Electronic Research Archive

http://www.aimspress.com/journal/era

ERA, 32(4): 2310–2322.
DOI: 10.3934/era.2024105
Received: 08 October 2023
Revised: 12 March 2024
Accepted: 13 March 2024
Published: 22 March 2024

Research article

SH-GAT: Software-hardware co-design for accelerating graph attention networks on FPGA

Renping Wang[1], Shun Li[1,*], Enhao Tang[1], Sen Lan[2], Yajing Liu[1], Jing Yang[1], Shizhen Huang[1] and Hailong Hu[1,*]

[1] College of Physics and Information Engineering, Fuzhou University, Fuzhou 350108, China
[2] College of Science, Shantou University, Shantou 515603, China

* **Correspondence:** Email: 211127144@fzu.edu.cn, huhl@fzu.edu.cn.

Abstract: Graph convolution networks (GCN) have demonstrated success in learning graph structures; however, they are limited in inductive tasks. Graph attention networks (GAT) were proposed to address the limitations of GCN and have shown high performance in graph-based tasks. Despite this success, GAT faces challenges in hardware acceleration, including: 1) The GAT algorithm has difficulty adapting to hardware; 2) challenges in efficiently implementing Sparse matrix multiplication (SPMM); and 3) complex addressing and pipeline stall issues due to irregular memory accesses. To this end, this paper proposed SH-GAT, an FPGA-based GAT accelerator that achieves more efficient GAT inference. The proposed approach employed several optimizations to enhance GAT performance. First, this work optimized the GAT algorithm using split weights and softmax approximation to make it more hardware-friendly. Second, a load-balanced SPMM kernel was designed to fully leverage potential parallelism and improve data throughput. Lastly, data preprocessing was performed by pre-fetching the source node and its neighbor nodes, effectively addressing pipeline stall and complexly addressing issues arising from irregular memory access. SH-GAT was evaluated on the Xilinx FPGA Alveo U280 accelerator card with three popular datasets. Compared to existing CPU, GPU, and state-of-the-art (SOTA) FPGA-based accelerators, SH-GAT can achieve speedup by up to 3283×, 13×, and 2.3×.

Keywords: graph; graph attention networks; co-design; FPGA; accelerator

1. Introduction

In recent years, graph neural networks (GNNs) have been applied across various domains, demonstrating remarkable performance in learning from graph-structured data, encompassing networking, biology, recommendation systems, and other fields. Graph convolutional networks (GCNs) [1], drawing inspiration from convolutional neural networks (CNNs), have demonstrated significant potential in real-time tasks, including node classification [2], link prediction [3], and graph classification [4].

However, GCNs predominantly learn parameters related to the graph structure, thereby limiting their effectiveness in handling inductive tasks. To overcome this limitation, the graph attention network (GAT) [5] introduces an attention mechanism that resolves the computational monolithic nature, inflexibility, and unsuitability for inductive tasks observed in GCNs. GATs have demonstrated remarkable performance in the aforementioned tasks, outperforming GCNs. Achieving efficient GAT inference computation poses several challenges. Central processing units (CPUs) excel in control-intensive computational tasks but may not be suitable for highly parallel tasks like GAT. Similarly, efficiently handling logically complex tasks is challenging for graphic processing units (GPUs), resulting in degraded computational performance in GAT due to irregular memory access problems. The slowdown of Moore's and Dennard's laws has prompted a shift toward domain-specific accelerators. Field Programmable Gate Arrays (FPGAs) have become popular for algorithm acceleration platforms due to their reconfigurability, customizability, and flexibility [6]. However, using FPGAs to customize the accelerator for GAT presents the following problems and challenges: 1) The GAT algorithm is not hardware-friendly, leading to significant delays in data splicing and softmax computation; 2) efficient computation of sparse matrix multiplication (SPMM) is challenging; and 3) irregular memory accesses poses a two-fold problem, involving complex addressing operations and generating pipeline stalls that reduce computation efficiency.

Previous researches [7–10] have concentrated on the computation of attention mechanisms, ignoring the potential of partial computational flow reconstruction to improve performance. These researches [7,8,10] use DDR as off-chip memory ([9] is not specific for GAT) and lack specific solutions with high-bandwidth memory (HBM)). However, this work conducts hardware-friendly algorithm optimizations, which involve splitting weights instead of data splicing and utilizing softmax approximation. Moreover, this work designs a load-balanced sparse matrix multiplication kernel that leverages the bandwidth advantage of HBM and maximizes potential parallelism to enhance data throughput. Lastly, this work addresses the issues of pipeline stalls and complex addressing resulting from irregular memory accesses through data pre-fetching, ensuring that the source node and its neighboring nodes are always adjacent. This effectively resolves the problems caused by irregular memory accesses related to complex addressing and pipeline stalls. In summary, this work proposes a software-hardware co-design for GAT (SH-GAT), which contains software optimization and an FPGA-based GAT accelerator designed to achieve more efficient GAT inference. SH-GAT provides the following key contributions:

GAT algorithm optimization: The GAT algorithm is optimized for hardware-friendliness by utilizing weight splitting and SoftMax approximation.

