H-GAT: A Hardware-Efficient Accelerator For Graph Attention Networks

Shizhen Huang, Enhao Tang*, and Shun Li

College of Physics and Information Engineering, Fuzhou University, Fuzhou 350116, China

Corresponding author. E-mail: 211127004@fzu.edu.cn

Received: Dec. 05, 2022; Accepted: Jul. 18, 2023

Recently, Graph Attention Networks (GATs) have shown good performance for representation learning on graphs. Furthermore, GAT leverage the masked self-attention mechanism to get a more advanced feature representation than the graph convolution networks (GCNs). However, GAT incurs large amounts of irregularity in computation and memory access, which prevents the efficient use of traditional neural network accelerators. Moreover, existing dedicated GAT accelerators demand high memory volumes and are difficult to implement onto resource-limited edge devices. Due to this, this paper proposes an FPGA-based accelerator, called H-GAT, which achieves excellent performance on acceleration and energy efficiency in GAT inference. H-GAT decomposes GAT operation into matrix multiplication and activation function unit. We first design an effective and fully-pipelined PE for sparse matrix multiplication (SpMM) and dense matrix-vector multiplication (DMVM). Moreover, we optimize the softmax data flow so that the computational efficiency of softmax can be improved dramatically. We evaluate our design on Xilinx Kintex-7 FPGA with three popular datasets. Compared to existing CPU, GPU, and state-of-the-art FPGA-based GAT accelerator, H-GAT can achieve speedup by up to 585×, 2.7×, and 11× and increases power efficiency by up to 2095×, 173×, and 65×, respectively.

Keywords: Graph neural network; FPGA; sparse-matrix-vector;

© The Author('s). This is an open-access article distributed under the terms of the Creative Commons Attribution License (CC BY 4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are cited.

http://dx.doi.org/10.6180/jase.202403_27(3).0010

1. Introduction

Over recent years, Graph Neural Networks (GNNs) have been applied in numerous fields due to their superior performance in learning graph data, including networking, biology [1], recommendation, [2], etc. Graph convolutional networks (GCNs) [3], which borrow ideas from convolutional neural networks (CNNs), performs well for many real-time related tasks, such as node classification [4], link prediction [5], graph classification [6], etc. Like most deep learning algorithms, GCNs also contains many layers, and each layer of GCN is divided into two phases, combination and aggregation.

Moreover, developed under the self-attention mechanism, Graph Attention Network (GAT) [7] introduces the attention mechanism based on GCNs. GATs have achieved an outstanding effect in the same related tasks compared to GCNs [7]. Unlike GCNs, GATs perform self-attention calculations before aggregation. The purpose of the self-attention mechanism in GAT is to compute the corresponding edge values between the central node and its neighboring nodes, which is unique to GAT. So existing GCN accelerators are incompatible with the GAT model, and another reason is that GCN has only matrix multiplication operation. However, there are many existing GCN accelerators, but hardware acceleration for GAT is relatively scarce, so the development of an accelerator for GAT is necessary. Moreover, as GAT grows in popularity and supports numerous real-world applications, it is natural for its inference workload to see heavy demand on edge devices soon [8]. It is not realistic to provide powerful hardware resources among these resource-limited devices, therefore a more lightweight approach is also required.

Based on our analysis of widely used datasets, the data distribution in the graph adjacency matrix and feature matrix is sparse and random, as shown in Table 1. It causes irregular data access in memory. Because of the presence of high sparsity of graph data, Traditional matrix multiplication architecture fail to maximize computational efficiency. Although existing GCN accelerators have addressed the irregularities caused by sparsity [9, 10], they cache large amounts of data with high-bandwidth memory (HBM) [11] to enhance accelerator Performance. Moreover, The Existing GAT accelerator ignores the important challenge of graph sparsity, although it reduces the use of DSP resources through quantization algorithms. For example, S-GAT [12] neglects to preprocess time and CPU communication time, and its PE unit does not exploit the sparsity of the graph for matrix multiplication. From the above two points, the key challenge of effective GAT implementation is the irregularity in computation and memory access.

Computation: GAT involves both sparse matrix multiplication and dense matrix multiplication, and also involves intensive exp operations in the activation function.

Memory access: The edge extraction process relies on the central node and its neighboring nodes, which incurs a large number of irregular memory accesses due to the sparsity of the graph. This memory access challenge is unique to the GAT.

