, leading to latency. GPUs face inefficiencies due to irregular memory access, reducing computational efficiency and increasing power consumption. In contrast, advanced FPGAs integrate ample resources, including storage, computation, logic, and bus resources. They excel in parallel computing, offering flexibility to handle GAT's complex demands. This establishes FPGAs as the ideal platforms for speeding up GAT inference.

To meet the challenges of efficient inference of GAT , We proposed an architecture OCO-GAT based on FPGA. "OCO" stands for the "Optimized Calculation Order". Our contributions in this paper are as follows:

1) Without modifying the original GAT algorithm, we reordered its calculations to decompose the complex computation, minimize division operations, and lessen data dependencies. Compared to the original GAT, the **GAT with optimized calculation order** averaged only 17.2% division operations across four datasets.
2) We devised **an accelerator tailored for GAT with the optimized calculation order**. Our design employs graph partition method to divide a complete graph into slices and sub-slices and exploits parallelism among sub-slices.
3) We utilized **efficient pipelined architecture** to exploit **parallelism between node pairs** during GAT's Attention and Aggregation stages. This design enables parallel computation of node pairs, minimizing pipeline stalls between tasks of different source nodes.
4) We introduced **a distributed fine-grained storage scheme** to enable computational parallelism while limiting the growth of on-chip storage resource usage.

II. Background

A. Graph Attention Network

The original calculation process of the GAT is as follows.

The Combination stage transformed the original node feature $h_i \in \mathbb{R}^{F\times 1}$ by the weight matrix $\mathbf{W} \in \mathbb{R}^{F'\times F}$:

$$h'_i = \mathbf{W}h_i \tag{1}$$

Then execute the Attention stage. The attention weight a_{ij} is calculated through $\mathrm{softmax}$. Where $\mathbf{a}^T \in \mathbb{R}^{1\times 2F'}$ represents the transpose of attention kernel, ij represents a node pair of a source node and a target node, i represents the source node, j represents the neighbor node of the source node (the target node). $||$ stands for concatenation operation, $\mathrm{LeakyReLU}$ stands for non-linear activation function.

$$a_{ij} = \frac{\exp(\mathrm{LeakyReLU}(\mathbf{a}^T[h'_i||h'_j]))}{\sum_{k\in N_i}\exp(\mathrm{LeakyReLU}(\mathbf{a}^T[h'_i||h'_k]))} \tag{2}$$

The last stage is Aggregation. Compute the final embedding z_i. σ denotes the activation function.

$$z_i = \sigma(\sum_{j \in N_i} a_{ij} h'_j) \tag{3}$$

B. Related Works

Existing efforts aimed at accelerating GAT inference typically commence with algorithmic optimizations before proposing corresponding hardware architectures. Hou et al. introduced a node tailoring algorithm based on sorting attention coefficients, suggesting that nodes with higher degrees may only require information from a subset of their neighbors to yield accurate results. They proposed an corresponding accelerator architecture NTGAT [11], including a pipeline insertion sorting scheme and a computational engine. By diminishing the number of aggregation computations, this architecture aims to reduce inference latency and power consumption. However, it lacks an appropriate graph partition method and slices switching strategy thus heavily relying on on-chip storage resources. He et al. employed the ternary weight networks (TWNs) [12], [13] in GAT, presenting FTW-GAT, which reduces the model's memory footprint, simplifies processing elements, and eliminates reliance on digital signal processors (DSPs). The architecture includes multi-level pipelines and corresponding computing units. Nonetheless, due to the complex computational process and data dependencies inherent in GAT inference, FTW-GAT still experiences numerous stalling cycles during pipeline execution.

Hence, our design aims to enhance computational parallelism and accelerate inference while mitigating a substantial surge in storage resource consumption.

