Graph-OPU: A Highly Integrated FPGA-Based Overlay Processor for Graph Neural Networks

Ruiqi Chen[1], Haoyang Zhang[1], Shun Li[2], Enhao Tang[2], Jun Yu[1], Kun Wang[1,*]
[1]State Key Lab of ASIC & System, Fudan University, Shanghai, China
[2]College of Physics and Information Engineering, Fuzhou University, Fuzhou, China
*kun.wang@ieee.org

2023 33rd International Conference on Field-Programmable Logic and Applications (FPL) | 979-8-3503-4151-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/FPL60245.2023.00039

***Abstract*—Field-programmable gate array (FPGA) is an ideal candidate for accelerating graph neural networks (GNNs). However, FPGA reconfiguration is a time-consuming process when updating or switching between diverse GNN models across different applications. This paper proposes a highly integrated FPGA-based overlay processor for GNN accelerations. Graph-OPU provides excellent flexibility and software-like programmability for GNN end-users, as the executable code of GNN models are automatically compiled and reloaded without requiring FPGA reconfiguration. First, we customize the instruction sets for the inference qprocess of different GNN models. Second, we propose a microarchitecture ensuring a fully-pipelined process for GNN inference. Third, we design a unified matrix multiplication to process sparse-dense matrix multiplication and general matrix multiplication to increase the Graph-OPU performance. Finally, we implement a hardware prototype on the Xilinx Alveo U50 and test the mainstream GNN models using various datasets. Graph-OPU takes an average of only 2 minutes to switch between different GNN models, exhibiting average 128× speedup compared to related works. In addition, Graph-OPU outperforms state-of-the-art end-to-end overlay accelerators for GNN, reducing latency by an average of 1.36× and improving energy efficiency by an average of 1.41×. Moreover, Graph-OPU achieves up to 1654× and 63× speedup, as well as up to 5305× and 422× energy efficiency boosts, compared to implementations on CPU and GPU, respectively. To the best of our knowledge, Graph-OPU represents the first in-depth study of an FPGA-based overlay processor for GNNs, offering high flexibility, speedup, and energy efficiency.**

I. Introduction

Graph Neural Networks (GNNs) have been applied in numerous fields, including recommendation systems, modeling physics systems, traffic prediction, and compound-protein interactions [1]. With its abundant parallel computing resources and high energy efficiency, field-programmable gate array (FPGA) is an ideal candidate for accelerating graph neural networks (GNNs) [2]. However, with the increasing complexity of the data dimensions and structures, accelerating one specific GNN model is no longer sufficient. Different GNN models are utilized within the same task to get better results [3, 4]. Fast model switching on FPGA-based GNN accelerators has become a new challenge.

Many FPGA-based works to accelerate GNN inference have emerged in recent years (e.g., AWB-GCN [5], LW-GCN [6], BoostGCN [7] and I-GCN [8]). While these accelerators can significantly improve the performance of GNNs, these accelerators are only designed to support a specific GNN model and their scalability is limited. Recent studies have tried to design the FPGA-based end-to-end overlay accelerator for GNN to be compatible with different GNN models. DeepBurning-GL [9] provides a set of pre-built templates that can be fused and parameterized by users to generate the final accelerator design. FP-GNN's [10] Adaptive GNN Accelerator (AGA) framework utilizes a unified processing module that can support the different phases in GNN models. FlowGNN [11] proposes a novel and scalable dataflow architecture that flexibly supports multiple GNN models based on message passing mechanisms. However, all of these designs require a considerable amount of reconfiguration time, involving preparation, logic synthesis, placement and routing, implementation, and generating the bitstream file [12]. It means that users may encounter a long wait time when switching GNN models. In addition, the ideal design should not compromise performance for flexibility.