Finally, we propose H-GAT, an FPGA-based accelerator for GAT inference. To summarize, our contributions are listed as follows:

- We optimize the softmax data flow to make activation function unit consume less computational complexity and be friendly to hardware.
- Based on modified CSR (MCSR) [13], we design an effective and lower power consumption for sparse matrix multiplication (SpMM) and dense matrix-vector multiplication (DMVM), which further increases the computation efficiency of H-GAT.
- We implement H-GAT on Xilinx Kintex-7 FPGA under three popular datasets. Compared with Intel I7-12700KF CPU, Nvidia RTX3090 GPU and FPGA-based GAT accelerator, H-GAT achieves up to 585×, 2.7×, and 11× speedup, while up to 2095×, 173 ×, and 65× better energy efficiency, respectively. Experimental results show that H-GAT achieves better performance and lower power consumption.

2. Background and related work

2.1. GAT background

Based on GCN, GAT are built by adding an attention mechanism to calculate the importance between nodes more precisely. Each layer in GATs has the same structure called graph attention layer. The graph attention layer can be divided into three steps: (a) linear transformation; (b) self-attention process; (c) feature aggregation. In the first step, the result of first step in expressed as h° and the corresponding expression is:

$$\mathrm{h}^\circ = \mathrm{hW} \tag{1}$$

In the second step, the self-attention mechanism calculates the attention coefficients e_{mn}. We use T to represent matrix transpose. Then e_{mn} can be calculated by:

$$e_{mn} = \text{leakyrelu}\left(a^T \text{concat}\left(h_m^\circ, h_n^\circ\right)\right) \tag{2}$$

The attention coefficients e_{mn} indicate the importance of node n to node m. The attention mechanism in GATs is masked attention: $\mathrm{n} \in \mathrm{N_m}$ and $\mathrm{N_m}$ are the neighborhoods of node m in the graph. Then the softmax function is used to normalize e_{mn}:

$$\alpha_{mn} = \frac{\exp\left(e_{mn}\right)}{\sum_{k \in N_m} \exp\left(e_{mk}\right)} \tag{3}$$

The final step is feature aggregation operation, and the result is the final output of each node. The corresponding expression is:

$$h' = elu\left(\sum_{n \in N_m} \alpha_{mn} h_n^\circ\right) \tag{4}$$

GATs achieve outstanding effects in various real-time tasks by applying the attention mechanism.

2.2. Motivation

The GAT is based on the GCN and incorporates the attention mechanism in Transformer, which represents each neighbor node's importance to the target node during the aggregation process by calculating the attention coefficients. The computed attention coefficients are more interpretable than the node-based approach, which makes GAT perform better than GCN in inference applications, such as Machine translation [14]. However, GAT often encounters challenges in real-world applications, as shown in Fig. 1. Real-time inference scenarios are very demanding for inference time, so accelerating the inference process of GAT is meaningful work. In addition, when deploying GAT on some edge devices, power consumption can be an important consideration because the power consumption that

Table 1. Dimensions and densities of widely used datasets [8]

Datasets	Nodes	Edges	Input Features	Classes	Feature Density	Edge Density	Weight Density
Cora	2708	10556	1433	7	1.3%	0.14%	100%
CiteSeer	3327	9104	3703	6	0.8%	0.08%	100%
PubMed	19717	88648	500	3	10.4%	0.02%	100%

edge devices can afford is very limited. Our goal is to design an accelerator for GAT that can bring an order of magnitude improvement to the inference speed of GAT and balance power consumption to accommodate edge computing.