III. GAT WITH OPTIMIZED CALCULATION ORDER

A. Algorithm Decomposition

The calculation of attention weights represented by equation **(2)** is disassembled by extracting the dense computation as given in equation **(4)**.

$$\mathbf{a}^T[h'_i||h'_j] = \mathbf{a}_1 h'_i + \mathbf{a}_2 h'_j = p_i + q_j \tag{4}$$

Here, $\mathbf{a}_1$ and $\mathbf{a}_2$ denote the left and right attention kernel vectors respectively, where $\mathbf{a}_1, \mathbf{a}_2 \in \mathbb{R}^{1\times F'}$ and $\mathbf{a}^T = [\mathbf{a}_1||\mathbf{a}_2]$. Consequently, the original calculation of attention weights in **(2)** can now be expressed by equations **(4)** to **(6)**. p_i and p_j represent the left and right attention coefficient vectors, respectively.

$$e_{ij} = \mathrm{LeakyReLU}(p_i + q_j) \tag{5}$$

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k\in N_i} \exp(e_{ik})} \tag{6}$$

B. Optimized Calculation Order

The computation process above reveals a significant data dependency between equations **(5)**, **(6)**, and the final Aggregation **(3)**. In equation **(6)**, calculating attention weights requires accumulating attention coefficients e_{ij} for all node pairs, determined by equation **(5)**. This necessitates traversing all neighbors of the source node i before the denominator's value is known, delaying the aggregation in equation **(3)**. This leads to two problems:

1) A "boundary" exists between the calculation of attention weights and the aggregation step of a source node. The pipeline stalls when switching between these steps, as it must await the completion of the current step before proceeding to the next step of the source node.
2) Given that division must be computed for each node pair, the corresponding hardware design will necessitate a considerable number of DSPs to execute numerous division operations.

To solve the above problems, the above equation **(6)** is substituted into equation **(3)**, and then the fraction $1/(\sum_{k\in N_i}\exp(e_{ik}))$ is extracted through the distributive law of multiplication, so that the division operation in equation **(6)** is moved back and the multiplication and addition operation in equation **(3)** is performed in advance. Thus, the entire GAT algorithm with an optimized calculation order is as follows:

$$h'_i = \mathbf{W} h_i \tag{7}$$

$$p_i = \mathbf{a}_1 h'_i,\ q_j = \mathbf{a}_2 h'_j \tag{8}$$

$$e_{ij} = \mathrm{LeakyReLU}(p_i + q_j) \tag{9}$$

$$e'_{ij} = \exp(e_{ij}) \tag{10}$$

$$a'_{ij} = e'_{ij} \cdot h'_j \tag{11}$$

$$z_i = \mathrm{ELU}(\frac{\sum_{j\in N_i} a'_{ij}}{\sum_{k\in N_i} e'_{ik}}) \tag{12}$$

By placing the summation at the final step in the equations above, the earlier mentioned "boundary" is removed. This adjustment enables computation without waiting for all neighbors to be traversed. The coefficient $e'ij$ and the product $e'ij \cdot h'_j$ can be calculated in a pipelined and parallel manner, with sequential accumulation. The final outcome involves division and a non-linear activation function (ELU) in the last step. Furthermore, this approach eliminates the need for division operations for each node pair, requiring division only once per node in equation **(12)**.

IV. ARCHITECTURE DESIGN

A. Design Overview

An overview of OCO-GAT is depicted in **Fig. 1**. Due to the typically large size of graph data, the External Storage is required to store both the data of a complete graph and the computing results. The Main Controller initiates read requests and write requests via DMA to read data from or write data to the External Storage. Additionally, the Main Controller controls the switching of computing tasks.

The Combination Module executes the Combination stage and a portion of the Attention stage, which is further explained in **section IV-B**. The Aggregation Part manages the remaining Attention stage and the Aggregation stage to derive final