Motivated by the emerging requirements, we propose Graph-OPU, a highly integrated FPGA-based overlay processor for GNN inference. To the best of our knowledge, Graph-OPU is the first hardware architecture specifically designed for mainstream GNNs with software programmability. Graph-OPU mainly consists of our customized instruction sets and several designs at the microarchitecture level that effectively eliminate the reconfiguration process. Moreover, Graph-OPU maintains excellent performance and scalability. To summarize, our contributions are listed as follows:

- **User Friendliness**: We design an overlay processor for various mainstream GNN models (Table I). To the best of our knowledge, Graph-OPU is the first FPGA-based general processor for GNN acceleration, supporting quick model switching for users.
- **Software-Hardware Co-optimization**: We propose a GNN-specific instruction set architecture (ISA) and microarchitecture, optimized for computation and data communication to increase the overall efficiency.
- **High Computation Efficiency**: We design a unified matrix multiplication for sparse matrix multiplication (SpMM) and general matrix multiplication (GEMM) based on the optimized PCOO format.
- **Competitive Performance with Flexibility**: Graph-OPU outperforms related works with a significant speedup of 128× when switching between different GNN models. Graph-OPU also outperforms the state-of-the-art (SOTA) end-to-end overlay accelerators for GNN by 1.36× latency reduction and 1.41× energy efficiency improvement on average. Compared with CPU and GPU, Graph-OPU achieves up to 1654× and 63× speedup, as well as up to 5305× and 422× better energy efficiency, respectively.

* Corresponding author.

1946-1488/23/$31.00 ©2023 IEEE
DOI 10.1109/FPL60245.2023.00039

Authorized licensed use limited to: Vrije Universiteit Brussel. Downloaded on November 07,2025 at 12:42:58 UTC from IEEE Xplore. Restrictions apply.

TABLE I
INFERENCE PHASES OF DIFFERENT GNN MODELS

Model	Edge Features Extraction (EFE)	Node Features Aggregation (NFA)	Node Features Combination (NFC)
GCN [13]	$M_{uv} = EFE\{H_u\} = H_u$	$A_v = NFA(M_{uv}) = \sum M_{uv}$	$H_v = NFC\{\sigma\{W(H_v, A_v)\}\}$
LightGCN [14]	$M_{uv} = EFE\{H_u\} = H_u$	$A_v = NFA(M_{uv}) = \sum M_{uv}$	$H_v = A_v$
GAT [15]	$Z^k_{uv} = W^k_e H_u, e_{uv} = LR\left(a^k \cdot Concat\left(W^k_e H_v, Z^k_{uv}\right)\right)$	$\alpha^k_u = Softmax\left(e^k_{uv}\right), A^k_v = \sum \alpha^k_v Z^k_{uv}$	$H_v = \sigma\left(Concat\left(A^k_v\right)\right)$
GIN [16]	$M_{uv} = EFE\{H_u\} = H_u$	$A_v = NFA\{M_{uv}\} = \sum M_{uv}$	$H_v = NFC\{\sigma\{BN\{W(H_v + A_v)\}\}\}$
GraphSage [17]	$M_{uv} = EFE\{H_u\} = H_u$	$A_v = NFA(M_{uv}) = Mean\{M_{uv}\}$	$H_v = NFC\{\sigma\{W(H_v, A_v)\}\}$

TABLE II
GNN NOTATIONS

Notation	Description	Notation	Description
u/v	Source/ Target node	A_v	Features of neighbors to be aggregated
M_{uv}	Edge message from u and v	W_v/W_e	Weights of nodes/edges
e_{uv}	Edge between u and v	k	Number of GNN layers
H_u/H_v	Features of node u/v	Z_{uv}/a_v	Vector of attention heads/attention weights

TABLE III
COMPARISON OF THE MODEL SWITCHING TIME WITH OTHER OPEN-SOURCE ACCELERATORS.

Accelerators	Models	Avg (min)
FlowGNN	GCN GAT GIN	569.15
Ref [18]	GCN	85.92
Ref [19]	GCN	120.47

II. BACKGROUND AND MOTIVATION

A. Background

Table II defines the notations in GNN inference phases. The inference processes of these GNN models can be summarized in three steps: (1) Edge Feature Extraction (EFE), which extracts meaningful information from edge attributes for better graph representation; (2) Node Feature Aggregation (NFA), which aggregates neighboring node features to capture local graph structure; and (3) Node Feature Combination (NFC), which combines aggregated features with learnable weights and activation functions to generate expressive node embeddings. As one of the first graph model, GCN [13] directly duplicates source node feature for EFE, sums up all the messages from neighboring nodes to be aggregated for NFA, and combines aggregated feature with local features for NFC. As for other variant, the improvement of LightGCN [14] lies in its simplification of the GCN architecture by removing the learnable weight matrices and non-linear activation functions in its NFA and NFC processes. The enhancement of GAT [15] is that it uses self-attention for more precise edge value calculation in the NFA process. Moreover, the use of LeakyReLU (LR) in GAT can improve the model's performance, especially in cases where the data has subtle negative values and requires more nuanced feature learning. GIN [16] improves the NFA stage by using an MLP and Batchnorm (BN) to aggregate node features, combining a learnable parameter to control the importance of central node features. GraphSAGE [17] adopts different aggregation functions (*e.g.*, $Mean$) to complete the NFA stage, allowing it to learn inductive node embeddings on large graphs.