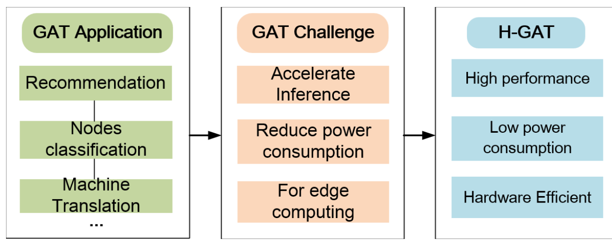


Fig. 1. The challenges of GAT accelerator and the advantages of H-GAT

2.3. GNN accelerator related work

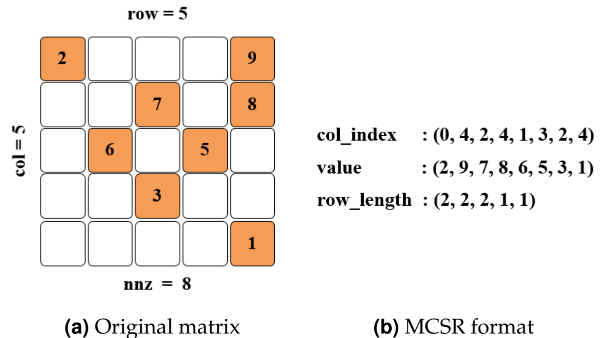
Recently, many domain-specific architectures have been proposed to partially address the challenges in GNN inference. Autotuning-Workload-Balancing GCN (AWB-GCN) [9] dynamically balanced the workload between the processing engine (PE) through three hardware-based task scheduling mechanisms. As an early FPGA-based GNN accelerator, the performance improvement of AWB-GCN is phenomenal. BoostGCN [15] designed a feature aggregation module and two feature update modules for different sparsity to optimize matrix computation. I-GCN [16] proposed a new algorithm for graph reconstruction to improve data locality and matrix operation efficiency. By merging nodes with shared neighbors, it prevents redundant operations in the aggregation phase. HyGCN [17] proposes a two-stage accelerator to deal with both the memory-intensive aggregation phase and the compute-intensive combination phase. However, the performance and hardware resource budget must be considered carefully when deploying GNN models on some resource-limited edge-computing devices [18].

3. Software preprocessing

In this section, we first describe in detail the data compression methods we use, followed by our load-balancing approach.

3.1. MCSR data compression format

As shown in Table 1, the graph adjacency matrix and the feature matrix of the first layer in GATs are often extremely sparse. Therefore, we compress these matrices to process only valuable information (non-zero elements) to save storage and reduce computation complexity. We use a hardware-friendly compression method, which is called MCSR format [13, 19]. It stores matrix information through three arrays, named col-index, value and row-length, as shown in Fig. 2b. The col-index stores the column indices of the non-zero elements. The value array stores only non-zero elements, and row-length stores the number of non-zero elements in each row. In addition, row-length along with a simple counter can quickly determine the current row being computed. It is worth mentioning that MCSR can be well applied to hardware parallel pipeline computing.



(a) Original matrix **(b)** MCSR format

Fig. 2. The sparse matrix storage format

3.2. Workload scheduling for balance

At first, we perform scheduling by row-based parallelization. Specifically, all non-zero elements in the same row of the sparse matrix are all distributed to the same PE, while elements of different rows are distributed to different PE in a round-robin manner, according to Fig. 3b. However, influenced by the high sparsity of the graph data, each row of the adjacency matrix will have different number of non-zero elements. So if we dispatch data to PE in a round-robin fashion, this leads to inefficiencies and data stagnation, shown as in the assignment step in Fig. 3b [8]. To increase PE efficiency, we design the PEs to work independently, each PE starts to compute a new row immedi-

ately when it finishes the previous one. This way, multiple rows are effectively concatenated before being distributed to one PE, this way we eliminate the time of data stagnation, which is shown as the concatenation step in Fig. 3c. Since it is unlikely for the density of a row to correlate with its row number, by the Law of Large Numbers we expect the sum of densities of rows assigned to each PE to be similar. Finally, we add zero elements at the end of each concatenated row to ensure all PEs compute the same amount of non-zero elements.

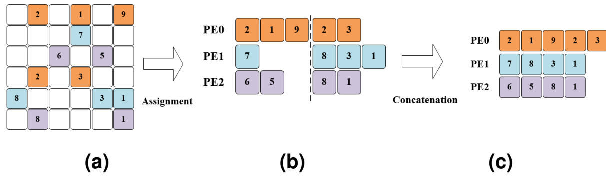


Fig. 3. (a) A sparse matrix (b) Row-based parallelization (c) Our scheduling strategy

4. Architecture

In this section, we focus on the hardware architecture of the H-GAT. First, we introduce the overall structure of H-GAT, and then we explain the hardware architecture of each main module.