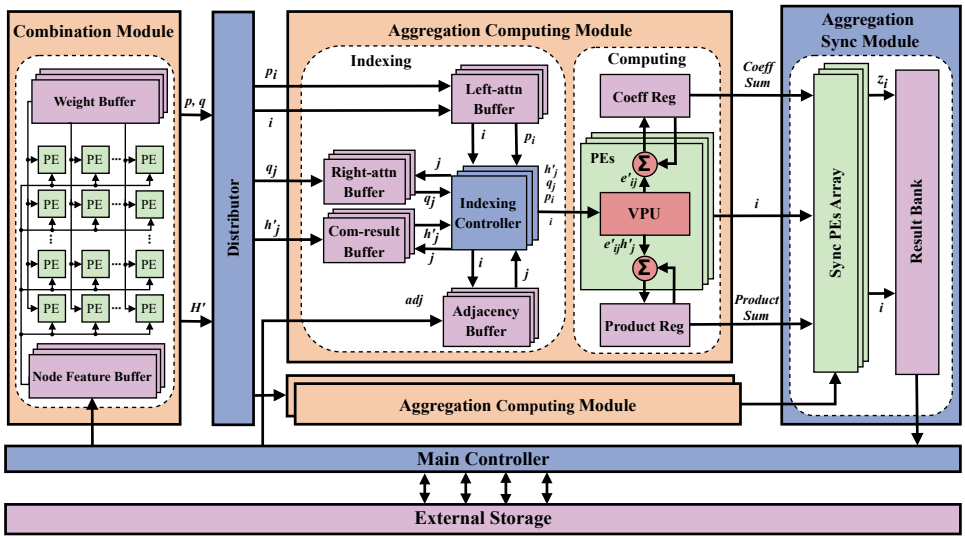

Fig. 1. Overall architecture of OCO-GAT.

embeddings, as detailed in **section IV-C**. The Aggregation Part is subdivided into several Aggregation Computing Modules and an Aggregation Sync Module. As described in **section IV-F** below, different sub-slices are fed into different Aggregation Computing Modules and processed in parallel. Moreover, different PEs compute different groups of tasks in parallel.

B. Combination Module

In the Combination Module illustrated in **Fig. 1**, the PEs array performs two types of linear transformations by accessing features, weights, and attention kernels stored in on-chip buffers. Firstly, the feature h_i is multiplied by the weights matrix $\mathbf{W}$ to obtain the combination result h'_i, as described in **(8)**. Additionally, h'_i undergoes vector inner products with the left and right attention kernel vectors to derive the left and right attention coefficient vectors p'_i and q'_i, respectively, as outlined in **(9)**. The outputs of the Combination Module are then transmitted to the corresponding buffers of the Aggregation Computing Modules via the Distributor.

C. Aggregation Part

Illustrated in **Fig. 1**, the Aggregation Part comprises several parallel Aggregation Computing Modules and an Aggregation Sync Module. The Aggregation Computing Modules index the on-chip memories and handle the primary computing tasks, while the Aggregation Sync Module collects intermediate results from parallel Aggregation Computing Modules, computes the final results, and stores them in the Result Bank.

1) Aggregation Computing Module: In **Fig. 1**, an Aggregation Computing Module is divided into the Indexing and Computing parts. The Indexing part includes Indexing Controllers and on-chip buffers. Controllers fetch data from buffers based on input source node IDs and transfer it to the Computing part. Within Computing, Vector Process Units (VPUs) perform primary computations. Processing Elements (PEs) accumulate coefficients and products from VPUs, storing partial sums in registers (Coeff Reg and Product Reg). Upon completing computations for all neighbors in the Adjacency Buffer, sums from registers are output to the Aggregation Sync Module.

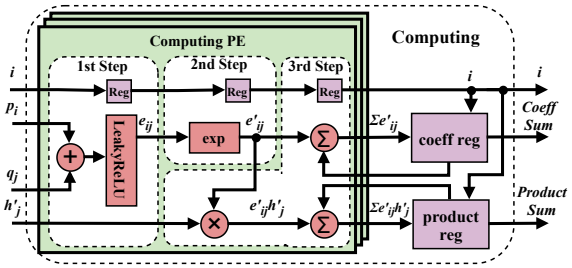

Fig. 2. The Computing Part of the Aggregation Computing Module.

Fig. 2 shows a detailed block diagram of the Computing part of the Aggregation Computing Module. The PEs in the computation part are divided into three steps:

1) **1st Step** receives the input left and right attention coefficients and performs addition followed by a nonlinear activation (LeakyReLU) as described in **(9)**.
2) **2nd Step** executes the exponential operation (exp) according to **(10)**.
3) **3rd Step** performs **(11)**, which involves multiplying the input h'_j by the e'_{ij}. Subsequently, it accumulates a part of the product $e'_{ij} \cdot h'_j$ and the e'_{ij} at the corresponding position of the source node i in the registers.