Load-Balanced SPMM: We design a load-balanced SPMM kernel that capitalizes on the bandwidth advantage of HBM and exploits potential parallelism to enhance data throughput.

Competitive performance: SH-GAT outperforms the Intel I7-12700KF CPU, Nvidia RTX3090 GPU, and the state of the art(SOTA) FPGA accelerators with remarkable speedups of up to 3283×, 13×, and 2.3×, while also achieving significantly improved energy efficiency of up to 44,053× and 631× compared with CPU and GPU, respectively.

2. Related work

Recently, many domain-specific architectures have been proposed to partially address the challenges in GNN inference. Autotuning-workload-balancing GCN (AWB-GCN) [11] achieves dynamic workload balancing among the processing engine (PE) using three hardware-based task scheduling mechanisms, demonstrating remarkable performance improvement as an early FPGA-based GNN accelerator. BoostGCN [12] introduces a feature aggregation module and two feature update modules tailored for different sparsity levels, optimizing matrix computation. I-GCN [13] proposes a novel algorithm for graph reconstruction to enhance data locality and matrix operation efficiency by merging nodes with shared neighbors, thereby avoiding redundant operations in the aggregation phase. HyGCN [14] presents a two-stage accelerator designed for memory-intensive aggregation and compute-intensive combination phases.

Traditional matrix multiplication architectures fail to achieve maximum computational efficiency due to the high sparsity of graph data [15, 16]. HBM offers a considerable advantage over DDR when dealing with random memory access and memory-intensive applications, due to its multiple channels that enable multichannel parallel access transfers. For instance, the Xilinx U280 FPGA accelerator card features HBM with 32 channels, each offering a bandwidth of 14.375 GB/s, summing up to 460 GB/s. Additionally, the existing GAT accelerator overlooks the critical challenge posed by graph sparsity, despite reducing the usage of DSP resources through quantization algorithms. For example, S-GAT neglects to consider preprocessing and CPU communication time, while its PE unit fails to leverage the sparsity of the graph for matrix multiplication. H-GAT is an accelerator for GAT based on edge devices. FP-GNN is a hardware adaptive architecture on GNNs, which can implement GAT inference by switching components. FTW-GAT quantizes the weights of GATs to ternary values, but it can't eliminate redundant memory access. All of them are not optimized for GAT's computational flow.

3. Software preprocessing

This section introduces the optimization of the GAT algorithm, graph data format, and data preprocessing methods.

3.1. GAT algorithm optimization

Figure 1 presents a comparison of optimization in the GAT computational flow, showcasing the original GAT formula on the left and our improved GAT formula on the right. The graph attention layer can be divided into two steps: self-attention process and feature aggregation. In the first step, the result is z_{ij}, while the second step calculates the attention coefficients e_{ij}. Let's represent matrix transpose using T and data splicing using $\|$. In the second step, the shared weights are divided α into α_1 and α_2, where α_1 and α_2 are used in the central and neighboring nodes, respectively. Here, z_i and z_j represent the features of the central node and its corresponding adjacent nodes, respectively. The purpose of splitting the shared weight α is to parallelize the computation of z_i and z_j, significantly improving computation efficiency during splicing. The next step is softmax function, replacing the original exponentiation with 2, as it is more hardware-friendly with minimal loss of accuracy [17]. Finally, z_{ij} is reused in the aggregation process, significantly improving the computation efficiency.

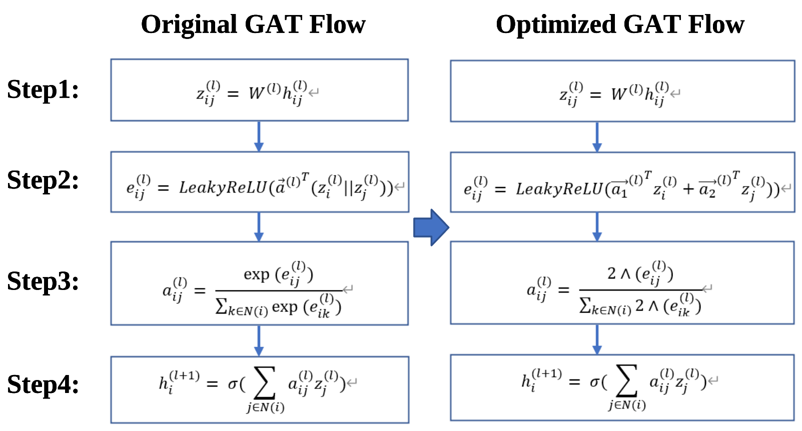
Original GAT Flow
Optimized GAT Flow
Step1:
Step2:
Step3:
Step4:
$z_{ij}^{(l)} = W^{(l)}h_{ij}^{(l)}$
$e_{ij}^{(l)} = LeakyReLU(\vec{a}^{(l)^T}(z_i^{(l)}||z_j^{(l)}))$
$e_{ij}^{(l)} = LeakyReLU(\vec{a_1}^{(l)^T}z_i^{(l)} + \vec{a_2}^{(l)^T}z_j^{(l)}))$
$a_{ij}^{(l)} = \frac{\exp(e_{ij}^{(l)})}{\sum_{k\in N(i)}\exp(e_{ik}^{(l)})}$
$a_{ij}^{(l)} = \frac{2\wedge(e_{ij}^{(l)})}{\sum_{k\in N(i)}2\wedge(e_{ik}^{(l)})}$
$h_i^{(l+1)} = \sigma(\sum_{j\in N(i)} a_{ij}^{(l)}z_j^{(l)})$

Figure 1. Comparison of GAT computational flow optimization.