4.1. Overview

Based on the advantages of FPGA parallelism, we redesigned the GAT inference process to make better use of the FPGA. The overall workflow of H-GAT is shown in Fig. 4. We divide the shared weight α in the self-attention mechanism into α_1 and α_2 which α_1 and α_2 are used in the central and neighboring nodes, respectively. Then the calculation formula of the attention coefficient becomes:

$$e_m = \alpha_1^T h_m^\circ, \quad e_n = \alpha_2^T h_n^\circ \tag{5}$$

$$e_{mn} = \text{leakyrelu}\left(e_m + e_n\right) \tag{6}$$

$$\alpha_{mn} = \frac{\exp\left(e_{mn}\right)}{\sum_{k \in N_m} \exp\left(e_{mk}\right)} \tag{7}$$

Among them, h_m° and h_n° are the features of the central node and the corresponding adjacent nodes after linear transformation respectively. Next, we change the aggregation of activation functions to relu which can simplify the architecture and reduce multiplications while maintaining accuracy:

$$h' = \text{relu}\left(\sum_{n \in N_m} \alpha_{mn} h_n^\circ\right) \tag{8}$$

The data flow of our overall architecture follows the optimized inference process. First, the Data-loader module is responsible for Caching enough h, w and α. The Data-loader module feeds the prepared data into the PE for calculation. But before that the data needs to be adjusted according to our scheduling strategy. Fig. 4 shows that h (feature) is represented by row-length, col-index and value by MCSR. Fig. 3c shows our scheduling strategy to ensure responsible balancing, and Fig. 4 shows the corresponding hardware. First, the col-index and value corresponding to each row are pre-prepared by row-length. If any (Sparse-PE) SP-PE completes the computation, the SP-PE returns the SP-PE completion signal and its corresponding SP-PE address to notify mux to send the data. The Data-loader will pack the Distributed h (row-length, col-index and value) and send it to the corresponding SP-PE. Next, the SP-PE and (Dense-PE) D-PE are used to calculate SpMM and DMVM. The PEs performs the calculation in this module, in which the resulting e_m and e_n are accumulated and entered into the Activation Function (AF) module. In addition, we cache for reuse in the aggregator module. In the AF module, the data goes through the leakyrelu and softmax modules and then into the aggregation module. Finally, the attention coefficient α_{mn} is aggregated with hw to complete a layer of operations.

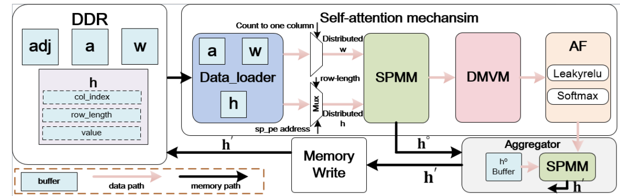


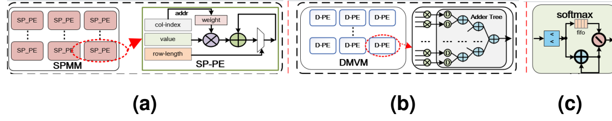
Fig. 4. The overall architecture and workflow of H-GAT

4.2. Self-attention mechanism module

We calculate h°, e_m, and e_n in this module which consists of three modules: Data-loader, PEs, and AF module.

Data-loader module performs reading features of nodes h stored in buffer h and the weights α and W stored in buffer α_{mn} and W. First, the adjacency matrix is compressed in the adj buffer according to the MCSR format. And then the central node and its neighboring node ID can be retrieved by the col-index and row-length. Next, the node id is used as the address signal to get the node feature. After that, the corresponding W and a are obtained by the col-index of the node feature. The PEs are responsible for calculating this module's sparse and dense matrix

multiplication. SP-PE gets data from Data-loader module through memory and then performs calculations. Further data inputs to D-PE to get e_m and e_n. When completing the calculations in D-PE, the result is input to AF module to perform leakyrelu and softmax calculations. The detailed description of the PEs module is in Fig. 5a and Fig. 5b.



(a) (b) (c)

Fig. 5. The hardware architecture of each computing unit (a) Detailed architecture of SPMM and an SP-PE (b) Detailed architecture of DMVM and a D-PE (c) The architecture of softmax accelerator.

4.3. PE architecture

An SP-PE consists of a sparse matrix-vector multiplication (SPMV). The overall SPMV architecture is shown in Fig. 5a. In terms of structure, the main modules of an SP-PE include a multiplier, adder, mux, and BRAM. The multiplier is the vector value indexed by the col-index. After multiplication is completed, the mux determines whether to continue the accumulation or output based on the value of row-length.