Registers are set in the hardware circuit of each step to store the IDs of the source nodes being processed, ensuring that the partial sums in **3rd Step** are accumulated accurately and stored in the correct designated positions within the registers.

2) Aggregation Sync Module: In **Fig.1**, the Aggregation Sync Module comprises a PE array and a Result Bank. Each PE within the array initially receives and accumulates partial sums from the Aggregation Computing Modules, then proceeds to compute the embedding of nodes. The Result Bank serves as a temporary storage for the node aggregation

results, i.e., the embeddings, which are then read by the Main Controller and transferred to the External Storage.

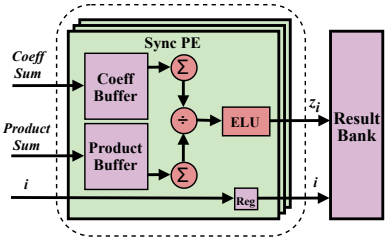

Fig. 3. The Sync PE of the Aggregation Sync Module.

Fig. 3 presents a detailed block diagram of the Sync PE in the Aggregation Sync Module. This Sync PE includes a Coeff Buffer and a Product Buffer, utilized for caching partial sums of coefficients and partial sums of products from the Aggregation Computing Modules. Consequently, upon completing the computation of one source node, the preceding modules can commence computing the next source node without delay.

By summing up the partial sum of coefficients of the source node i from each Aggregation Computing Module , we derive the denominator $\sum_{k\in N_i} e'ik$ of **(12)**. Similarly, by summing up the partial sum of products, we obtain the numerator $\sum_{j\in N_i} a'_{ij}$ of **(12)**. Dividing the sum of coefficients by the sum of products, and subsequently applying ELU to the division result, yields the embedding z_i of the source node i.

D. Pipeline Design

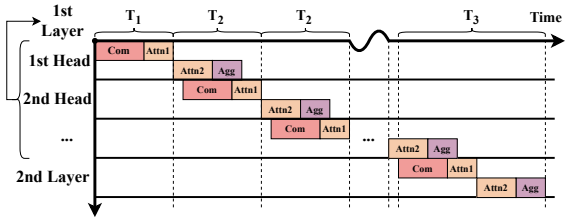

Fig. 4. Inter-head and inter-layer pipelines.

1) Inter-head and Inter-layer Pipeline: In **Fig. 4**, our approach utilizes both inter-head (attention heads) and inter-layer pipelines. Initially, the Combination Module conducts feature transformation in the Combination (**Com**) and Attention (**Attn1**) stages of the 1st attention head in the 1st layer. Subsequently, the Aggregation Part manages the remaining stages (**Attn2** and **Agg**). Concurrently, **Com** and **Attn1** of the 2nd attention head commence. This iterative process continues until all heads in the 1st layer are computed. During the final **Attn2** and **Agg** stages of the last attention head in the 1st layer, **Com** and **Attn1** of the 2nd layer are executed simultaneously. OCO-GAT employs ping-pong buffers to prevent data conflicts and ensure smooth pipeline operation.

2) Pipeline in the Attention and the Aggregation stage: **Fig. 5** illustrates the adjacency matrix of an example graph. Numbers correspond to the IDs of nodes. Each square with a letter represents a node pair, and squares of the same color indicate that they belong to the same source node.

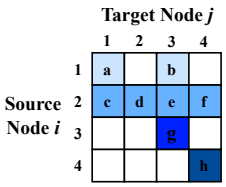

Fig. 5. The adjacency matrix of an example graph.

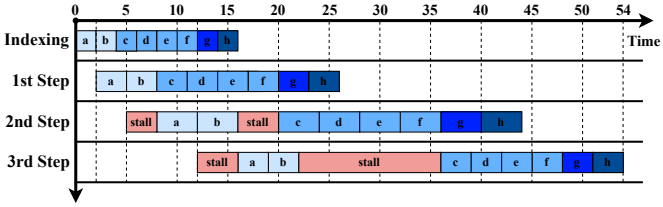

Fig. 6. Original GAT's node-grained pipeline.