3.2. *Graph data format*

The feature matrix in GAT is often highly sparse. To conserve storage and reduce computation complexity, feature matrices are compressed to retain only valuable information (nonzero elements). Additionally, computational process needs to simultaneously identify whether each node belongs to a source node or a neighbor node. Building on the aforementioned points, the GCSR format is proposed as a graph data representation. In this work, one row of the adjacency matrix is chosen as a subgraph, which contains the interactions of a node with all other neighboring nodes. There is an overlap between the subgraphs, but this prevents boundary effects from affecting the accuracy of the results. GCSR is utilized to store node features and subgraph location information using three arrays: col-index, value, and node-info, illustrated in Figure 2(c). Figure 2(a) depicts the edge messages of two subgraphs. One subgraph's source node is node$_0$, with neighbors node$_{130}$ and node$_{270}$, while the other subgraph's source node is node$_1$, with neighbors node$_{350}$ and node$_{450}$. Figure 2(b) illustrates the features of each node. The col-index array stores the column indices of the nonzero elements, and the value array retains only nonzero elements. The node-info array comprises row-length and node-flag, with row-length denoting the number of nonzero elements in each row, and node-flag identifying whether the current node is a source or a neighbor. GCSR is well-suited for GNN parallel pipeline computing, enabling the transmission of node features in multiple channels to achieve higher bandwidth.

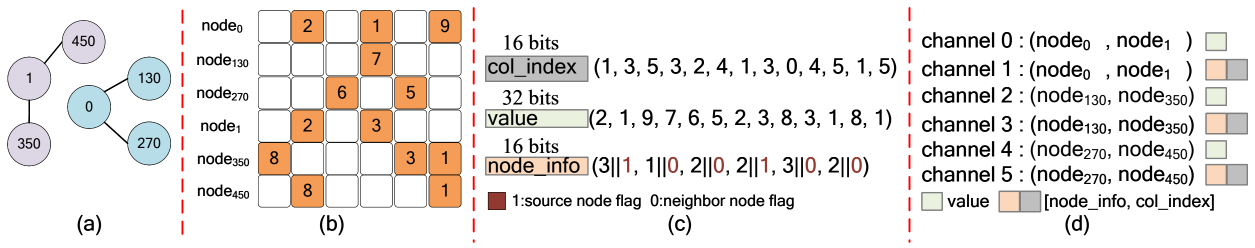
450
1
0
130
350
270
(a)
node0
node130
node270
node1
node350
node450
(b)
16 bits
col_index (1, 3, 5, 3, 2, 4, 1, 3, 0, 4, 5, 1, 5)
32 bits
value (2, 1, 9, 7, 6, 5, 2, 3, 8, 3, 1, 8, 1)
16 bits
node_info (3||1, 1||0, 2||0, 2||1, 3||0, 2||0)
1:source node flag 0:neighbor node flag
(c)
channel 0 : (node0 , node1)
channel 1 : (node0 , node1)
channel 2 : (node130, node350)
channel 3 : (node130, node350)
channel 4 : (node270, node450)
channel 5 : (node270, node450)
value [node_info, col_index]
(d)

Figure 2. GCSR Description. (a) edge message; (b) node features; (c) GCSR data compression format; (d) node features for each AXI channel.

Figure 2(d) illustrates our parallel strategy for transmitting node features using multiple channels. Two channels in a group, with one channel dedicated to transmitting values and the other for sending merged node-info and col-index. Employing three groups of channels (6 channels in total), the first group handles the transmission of source nodes, while the other groups are responsible for transmitting neighbor nodes. To ensure subgraph information integrity, the subgraph's node information is limited to fewer channels than the three groups provide, keeping the remaining channels idle. However, this does not imply that the remaining channels will remain idle indefinitely. In cases where subsequent subgraph node information exceeds the capacity of the three groups of channels, the extra node information will occupy the previously idle channels.

3.3. Data preprocessing

In the data preprocessing phase, our primary focus is on mitigating the irregular access of edge messages for z_{ij}. The edge message (adjacency matrix) stores the connection relationships of each node. As observed in Figure 1, during the computation of z_{ij}, the features z_j of the neighboring nodes corresponding to the current z_i (source node) need to be pre-fetched based on the edge messages for the subsequent computation step. However, this places pressure on both hardware aspects: 1) To maintain high parallelism, z_{ij} is simultaneously accessed in multiple on-chip memories to ensure unimpeded throughput for subsequent parallel computations. However, irregular memory accesses in multiple on-chip memories entail significant control overhead. 2) Irregular memory accesses lead to pipeline stalls when computing the attention coefficient e_{ij}, reducing computation efficiency. This is because when z_{ij} is irregularly accessed, it must wait until all z_{ij} computations are completed. Data preprocessing is performed to address the aforementioned problems. Specifically, we pre-fetch the features of the source node and its neighboring nodes are pre-fetched based on the edge messages, ensuring that the features of the source node and its neighbors remain adjacent. This eliminates the necessity for irregular and complex addressing of z_{ij} and resolves the associated complex addressing operations and pipeline stall problems caused by irregular access. As a result, the overall computation remains unimpeded, enabling the neighbor nodes required by the source node to be computed first without waiting for all z_{ij} calculations to be completed. Additionally, it partially alleviates the on-chip memory pressure.