B. Motivation

To better understand the flexibility of existing solutions, we conducted an extensive survey on various FPGA-based GNN accelerators and overlay accelerators, focusing on evaluating model switching time. This process consists of key stages such as preparation, logic synthesis, placement and routing, implementation, and bitstream file generation. We selected three open-source works developed using Verilog HDL [18] and HLS [11, 19] for testing configuration time. Results indicate a minimum of 85 minutes is needed for reconfiguration (As shown in Table. III and details in Sec V). This substantial time investment is required when users need to adjust configuration parameters or switch models [20]. Therefore, a flexible FPGA-based GNN tool with software-programmability is needed to support using different GNN models in a single task [3, 4] while maintaining competitive performance.

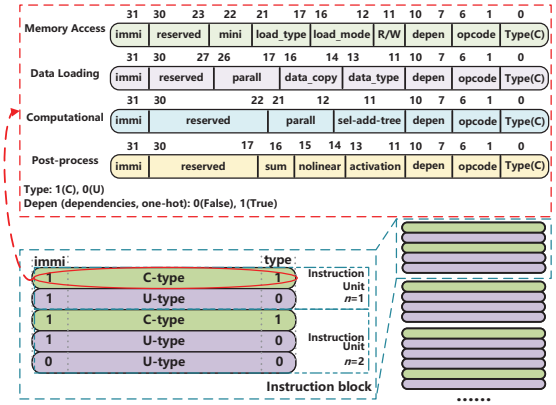

Fig. 1. Configuration of instruction block and instruction unit.

III. INSTRUCTION SET ARCHITECTURE

Graph-OPU's data execution path, determined by instructions, enables compatibility with multiple GNN models, unlike traditional accelerators' singular fixed datapath. GNN operations are adapted based on the OPU ISA framework [21], with supplementary parameters incorporated. Furthermore, we enhance the instruction execution mechanism for increased efficiency on the underlying hardware.

A. Instruction Description

Each of our 32-bit complex instructions is classified into either Conditional (C-type) or Unconditional (U-type) types, as depicted in Fig. 1. C-type instructions determine target operations and set trigger conditions, while U-type instructions provide related operation parameters for their paired C-type. We maintain the instruction unit and block design of the OPU ISA, identifying the lowest ($Type$) and highest ($immi$) bits

Authorized licensed use limited to: Vrije Universiteit Brussel. Downloaded on November 07,2025 at 12:42:58 UTC from IEEE Xplore. Restrictions apply.

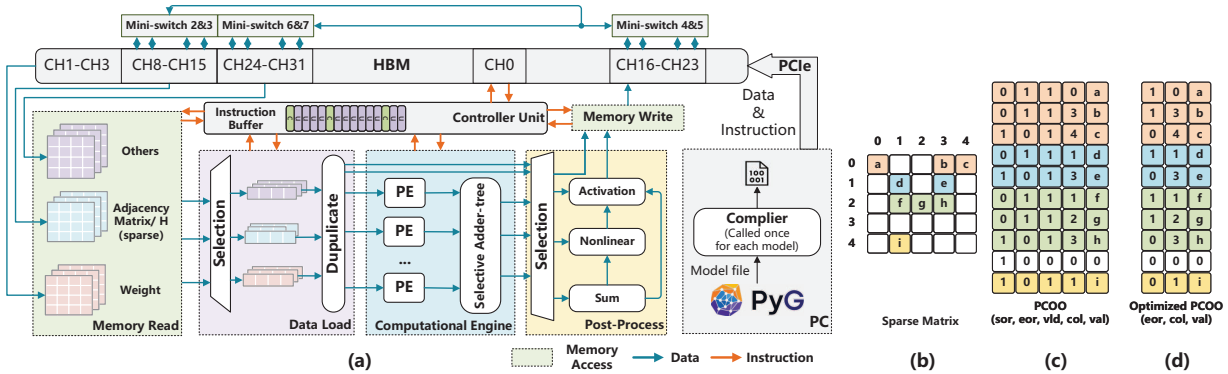

Fig. 2. Overview of Graph-OPU microarchitecture and optimized data formats. (a) Graph-OPU microarchitecture; (b) the sparse matrix; (c) PCOO format for sparse matrix; (d) optimized PCOO format for sparse matrix.

of each instruction. This design is retained due to the frequent data operations in GNNs, allowing for additional operational details and ensuring continuity and integrity of operations to the greatest extent.