The overall D-PE architecture is shown in Fig. 5b. D-PE is responsible for calculating DMVM, which includes lots of adders and multipliers. After receiving the data from SP-PE, the multiplier starts the multiplication operation. Afterward, the multiplication result is sent to the additional tree for summation.

4.4. Softmax architecture

After the PEs calculation to get the e_{mn}, the data is input to the AF module. The AF module is responsible for the activation function part of the GAT, which consists of leakyrelu and softmax. The operation of softmax is:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \Longrightarrow \frac{2^{z_i}}{\sum_{j=1}^{K} 2^{z_j}} \tag{9}$$

Note that we replace the original power from e to 2, and this is more friendly to hardware [20]. The overall softmax architecture is shown in Fig. 5c. Since our PEs section is a full pipeline structure, our softmax accelerator also uses a full pipeline structure in order not to break the overall pipeline structure. Since our power is 2, we replace it with a shift, which saves a lot of resources and does not lose accuracy. The data flow is divided into two streams after the shift operation. The lower data path sums up all the data as the denominator of the softmax formula. The upper data stream is data cached via FIFO, aiming to wait for the lower data stream to complete its summation. Finally, the data from the upper and lower data paths are input to the divider for division.

4.5. Aggregator module

This module aggregates h° using the α_{mn} derived from the self-attention mechanism module. Since h° is already done in the Self-attention Mechanism Module, we reuse this data via h° ram. This aggregator is also calculated using the matrix multiplication units of PEs. Once the aggregator receives the α_{mn} from self-attention mechanism module, it aggregates and sends the result to buffer h in DDR.

5. Experiments

5.1. Experimental setup

The proposed H-GAT is coded in Verilog HDL and synthesized by Vivado 2020.2 on the Xilinx Kintex-7 K325T FPGA with 16-bit data width.

We use the Pytorch Geometric (PyG) tool to convert 16-bit floating-point to 16-bit integer without loss of accuracy. We comprehensively evaluate H-GAT using three popular datasets (Cora, CiteSeer, and PubMed). The size and sparsity of each datasets are shown in Table 1. We evaluate H-GAT on a two-layer GAT which use a hidden size of 8 and head number of 2. We then compare a final implementation against existing computing platforms. In order to evaluate the capabilities of H-GAT, we complete the deployment of H-GAT on the Xilinx Kintex-7 K325T FPGA. We then compare H-GAT with two typical baseline systems, including the Intel I7-12700KF CPU and the Nvidia RTX3090 GPU platform. Note that both platforms are the inference process of running the GAT model under PyG. Finally, we compare H-GAT with the state-of-the-art (SOTA) FPGA-based GAT accelerator S-GAT, which run on the Inspur F10A board. Although these platforms are not edge platforms, we still outperform them in terms of performance and power consumption in an unfair way.

5.2. Comparison with CPU and GPU

With device information in the experimental setup, we compare the latency in Fig. 6a, and in Table 2 can also clearly understand the data. H-GAT performs average 585× (min 238×, max 1248×) faster than CPU and 2.7× (min 1.1×, max 3.7×) faster than GPU under GAT model with different datasets. As shown in Fig. 6, although H-GAT is hardware designed for edge devices, our performance is better than both CPU and GPU performance. This is because our design is fully-pipeline for the computational process. And then we have full scheduling calculation for the sparse part of the computation. In contrast, H-GAT

performs well in GPU for Cora and CiteSeer datasets but slightly less in PubMed datasets on the GPU side. This is because it contains the biggest matrix in the dataset and concat operation requires taking a series of accumulation for huge-size matrixes. We compare the energy efficiency of our design on different platforms in Fig. 6b, which are evaluated based on Eq. (10) and Eq. (11) [12], and in Table 3 can also clearly understand the data. Compared with CPU and GPU, H-GAT shows the average 2095× and 173× better energy efficiency.