For illustration, there is a small-scale computation as an example. Figure 3(a) displays the edge message, while Figure 3(b) depicts the challenges encountered without data preprocessing. On the other hand, Figure 3(c) illustrates the process after data preprocessing. Without data preprocessing, the features h are entered individually into SPMM, resulting in the derivation of z_{ij}. To enable parallel reading and writing, z_{ij} is stored in ram_0 to ram_2. However, this gives rise to two challenges: 1) the complex addressing operations associated with irregular memory accesses. While random access to a single ram using an address as an index is feasible, performing random memory access to multiple rams is difficult due to the need to index the ram numbers across multiple rams and index the addresses within the same ram. 2) Irregular memory accesses cause pipeline stalls. After deriving the source node z_0, the neighbor nodes z_{130} and z_{270} must wait a considerable time to access it, leading to a pipeline stall. To overcome these challenges, data pre-fetching is implemented, the features of the source node h_0 and its neighboring nodes h_{130}, h_{270} are pre-fetched based on the edge messages, ensuring that these features remain adjacent. Consequently, irregular and complex addressing of z_{ij} is unnecessary. This optimization significantly improves overall computational efficiency and speed, and reduces on-chip memory utilization.

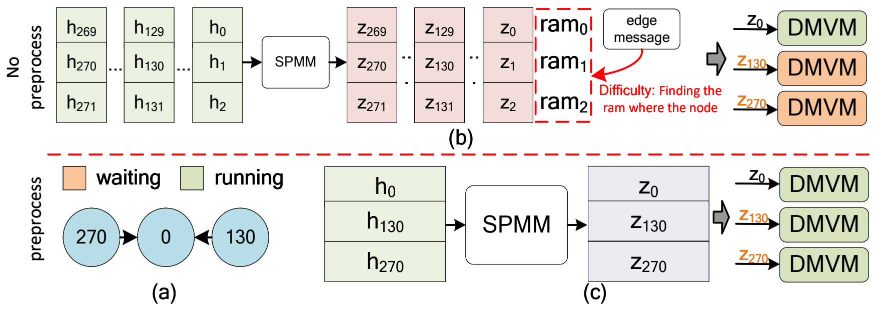

Figure 3. Comparison of data preprocessing. (a) the edge message; (b) no preprocessing; (c) preprocessing method.

4. Hardware architecture

4.1. Overview

This section focuses on the hardware architecture of SH-GAT. Figure 4 provides an overview of the proposed hardware architecture. The HBM contains all the input data, including the weight matrix, α vector, and compressed feature vector (in GCSR format). The memory controller reads this input data in parallel from the 15 AXI channels (CH) and caches it to the on-chip memory. The schedule handles data collation for features and data partitioning for weights. It includes separate loaders for each input data, such as the feature loader, weight loader, and α loader. Each loader is equipped with a corresponding buffer. In the GCSR format, a graph is divided into multiple subgraphs for storage, and when multiple subgraphs loaded fill up the buffer, the loader stops loading data until a subgraph is computed. The workload is then assigned to the computational engine, which comprises self-attention and an aggregator. The self-attention module computes z_{ij}, e_i, e_j, and α_{ij} using five components: loaders, buffers, SPMM, dense matrix vector multiplication (DMVM), and activation function (AF) module. The weight loader and feature loader assign weights and features to SPMM, resulting in the computation of z_{ij}. Subsequently, z_{ij} is calculated in the DMVM array using the α vector to produce e_i and e_j. e_i and e_j are further computed by the leaky ReLU and softmax to obtain α_{ij}. In the aggregator, z_{ij} is reused, and z_{ij} along with α_{ij} performs the final operation in the SPMM to obtain the updated feature $h^{(l+1)}$. The updated feature $h^{(l+1)}$ is subsequently written back to the HBM channels 15 to 20 through the memory controller for the next computation layer. Detailed descriptions of our schedule, SPMM, DMVM array, and softmax will be presented in the following sections.