Matrix operations in GNNs often result in numerous identical instructions within an instruction block. GNNs exhibit data structure and operation irregularities, such as alternating between SpMM and GEMM. Thus, we divide C-type instructions into four categories using 2-bit in *opcode*: memory access, data loading, calculation, and post-processing, corresponding to the micro-architecture. Instructions are executed in their respective modules, and a 4-bit one-hot vector (*depen*) expresses dependencies among these modules, enabling better parallelism in GraphOPU operations.

1) *Memory Access* manages data transfer between High Bandwidth Memory (HBM) and onboard memory, as well as the reverse transfer from onboard memory to HBM. The operation mode is switched using the $load\_mode$ bit, while $data\_type$ identifies the data source, which can be sparse data, weight, or other types. The mini parameter controls the HBM mini switch module.
2) *Data Loading* moves data from the onboard memory to the computational module. Its operation pattern can be tailored by adjusting the $data\_copy$ and $parall$ parameters, aligning with the data requirements of the computational module.
3) *Computational* controls all processing engines (PEs) and the selective adder-tree. By utilizing the selective adder-tree, the datapath can be altered to accommodate specific computational processes, such as the attention calculation in GAT, which requires multiple multiplications followed by accumulation. The parallelism of multipliers can be adjusted using the $parall$ parameter, ensuring sufficient design space exploration for different networks.
4) *Post-process* includes activation, non-linear, and intermediate result addition operations. A selected combination of operations is executed when the post-processing is triggered. Activation functions include ReLU, GELU, and LeakyReLU, while non-linear operations consist of batch normalization and softmax.

In addition to the four categories of C-type instructions, several U-type instructions have been designed. These U-type instructions provide more specific parameters in conjunction with their related C-type instructions. Such parameters include memory address, memory size, operation round, and operation dimension.

B. Realization of the Graph-OPU ISA on TVM and LLVM

We utilize TVM [22] and LLVM [23] to convert user's model files (.pth or ONNX) to Graph-OPU ISA. TVM outlines the model's computational steps and schedules them efficiently. Considering the Graph-OPU architecture, TVM optimizes computations for performance and energy efficiency. TVM generates LLVM intermediate representation (IR) code for Graph-OPU hardware, matching vector-type IR code to the microarchitecture's computational engine and intrinsic functions, ensuring efficient microarchitecture support.

LLVM generates Graph-OPU ISA instructions for GNN models using IR and intrinsics. The IR to ISA conversion employs direct mapping. If matching intrinsics are unavailable, instructions can't be divided, and users must devise custom methods for desired operations. For instance, GAT's EFE operation involves twice sequential multiplications and additions, which are mapped to LLVM intrinsics using initial results as weight input for subsequent operations.

Compared to model switching through re-configuration, the OPU-ISA-based compiler has demonstrated significantly reduced time consumption. Detailed results of the compiler time consumption are presented in Sec V.

IV. Microarchitecture

The hardware modules in the microarchitecture of Graph-OPU are fixed. At runtime, different GNN model operations are implemented according to the instruction updates and by controlling the transformation of the datapath. Therefore, the hardware modules in Graph-OPU are directly controlled by the parameters in the instructions, which also means that Graph-OPU can explore more parallel combinations. As shown in Fig. 2(a), the microarchitecture of Graph-OPU includes a memory access module, data loading module, computational engine module, and post-process module.

Authorized licensed use limited to: Vrije Universiteit Brussel. Downloaded on November 07,2025 at 12:42:58 UTC from IEEE Xplore. Restrictions apply.