$$\text{Efficiency} = \frac{\text{performance}}{\text{power}} \tag{10}$$

$$\text{performace} = \frac{1}{\text{Time}} \tag{11}$$

Table 2. Latency comparison with CPU, GPU, and S-GAT

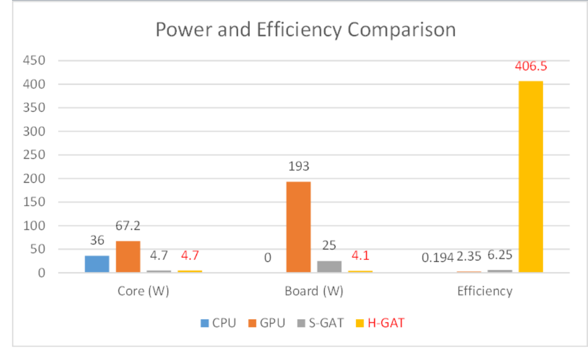
Datasets	Latency (ms)			
	H-GAT	CPU	GPU	S-GAT
Cora	0.6	142.8	2.2	6.4
CiteSeer	0.8	215.1	2.7	N/A
PubMed	5.7	7111.7	6.2	N/A

Table 3. Power and Efficiency comparison with CPU, GPU, and S-GAT

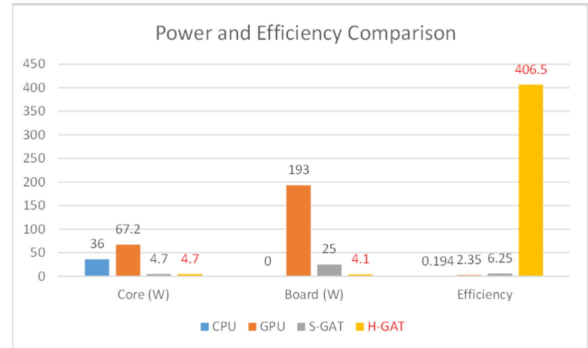
Models	Core (W)	Board (W)	Efficiency
CPU	36	N/A	0.194
GPU	67.2	193	2.35
S-GAT	4.7	25	6.25
H-GAT	4.7	4.1	406.5

5.3. Comparison with FPGA accelerators

We evaluate the overall resource consumption, latency, and energy efficiency of H-GAT against the state-of-the-art FPGA-based GAT accelerator S-GAT for the same datasets. Although S-GAT is deployed on the Inspur F10A board, we made a comparison. Since S-GAT only has performance and energy efficiency for the Cora dataset, we compare only the Cora dataset. As shown in Fig. 6a, H-GAT performs 11× faster than S-GAT. S-GAT does not exploit the sparsity of graph data and the high parallelism in the design



(a)



(b)

Fig. 6. (a) Latency comparison with CPU, GPU, and S-GAT (b) Power and Efficiency comparison with CPU, GPU, and S-GAT

of PE. At the same time, H-GAT shows 65× better energy efficiency compared with S-GAT in Fig. 6b. The resource utilization comparison is shown in Table 4. In terms of resources, we are equally better than S-GAT in the two-layer and two-head GAT.

6. Conclusions

In this paper, we have designed and implemented an accelerator for GAT model in resource-limited hardware platform such as edge devices. H-GAT consists of high-performance sparse matrix multiplication, dense matrix multiplication units and softmax modules, whose high-performance needs through sparse matrix compression,

Table 4. Power and Efficiency comparison with CPU, GPU, and S-GAT

H-GAT resource utilization on Xilinx Kintex-7 K325T FPG						
Resource	LUT	LUTRAM	FF	BRAM	DSP	Frequency
Used	38894	542	41960	118.5	244	
Available	203800	64000	407600	445	840	200 MHz
Utilization (%)	19.2 %	0.84 %	10.3 %	26.6 %	29.1 %	
S-GAT resource utilization on Inspur F10A board						
Used	250570	N/A	338490	683	148	216MHz

workload assignment and pipeline design. Experiments show that for GAT, H-GAT reduces latency by up to 585×, 2.7×, and 11× compared to CPU, GPU, and S-GAT and increases power efficiency by up to 2095×,173×, and 65×. For future work, A generic architecture for graph neural network models will be designed and implemented.

7. Acknowledgements

This work was supported by Fujian Natural Science Foundation under Grant 2023J01398.

References

[1] K. Atz, F. Grisoni, and G. Schneider, (2021) *"Geometric deep learning on molecular representations"* **Nature Machine Intelligence** *3*(12): 1023–1032. DOI: 10.1038/s42256-021-00418-8.

[2] X. He, K. Deng, X. Wang, Y. Li, Y. Zhang, and M. Wang. "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation". In: *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '20. Virtual Event, China: Association for Computing Machinery, 2020, 639–648. DOI: 10.1145/3397271.3401063.