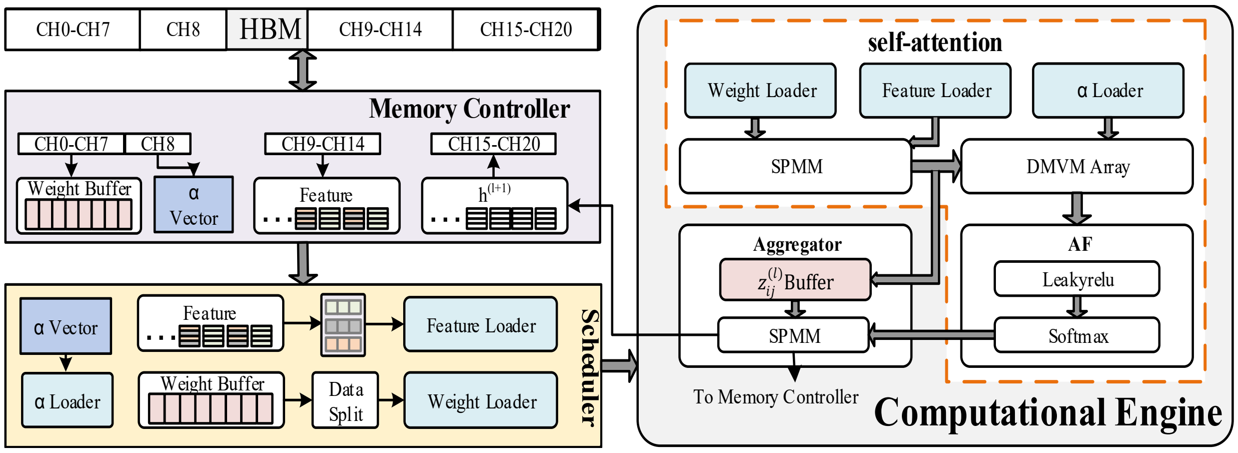

Figure 4. The overall architecture and workflow of SH-GAT.

4.2. *Schedule and SPMM*

Schedule. The hardware architecture of schedule and SPMM is illustrated in Figure 5. The schedule includes loaders for each input data, namely, the feature loader and weight loader, each equipped with a buffer to cache the data. The weight loader sequentially inputs the weight from columns W_0 to W_n using counters, where W_0 represents the first column of weight data. To address data conflict problems arising from sparse matrices' irregularity, the weight loader copies weights, enabling highly parallel computation and effectively resolving data conflict issues. The feature loader uses scheduling to distribute features to SP-PE arrays. The feature is represented by node-info, col-index, and value using the GCSR format, as shown in Figure 2. First, node-info prepares the corresponding col-index and value for each row in advance. If a SP-PE completes its computation, it returns the SP-PE completion signal and its corresponding SP-PE address to notify the schedule to send the data. The schedule then packages the distributed feature (node-info, col-index, and value) and sends it to the corresponding SP-PE. This scheduling strategy resolves the load-balancing problem and ensures efficient computation utilizing the enormous computational resources available.

SPMM. Figure 4 illustrates our approach of performing three groups of HBM channel parallel transmission accesses to the features. The SPMM consists of multiple sets, each containing three SP-PEs used to compute the sparse matrix-vector multiplication (SPMV). Figure 5(a) depicts the detailed architecture of the SP-PE module. The multiplier uses the vector value indexed by the col-index. After multiplication, the mux determines whether to continue accumulation or output based on the value of node-info. SPMM adopts a highly parallel strategy, and SP-PE adopts a fully streaming architecture. SP-PEs operate in a standalone mode dedicated to performing the computation of the inner product of a node feature with a column in the weight matrix. This design allows each SP-PE to signal the scheduler to assign the next pending node feature in the subgraph as soon as it completes the computation of the current node feature. Since there is no data dependency among the SP-PEs in this computational mode, SP-PEs are able to realize parallel and independent computation, which greatly improves the overall computational efficiency. Our scheduling strategy effectively solves the load-balancing problem, contributing to the overall computational efficiency of SPMM.

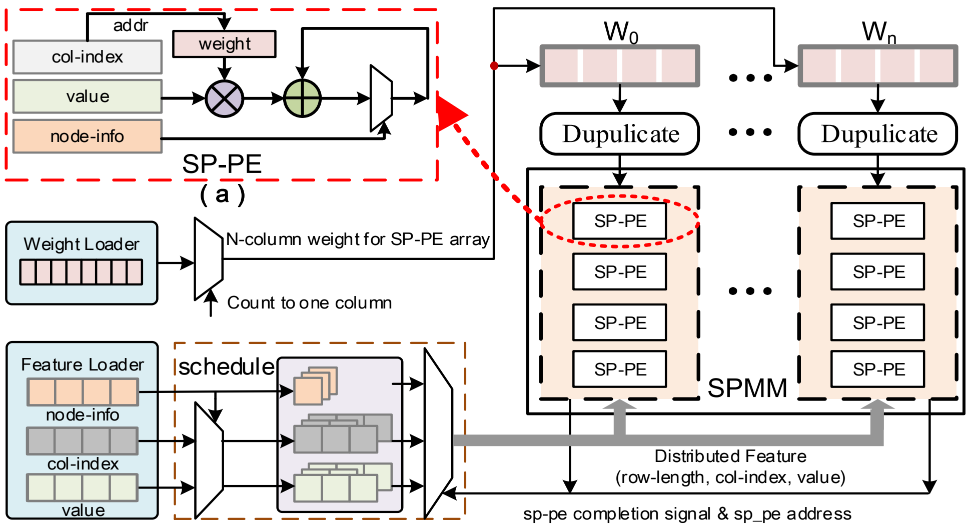

Figure 5. The hardware architecture for data scheduling and SPMM.