In the original GAT, equation **(5)** is denoted as the **1st Step**, **(6)** is termed the **2nd Step**, and **(3)** is labeled as the **3rd Step**.

In this example, **Fig. 6** depicts the node-grained pipeline of the original GAT in the Attention and Aggregation stages. Due to "boundary" between different node pairs within these three steps, a hardware module corresponding to one step can only sequentially compute tasks for each node pair. Without clarity on task results belonging to specific node pairs, simultaneously processing tasks for different node pairs will cause errors.

To resolve this, a refined pipeline design subdivides steps into sub-steps. Registers between these sub-steps enable hardware modules to concurrently process tasks for multiple node pairs. Additional registers store source node IDs within each sub-step, detailed in **section IV-C1**, allowing error-free computation for different source nodes and increasing parallelism. Thus, the node-pair-based pipeline of the original GAT in Attention and Aggregation stages is shown in **Fig. 7**.

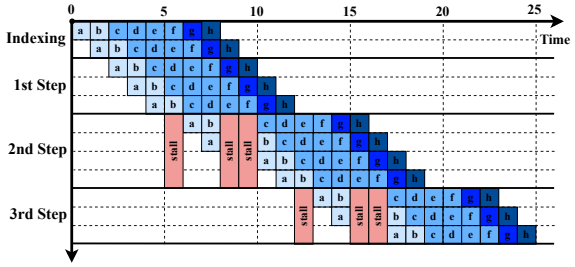

Fig. 7. Original GAT's node-pair-grained pipeline.

By employing the above optimization, the "boundary" between node pairs is effectively eliminated. However, there are still stalls due to the "boundary" between **2nd Step** and **3rd Step** as discussed in **section III-B**. For instance, the computation of source node **2** in the **3rd Step** must wait until the computation of **2** in the **2nd Step** is completed.

3) Node-pair-grained pipeline of OCO-GAT: To eliminate the "boundary" between steps, as illustrated in **Fig. 8**, we designed a node-pair-grained pipeline for the Aggregation Part of OCO-GAT. As described in **sections IV-C1** and **IV-C2**, in

the GAT with optimized calculation order, equation **(9)** and **(10)** is designated as the **1st Step** and **2nd Step** respectively, while the **(11)** is assigned as the **3rd Step**. Equation **(12)** is handled by the **Sync**. The pipeline of the Aggregation Part comprises **Indexing**, **1st Step**, **2nd Step**, and **3rd Step** within the Aggregation Computing Module, alongside **Sync** within the Aggregation Sync Module.

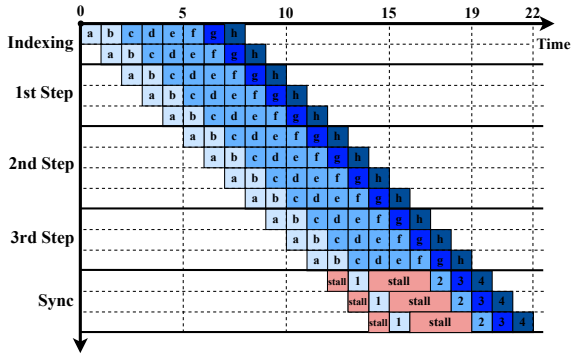

Fig. 8. Our work's node-pair-grained pipeline for the Aggregation Part.

Thanks to the optimized calculation order, steps no longer wait for previous steps to finish traversing all source node neighbors. This eliminates pipeline stalls between hardware modules of these steps. The Aggregation Sync Module includes intermediate buffers for coefficient and product sums, facilitating accurate node embedding computation. Stalls may occur in **Sync** when a source node has multiple neighbors, yet these are concealed during execution due to parallel operation with preceding steps and only introduce delays of processing **Sync** of the last node such as Node **4** in **Fig. 5**.