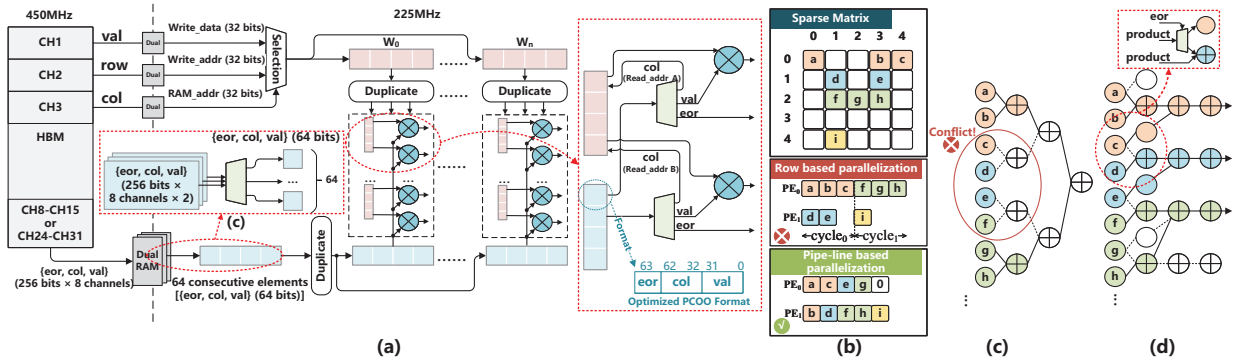

Fig. 3. Computational Engine Module. (a) The architecture of the PE with HBM datapath; (b) the PE parallelization to compute on a sparse matrix; (c) traditional adder-tree with conflict; (d) the selective adder-tree.

A. Memory Access Module

The performance of FPGA-based GNN accelerators is often bottlenecked by memory bandwidth [24, 25] due to the large amounts of data that need to be stored and transferred. While High-Bandwidth Memory (HBM) provides sufficient memory bandwidth, it requires customized access mechanisms to fully utilize FPGA resources. Hence, we design a memory access module that can efficiently read from and write to HBM, catering to the data requirements of our system.

The corresponding data and instructions are generated by the user's PyTorch [26] model file on the PC side, are transmitted via the PCIe, and stored in the HBM. The HBM operates at 450MHz and utilizes a total of 28 channels with 256-bit per channel. The Graph-OPU runs at 225MHz, which results in a 512-bit width for each channel in the Memory Access Module. With a maximum access number of 21 HBM channels, this design leverages the peak bandwidth (316 GB/s) of the Alveo U50 platform [27], ensuring optimal performance and efficient memory access. The Memory Access Module enables efficient data communication between the HBM and on-chip Block RAM (BRAM), based on relevant instructions. The BRAM stores weight data ($Weight$), sparse data ($Sparse$), and other computation-related data ($Others$). As shown in Fig. 2(a), weight data includes data in dense vector form that participates in computation. For accessing the dense vectors, we utilize three-channels, where one of the channels transfers value, and the other two channels are for the row index and column index corresponding to the values. Since adjacency matrices are extremely sparse, with less than 1% non-zero elements [28], buffering all sparse data in BRAM would consume an impractical amount of storage. Thus, for accessing sparse data ($Sparse$), only buffering and computing the non-zero elements is necessary. Building on the packet-level column-only coordinate-list (PCOO) format in [6] (Fig. 2(c)), we propose an optimized PCOO format to compress the sparse matrix. Our format contains all element information in 64 bits, with 1 bit for the end or row signal (eor), 31 bits for column index (col), and 32 bits for value (val), shown in Fig. 2(d). The memory access module designs leverages 8 HBM channels for accessing the sparse or others, allowing for the transfer of 64 sparse elements in a single operation, which significantly enhances data parallelism.

B. Data Loading Module

The data loading module selects and replicates data to facilitate parallel computation on our architecture, as described in detail in the following section.

To facilitate parallel computation on our architecture, the data loading module retrieves data from the on-chip buffer, then reorders and divides it as needed. To ensure data correctness, the module selects data (e.g., A, H) to import into the buffer based on the current operation progress, provided by the C-type instruction. To support our proposed parallel computation strategy, the module replicates sparse elements and dense vectors multiple times. The advantages are twofolds: 1. It enables exploration of a larger parallel space for sparse elements; 2. It prevents reading conflicts for dense vectors. The following section provides further details on this approach.

C. Computational Engine Module

The computational process of GNN involves a significant number of matrix multiplications, such as SpMM and GEMM. The efficiency of the matrix multiplier is critical to the acceleration performance [29]. Therefore, we design the computational engine module to handle these matrix multiplications, which comprises PEs and selective adder-tree, shown in Fig. 3.