4.3. *DMVM array, softmax, and aggregator*

DMVM array. Figure 4 illustrates our approach of performing three sets of HBM channel parallel transmission accesses to the features. The DMVM comprises three DMVMs, and its overall architecture is presented in Figure 6(a). The DMVM is responsible for performing the vector inner product. To achieve this, the α loader splits α into α_1 and α_2 based on the source node flag. Upon receiving the data from α_1, α_2, and z_{ij}, the multiplier initiates the multiplication operation, and the resulting products are sent to the additional tree for summation.

Softmax. After the DMVM calculation, the data is fed into the AF module to obtain e_i and e_j. The AF module plays a crucial role in the GAT's activation function, encompassing leakyrelu and softmax. The operation of softmax is:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \Rightarrow \frac{2^{z_i}}{\sum_{j=1}^{K} 2^{z_j}}. \tag{4.1}$$

Regarding the softmax operation, this work optimized it by replacing the original power operation with a shift from e to 2, making it more hardware-friendly. Given that our PEs section follows a full pipeline structure, this work designed the softmax accelerator accordingly to maintain the overall pipeline structure. The shift operation with a power of 2 efficiently conserves resources without compromising accuracy. Figure 6(b) illustrates the comprehensive softmax architecture, comprising shift registers, an adder, and a divider. Initially, the input data is bifurcated into two paths following the shift operation. One path enters the add-tree for summation, while the other is directed to the register to await the division operation. Upon the completion of summation, the final division operation is executed to derive the value of α_{ij}.

Aggregator. This module is responsible for aggregating z_{ij} using the corresponding α_{ij}. As z_{ij} is already computed during the self-attention, the aggregator efficiently reuses this data via the z_{ij} buffer. The aggregation process in this module is also calculated using the SPMM. Once the aggregator receives the relevant α_{ij} values from the attention mechanism module, it performs the aggregation and sends the resulting output to the memory controller. By reusing the α_{ij} data in the aggregator, the amount of redundant computation is significantly reduced, leading to improved overall performance.

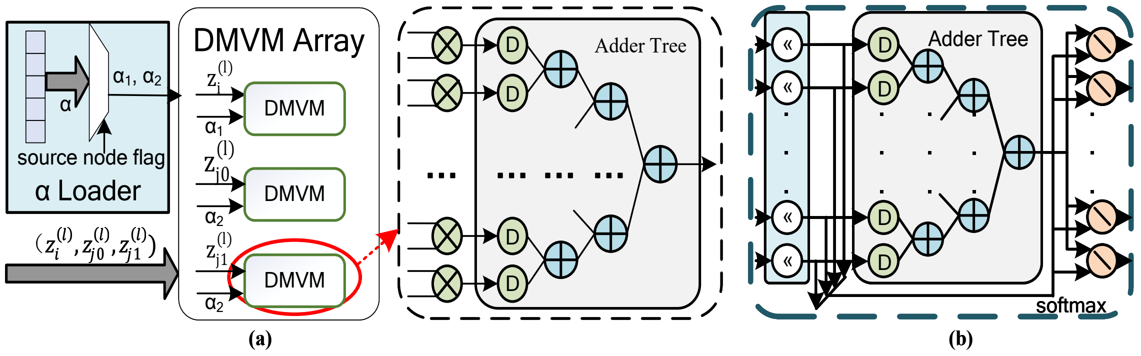

Figure 6. (a) The hardware architecture of DMVM array; (b) The hardware architecture of softmax.

5. Evaluation

5.1. Experiment setup

The proposed SH-GAT are implemented using Verilog HDL. To evaluate the performance of SH-GAT, we successfully deployed SH-GAT on a Xilinx FPGA Alveo U280 accelerator card. SH-GAT was fully evaluated on three datasets: Table 1 lists the size and sparsity of the datasets. We conduct a comprehensive comparison of SH-GAT with Intel(R) Xeon(R) Gold 5218R CPUs (CPU) and NVIDIA RTX3090 GPUs (GPU). We compare SH-GAT with the SOTA FPGA-based GAT accelerator FTW-GAT. Additionally, for comprehensive comparison, we also evaluate it against the other advanced accelerators, FP-GNN, H-GAT, and S-GAT.

Table 1. Dimensions and densities of widely used datasets.

Datasets	Nodes	Edges	Input feature	Classes	Feature density	Edge density	Weight density
Cora	2708	10,556	1433	7	1.3%	0.14%	100%
CiteSeer	3327	9104	3703	6	0.8%	0.08%	100%
PubMed	19,717	88,648	500	3	10.4%	0.02%	100%

5.2. Softmax accuracy loss

We approximate the softmax function, and Table 2 presents the accuracy loss across datasets which reveals that our average accuracy loss is close to 0.01. This computation achieves better hardware performance with very little loss of accuracy because the exponential computation is replaced with a shift operation.

Table 2. Accuracy loss across datasets.