E. Distributed Fine-grained Storage Scheme

We use a distributed fine-grained storage scheme to eliminate storage conflicts, ensure computational parallelism, and mitigate on-chip storage resource growth. For the Aggregation stage, four types of data—left attention coefficients, right attention coefficients, combination results, and adjacency information—are required. Allocating four coarse-grained on-chip memories per slice necessitates shared buffers among all Aggregation Computing Modules will result in high bandwidth demands and potential access conflicts. So we adopt a distributed fine-grained storage scheme (**Fig. 1**), assigning on-chip buffers for each data type within every Aggregation Computing Module based on task-specific partitions of graph data. This method maintains parallelism while effectively managing on-chip storage resource consumption, without exponential growth as the number of modules increases.

F. Graph Partition

As noted earlier, the Main Controller controls task switching, and Aggregation Computing Modules execute tasks in parallel. Our distributed fine-grained storage scheme partitions data based on these modules' distinct tasks. How are tasks divided? We use the graph partition method illustrated in **Fig. 9**, marking the computation sequence with red arrows.

Fig. 9(a) shows an adjacency matrix of a complete graph, segmented into green rectangles based on on-chip storage capacity and adjacency size. The Distributor and Main Controller partition data into sub-slices by target node IDs for Aggregation Computing Modules. Each sub-slice is stored in distributed on-chip storage. **Fig. 9(b)** depicts N sub-slices when using N modules, each dedicated to an Aggregation Computing Module task. Tasks within sub-slices are further subdivided into source node groups based on adjacency matrix columns. PEs within modules compute these groups in parallel. **Fig. 9(c)** illustrates a group within a sub-slice: white squares denote no connections, dark green squares indicate connections, and solid arrows show sequential aggregation computation for source nodes and their neighbors.

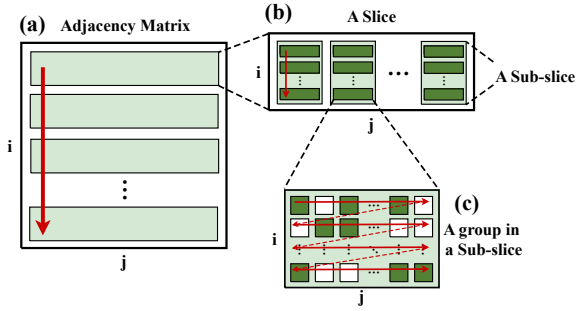

Fig. 9. Graph Partition.

V. EXPERIMENTS

A. Experimental Setup

1) Platforms and Methods: We utilize Verilog HDL to design OCO-GAT and employ Xilinx Vivado 2021.1 for synthesis and implementation on the Xilinx Alveo U250 hardware platform. External storage includes DDR4 and the Xilinx Memory Interface Generator. We evaluate the performance and power efficiency of OCO-GAT against PyTorch Geometric (PyG) [14] running on CPU (AMD Ryzen Threadripper 3960X) and GPU (NVIDIA GeForce RTX 3090, PyTorch version 1.9.0). Additionally, comparisons are made with FPGA-based accelerators FP-GNN [15], FTW-GAT, and NTGAT. Python is used for data preprocessing and generating hardware architecture parameters based on dataset statistics.

TABLE I
STATISTICS OF DATASETS

Dataset	Nodes	Edges	Feature	Weight
Cora	2708	10556	1433	16
Citeseer	3327	9104	3703	16
Pubmed	19717	88648	500	16
PPI	44906	1226368	50	128

2) Datasets: As indicated in **Table I**, four datasets are used in our experiments: Cora, Citeseer, Pubmed, and PPI. Across all datasets, we apply a two-layer GAT with eight attention heads. The hidden layer dimension (weight dimension) is set to 16 for Cora, Citeseer, and Pubmed, while for PPI, it is set to 128.