[3] T. N. Kipf and M. Welling. *Semi-Supervised Classification with Graph Convolutional Networks*. Tech. rep. arXiv:1609.02907. arXiv:1609.02907 [cs, stat] type: article. arXiv, 2017. DOI: 10.48550/arXiv.1609.02907.

[4] S. Abu-El-Haija, A. Kapoor, B. Perozzi, and J. Lee. *N-GCN: Multi-scale Graph Convolution for Semi-supervised Node Classification*. Tech. rep. arXiv:1802.08888. arXiv:1802.08888 [cs, stat] type: article. arXiv, 2018. DOI: 10.48550/arXiv.1802.08888.

[5] M. Zhang and Y. Chen. "Link prediction based on graph neural networks". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, 5171–5181.

[6] M. Zhang, Z. Cui, M. Neumann, and Y. Chen, (2018) *"An End-to-End Deep Learning Architecture for Graph Classification"* **Proceedings of the AAAI Conference on Artificial Intelligence** *32*(1): DOI: 10.1609/aaai.v32i1.11782.

[7] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. "Graph Attention Networks". In: arXiv, 2018. DOI: 10.48550/arXiv.1710.10903.

[8] Z. Tao, C. Wu, Y. Liang, and L. He. *LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator*. Tech. rep. arXiv:2111.03184. arXiv, 2021. DOI: 10.48550/arXiv.2111.03184.

[9] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt. "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing". In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020, 922–936. DOI: 10.1109/MICRO50266.2020.00079.

[10] S. Liang, Y. Wang, C. Liu, L. He, H. LI, D. Xu, and X. Li, (2021) *"EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks"* **IEEE Transactions on Computers** *70*(9): 1511–1525. DOI: 10.1109/TC.2020.3014632.

[11] K. Kamalakkannan, G. R. Mudalige, I. Z. Reguly, and S. A. Fahmy. "High-Level FPGA Accelerator Design for Structured-Mesh-Based Explicit Numerical Solvers". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ISSN: 1530-2075. 2021, 1087–1096. DOI: 10.1109/IPDPS49936.2021.00117.

[12] W. Yan, W. Tong, and X. Zhi. "S-GAT: Accelerating Graph Attention Networks Inference on FPGA Platform with Shift Operation". In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. ISSN: 2690-5965. 2020, 661–666. DOI: 10.1109/ICPADS51040.2020.00093.

[13] M. Hosseinabady and J. L. Nunez-Yanez, (2020) *"A Streaming Dataflow Engine for Sparse Matrix-Vector Multiplication Using High-Level Synthesis"* **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems** *39*(6): 1272–1285. DOI: 10.1109/TCAD.2019.2912923.

[14] D. Bahdanau, K. Cho, and Y. Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. Tech. rep. arXiv:1409.0473. arXiv, 2016. DOI: 10.48550/arXiv.1409.0473.

[15] B. Zhang, R. Kannan, and V. Prasanna. "BoostGCN: A Framework for Optimizing GCN Inference on FPGA". In: *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. ISSN: 2576-2621. 2021, 29–39. DOI: 10.1109/FCCM51124.2021.00012.

[16] T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, M. Herbordt, Y. Lin, and A. Li. “I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement through Islandization”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, 1051–1063. DOI: 10.1145/3466752.3480113.

[17] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie. “HyGCN: A GCN Accelerator with Hybrid Architecture”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. ISSN: 2378-203X. 2020, 15–29. DOI: 10.1109/HPCA47549.2020.00012.

[18] Z. Zhou, B. Shi, Z. Zhang, Y. Guan, G. Sun, and G. Luo. “BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices”. In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. ISSN: 0738-100X. 2021, 1009–1014. DOI: 10.1109/DAC18074.2021.9586181.

[19] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang. “MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product”. en. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Athens, Greece: IEEE, 2020, 766–780. DOI: 10.1109/MICRO50266.2020.00068.

[20] Z. Xu, J. Yu, C. Yu, H. Shen, Y. Wang, and H. Yang. “CNN-based Feature-point Extraction for Real-time Visual SLAM on Embedded FPGA”. In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. ISSN: 2576-2621. 2020, 33–37. DOI: 10.1109/FCCM48280.2020.00014.