Dataset	Cora	Citeseer	Pubmed
Accuracy loss	0.15	0.006	0.012

5.3. Comparison with CPU and GPU

The comparison on latency is shown in Figure 7(a). Additionally, speedup ratios are computed using logarithm (base 10) for graph visibility. The SH-GAT outperforms the CPU by an average speedup of 3283× and the GPU by 13× under the GAT model using various datasets. SH-GAT demonstrates excellent performance with efficient data preprocessing, algorithm optimization, and high throughput computing units. However, its performance is slightly reduced on the GPU side for the PubMed dataset, which contains the largest matrix in the dataset. Figure 7(b) presents a comparison of the energy efficiency of our design on various platforms. Compared to CPU and GPU, SH-GAT exhibits significantly better energy efficiency, with an average improvement of 44,053× and 631×, respectively. The energy-efficiency improvement ratio is logarithmically computed (base 10). The computation unit features a full pipeline architecture, and this work has effectively addressed the load-balancing problem through scheduling. Furthermore, each SP-PE is equipped with an enable port and a gated clock, ensuring that modules not involved in the computation remain inactive to conserve power consumption.

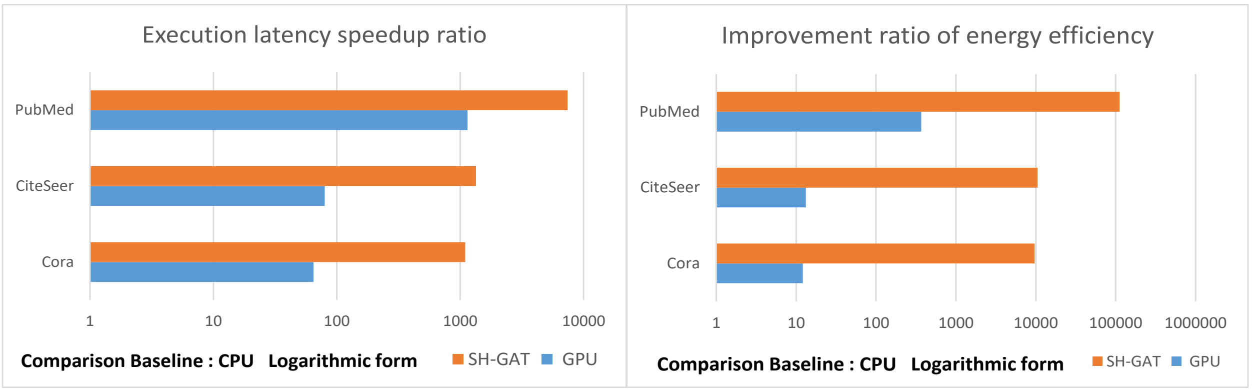

Figure 7. Comparison with CPU and GPU implementations. (a) speedup ratio comparison; (b) improvement ratio of energy efficiency comparison.

5.4. Comparison with FPGA-based accelerators

This work assesses the overall resource consumption and latency of SH-GAT compared to FPGA-based GAT accelerators S-GAT, H-GAT, FTW-GAT, and an overlay accelerator FP-GNN using the Cora, CiteSeer, and PubMed datasets. The model is set to two layers and the feature dimension is set to 16. Table 3 demonstrates that SH-GAT achieves a speedup of 2.3× compared to FTW-GAT, the SOTA accelerator. S-GAT fails to leverage the sparsity of graph data and the high parallelism in the PE design. H-GAT is limited by the resources of the edge device. FTW-GAT and FP-GNN don't optimize GAT's computational flow, resulting in a lot of redundant memory access. SH-GAT reduces redundant memory accesses by optimizing the computational flow of GAT and the compression format, and enables efficient parallel computation through an HBM-based architecture. Despite FTW-

GAT and FP-GNN utilizing more DSPs, SH-GAT still outperforms them due to our efficient data preprocessing, which avoids complex addressing and pipeline stall issues, ensuring SH-GAT maintains efficient computation. Additionally, the substantial data read from HBM is optimally utilized through GCSR's efficient data allocation, resulting in significant improvements in computational regularity. This approach reduces the dependency on DSPs, enabling better performance with just 732 DSPs, while FP-GNN heavily relies on DSPs and experiences reduced operation frequency. Table 4 shows that our dependence on on-chip resources is low, which indirectly implies that our power consumption is relatively low.

Table 3. Comparison with FPGA-based accelerators on latency.

Datasets	Latency (us)				
	SH-GAT (Ours)	FTW-GAT [10]	FP-GNN [9]	H-GAT [8]	S-GAT [7]
Cora	19.4	44.9	46.3	600	6400
CiteSeer	22.1	50.8	71.4	800	N/A
PubMed	150.2	339	616	5700	N/A

Table 4. Comparison with FPGA-based accelerators on resource utilization.

Accelerators	FPGA	Frequency	LUT	FF	BRAM	DSP
SH-GAT (Ours)	Alveo-U280	225 MHz	110 K	125 K	1428	732
FTW-GAT [10]	VCU128	225 MHz	437 K	470 K	1502	1216
FP-GNN [9]	VCU128	225 MHz	1068 K	727 K	1792	8740
H-GAT [8]	K325T	200 MHz	39 K	42 K	119	244
S-GAT [7]	Inspur F10A	216 MHz	250 K	338 K	683	148