TABLE II
COMPARISON WITH BASELINES

	CPU	GPU	NTGAT [11]	FP-GNN [15]	FTW-GAT [16]	OCO-GAT
Platfrom	TR 3960X	RTX 3090	Alveo U200	VCU128	VCU128	Alveo U250
Frequency	2.9GHz	1.4GHz	300MHz	225MHz	225MHz	250MHz
Bandwidth	-	936GB/s	77GB/s	460GB/s	460GB/s	77GB/s
Size	24 Cores, 48 threads	10496 CUDA Cores	4096 PEs, 64 node engines	2048 PEs, 256 ALUs	7040 PEs, 64 ALUs, 1024 MACs	5120 CPEs, 64 ACPEs, 16 ASPEs
Resources Utilization						
DSP	-	-	6208 (91%)	8704 (96%)	1216 (13%)	6272 (51%)
On-chip RAM	-	-	32.5MB (93%)	7.0MB (89%) (1792 BRAMs)	5.87MB (75%) (1502.5 BRAMs)	6.54MB (61%) (1674 BRAMs)
LUT	-	-	-	1068077 (82%)	436657 (33%)	1027253 (59%)
FF	-	-	-	727254 (28%)	470222 (18%)	621250 (18%)
Inference Latency						
Cora	14.72ms (1×)	1.287ms (11.4×)	0.559ms (26.3×)	46.3μs (317.9×)	44.9μs (327.8×)	33.1μs (444.7×)
Citeseer	17.39ms (1×)	1.412ms (12.3×)	1.465ms (11.9×)	71.4μs (243.6×)	50.8μs (342.3×)	47.8μs (363.8×)
Pubmed	199.4ms (1×)	1.706ms (116.9×)	2.097ms (95.1×)	616μs (323.7×)	339μs (588.2×)	286μs (697.2×)
PPI	1714ms (1×)	46.39ms (36.9×)	-	-	-	17.37ms (98.7×)
Energy Efficiency (graph/kJ)						
Cora	-	8.27×10^3 (1×)	3.68×10^4 (4.5×)	1.46×10^6 (176.5×)	3.65×10^6 (441.4×)	1.65×10^6 (199.5×)
Citeseer	-	7.23×10^3 (1×)	1.41×10^4 (2.0×)	9.46×10^5 (130.8×)	3.14×10^6 (434.3×)	1.14×10^6 (157.7×)
Pubmed	-	5.43×10^3 (1×)	9.81×10^3 (1.8×)	1.10×10^5 (20.3×)	3.87×10^5 (71.3×)	1.91×10^5 (35.2×)
PPI	-	6.76×10^1 (1×)	-	-	-	3.15×10^3 (46.6×)

B. Algorithm Evaluation

We evaluate how optimizing calculation order impacts division operations in GAT across diverse datasets. The reduction in division operations for the optimized GAT compared to the original is shown in **Fig. 10**. Results indicate that the optimized version required only 3.47% to 26.76% of the division operations across datasets. This reduction is influenced by the density of adjacency matrix and the size of the graph and it is particularly effective for dense or larger graphs.

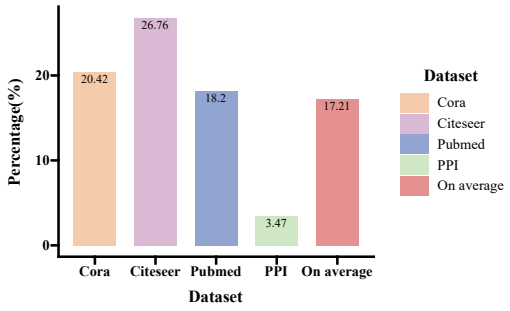

Fig. 10. The percentages of the number of division operations.

C. Hardware Performance

1) Resource Utilization: **Table II** displays the hardware resource usage of OCO-GAT. Our configuration includes 5120 PEs (CPEs) in the Combination Module, 64 Computing PEs (ACPEs) in the Aggregation Computing Modules, and 16 Sync PEs (ASPEs) in the Aggregation Sync Module. The Main Controller integrates four DDR_MIGs with a bandwidth up to 77GB/s. The results show that OCO-GAT operates efficiently without resource bottlenecks at this scale, suggesting potential for scaling up to speed up inference or handle larger graphs.

2) Energy Efficiency: The dynamic power consumption of OCO-GAT measures 18.3W according to Vivado power reports. GPU power measurements, obtained via the Nvidia System Management Interface, include active and idle states, with dynamic GPU power calculated by difference. Notably, NTGAT only specifies its total on-chip power at 48.6W, omitting details on dynamic power. As shown in **Table II**, we calculate energy efficiency (graphs/kJ) based on dynamic power and inference latency. This metric indicates the number of graphs processed per kilojoule of energy.