6. Conclusions

This work proposes SH-GAT, an FPGA-based accelerator for GAT that prioritizes high-throughput and energy-efficiency. SH-GAT employs algorithmic optimization, efficient data preprocessing, and a high-throughput, load-balanced computation engine to execute GAT inference efficiently. Algorithmic optimization enhances hardware-friendliness in GAT computation, while data preprocessing resolves issues arising from irregular memory accesses. Additionally, a high-throughput and load-balanced SPMM kernel is designed to leverage HBM's bandwidth advantage and exploit potential parallelism, thus enhancing data throughput. SH-GAT exhibits high energy efficiency and considerable scalability potential for GAT inference computation. This is attributed to the presence of multiple memory channels and the ability to add parallel computational units. In comparison to CPU and GPU, SH-GAT attains speedups of up to 3283× and 13× and exhibits energy efficiency improvements of up to 44,053× and 631×, respectively. Moreover, SH-GAT outperforms the SOTA FPGA-based GAT accelerator by achieving a 2.3× speedup. Although these optimizations result in performance improvements, they inevitably introduce some overheads, such as data preprocessing in GCSR formats, etc, and as the structure of the GAT model changes, such specialized accelerators will face the challenge of adapting to the new model structure for agility designs. Moreover, the explosion of information in the real world is making dataset sizes ever larger. For very large graphs, a multilevel graph partitioning approach may be required. Moreover, it is necessary to consider how to reduce the impact of boundary effects and loss of information, which will be a future research direction.

Use of AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

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

Conflict of interest

The authors declare there are no conflicts of interest.

References

1. T. N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, preprint, arXiv:1609.02907.
2. S. Abu-El-Haija, A. Kapoor, B. Perozzi, J. Lee, N-GCN: Multi-scale graph convolution for semi-supervised node classification, in *Proceedings of The 35th Uncertainty in Artificial Intelligence Conference*, **115** (2020), 841–851. Available from: https://proceedings.mlr.press/v115/abu-el-haija20a.html.
3. M. Zhang, Y. Chen, Link prediction based on graph neural networks, in *Advances in Neural Information Processing Systems*, **31** (2018), 5171–5181. Available from: https://proceedings.neurips.cc/paper_files/paper/2018/file/53f0d7c537d99b3824f0f99d62ea2428-Paper.pdf.
4. M. Zhang, Z. Cui, M. Neumann, Y. Chen, An end-to-end deep learning architecture for graph classification, in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, **32** (2018), 4438–4445. https://doi.org/10.1609/aaai.v32i1.11782
5. P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, Graph attention networks, preprint, arXiv:1710.10903.
6. R. Chen, H. Zhang, Y. Li, R. Zhang, G. Li, J. Yu, et al., Edge FPGA-based onsite neural network training, in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, (2023), 1–5. https://doi.org/10.1109/ISCAS46773.2023.10181582
7. W. Yan, W. Tong, 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)*, (2020), 661–666. https://doi.org/10.1109/ICPADS51040.2020.00093
8. S. Huang, E. Tang, S. Li, H-GAT: A hardware-efficient accelerator for graph attention networks, *J. Appl. Sci. Eng.*, **27** (2023), 2233–2240. http://dx.doi.org/10.6180/jase.202403_27(3).0010

9. T. Tian, L. Zhao, X. Wang, Q. Wu, W. Yuan, X. Jin, FP-GNN: Adaptive FPGA accelerator for graph neural networks, *Future Gener. Comput. Syst.*, **136** (2022), 294–310. https://doi.org/10.1016/j.future.2022.06.010
10. Z. He, T. Tian, Q. Wu, X. Jin, FTW-GAT: An FPGA-based accelerator for graph attention networks with ternary weights, *IEEE Trans. Circuits Syst. II Express Briefs*, **70** (2023), 4211–4215. https://doi.org/10.1109/TCSII.2023.3280180
11. T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, et al., 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. https://doi.org/10.1109/MICRO50266.2020.00079
12. B. Zhang, R. Kannan, 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)*, (2021), 29–39. https://doi.org/10.1109/FCCM51124.2021.00012
13. T. Geng, C. Wu, Y. Zhang, C. Tan, C. Xie, H. You, et al., I-GCN: A graph convolutional network accelerator with runtime locality enhancement through Islandization, in *54th Annual IEEE/ACM International Symposium on Microarchitecture*, (2021), 1051–1063. https://doi.org/10.1145/3466752.3480113
14. M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, et al., HyGCN: A GCN accelerator with hybrid architecture, in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (2020), 15–29. https://doi.org/10.1109/HPCA47549.2020.00012
15. Y. Gao, L. Gong, C. Wang, T. Wang, X. Zhou, SDMA: An efficient and flexible sparse-dense matrix-multiplication architecture for GNNs, in *2022 32nd International Conference on Field-Programmable Logic and Applications (FPL)*, (2022), 307–312. https://doi.org/10.1109/FPL57034.2022.00054
16. R. Chen, H. Zhang, Y. Ma, J. Chen, J. Yu, K. Wang, eSSpMV: An embedded-FPGA-based hardware accelerator for symmetric sparse matrix-vector multiplication, in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, (2023), 1–5. https://doi.org/10.1109/ISCAS46773.2023.10181734
17. Z. Xu, J. Yu, C. Yu, H. Shen, Y. Wang, 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)*, (2020), 33–37. https://doi.org/10.1109/FCCM48280.2020.00014