3) Inference Latency: Our work significantly boosted GAT inference speeds, achieving a remarkable acceleration by 301.1× and 19.3× on average compared to CPU and GPU. In comparison with NTGAT, OCO-GAT utilized fewer storage but achieved 18.3× improvements on average. Compared to FP-GNN, OCO-GAT runs 1.68× faster with fewer resources. OCO-GAT delivers superior performance than FTW-GAT, reducing inference latency by an average of 1.20×.

VI. CONCLUSION

In this paper, we introduce OCO-GAT, a highly efficient GAT accelerator based on FPGA. We propose an optimized calculation order for GAT, serving as a blueprint for the hardware design of the overall architecture, processing elements, and the multi-level pipeline. Through an exploration of node pairs parallelism, we mitigate pipeline stalls and reduce costly division operations, thereby enhancing overall performance. Moreover, we utilize a graph partition method and employ a distributed fine-grained storage scheme to prevent excessive storage resources utilization while ensuring parallel computation. Experimental results demonstrate that OCO-GAT outperforms multiple baselines across various datasets and achieves high energy efficiency.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China under Grant 62276278 and GuangDong Basic and Applied Basic Research Foundation under Grant 2022A1515110006 and 2024A1515011259.

REFERENCES

[1] X. Song, T. Zhi, Z. Fan, Z. Zhang, X. Zeng, W. Li, X. Hu, Z. Du, Q. Guo, and Y. Chen, "Cambricon-g: A polyvalent energy-efficient accelerator for dynamic graph neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, pp. 116–128, 2021.

[2] S. Zhang, H. Tong, J. Xu, and R. Maciejewski, "Graph convolutional networks: a comprehensive review," *Computational Social Networks*, vol. 6, no. 1, 2019.

[3] D. F. Nettleton, "Data mining of social networks represented as graphs," *Computer Science Review*, vol. 7, no. 1, p. 1 – 34, 2013.

[4] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: A survey," *ACM Computing Surveys*, vol. 55, no. 5, 2022.

[5] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The Web Conference 2019 - Proceedings of the World Wide Web Conference, WWW 2019*. Association for Computing Machinery, Inc, 2019, p. 417 – 426.

[6] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey," *Expert Systems with Applications*, vol. 207, 2022.

[7] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *34th International Conference on Machine Learning, ICML 2017*, vol. 3. International Machine Learning Society (IMLS), 2017, p. 2053 – 2070.

[8] C. McGill, M. Forsuelo, Y. Guan, and W. H. Green, "Predicting infrared spectra with message passing neural networks," *Journal of Chemical Information and Modeling*, vol. 61, no. 6, p. 2594 – 2609, 2021.

[9] P. Veličković, A. Casanova, P. Liò, G. Cucurull, A. Romero, and Y. Bengio, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 2018.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 2017-December. Neural information processing systems foundation, 2017, p. 5999 – 6009.

[11] W. Hou, K. Zhong, S. Zeng, G. Dai, H. Yang, and Y. Wang, "Ntgat: A graph attention network accelerator with runtime node tailoring," in *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*. Institute of Electrical and Electronics Engineers Inc., 2023, p. 645 – 650.

[12] C. Zhu, H. Mao, S. Han, and W. J. Dally, "Trained ternary quantization," in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 2017.

[13] B. Liu, F. Li, X. Wang, B. Zhang, and J. Yan, "Ternary weight networks," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, vol. 2023-June. Institute of Electrical and Electronics Engineers Inc., 2023.

[14] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," in *7th International Conference on Learning Representations, ICLR 2019 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, 2019.

[15] T. Tian, L. Zhao, X. Wang, Q. Wu, W. Yuan, and X. Jin, "Fp-gnn: Adaptive fpga accelerator for graph neural networks," *Future Generation Computer Systems*, vol. 136, p. 294 – 310, 2022.

[16] Z. He, T. Tian, Q. Wu, and X. Jin, "Ftw-gat: An fpga-based accelerator for graph attention networks with ternary weights," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 70, no. 11, p. 4211 – 4215, 2023.

[17] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, "A systematic survey of general sparse matrix-matrix multiplication," *ACM Computing Surveys*, vol. 55, no. 12, 2023.