We have customized the datapath, storage and sharing of the data to meet the requirements of SpMM and GEMM to fully utilize FPGA resources. Fig. 3(a) illustrates the architecture of the PE that includes HBM datapath. Specifically, we utilize BRAMs to implement a Weight column shared by multiple multipliers. With this design, one PE can perform the multiplication of 64 sparse elements with one Weight column in parallel. To leverage the potential parallelism further, we consider duplicating these 64 elements n times and performing the multiplication with n columns of Weights separately. In our Graph-OPU, we set n to 16, where 8 PE blocks perform the multiplication of 64 sparse elements with 16 columns of distinct weights. Upon the completion of computation for all the 16 column weights, the data loading module updates the weights of the 16 columns, copies and distributes them

Authorized licensed use limited to: Vrije Universiteit Brussel. Downloaded on November 07,2025 at 12:42:58 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
COMPARISON WITH OTHER FPGA-BASED END-TO-END OVERLAY GNN ACCELERATORS

	FPGA	Model	Freq (MHz)	LUT	FF	BRAM	DSP	Efficiency (Throughput/DSPs)
Graph-OPU	Alveo-U50	GCN	225	475K	427K	927	2742	0.157
		Light-GCN						
		GAT						
		GIN						
		GraphSage						
DeepBurning-GL	Alveo-U50	GCN	200	671K	505K	403	5892	0.043
FP-GNN	VCU128	GCN	225	717K	517K	1792	8192	0.126
		GAT		1068K	727K	1792	8740	
FlowGNN	Alveo-U50	GCN	300	229K	192K	185	1048	N/A
		GAT		149K	134K	335	2488	
		GIN		263K	166K	204	1741	

to the 16 different PE blocks. As shown in Fig. 3(b), this design ensures the independence of each PE operation and solves the imbalance problem during sparse matrix operation. Suppose we were using two PEs or threads to compute on a sparse matrix. The intuitive approach to parallelizing SpMM is row-based parallelization [30], also illustrated in Fig. 3(b). This approach requires two steps to process the computation. However, this workload distribution leads to imbalanced resource utilization and energy waste. Our design ensures the independence of each PE operation and alleviates the workload imbalance during sparse matrix operations. Meanwhile, the optimized PCOO format retains compatibility with dense matrices and facilitates fast local encoding through a simple counter mechanism, thereby enabling the datapath to be compatible with GEMM.

As shown in Fig. 3(d), a selective adder-tree based on the reconfigurable adder-tree [31] is employed. This design utilizes the eor signal to alter the datapath, enabling the accurate accumulation of product results from different rows. This approach mitigates the conflicts that arise from varying inputs in traditional adder-trees with fixed datapaths, as illustrated in Fig. 3(c). The selective adder-tree design offers improved flexibility and adaptability for processing diverse inputs without conflicts.

D. Post-process Module

The post-process module consists of three key units: the summation unit, the nonlinear unit, and the activation function unit. The summation unit comprises accumulators and adder-trees, which generate outputs that are used for softmax, GAT attention computation, and other operations. The accumulator can also output the current number of valid data for subsequent operations. The summation unit can flexibly perform various nonlinear operations, such as softmax and batch-normalization, under the control of instructions. To optimize for hardware implementation, we use a base of 2 instead of e for power calculation, enabling us to implement division using shift operation [32], while incurring an insignificant accuracy loss. The activation function unit includes three standard activation functions: ReLU, LeakyReLU, and GELU. These functions are implemented using a multiplier with instruction control.

E. Collaboration Between Different Modules

Graph-OPU implements the inference of GNN models by generating instructions to schedule these four modules. For the inference process of GCN, LightGCN, GIN, and GraphSAGE, EFE and NFA only need to complete the aggregation operation, consisting of SpMM. According to the instructions generated in the model file, SpMM is completed by Graph-OPU in the following process: fetching A (adjacency matrix) and HW (features after transformation), caching, copying and loading them into the computational module, determining whether post-process is needed, and writing the results back to HBM. The NFC stages of GCN and GIN are implemented by GEMM, using our unified computational module. As to complete the unique operations of GEMM, the relevant instructions call the units in the post-process module to distinguish it from SpMM. For GAT, the weights among the nodes are recalculated in the EFE stage, and then the SpMM and GEMM operations are performed successively, followed by the LeakyReLU operation in the post-process Module. After entering the NFA stage, the post-process Module is called to perform the softmax operation first, and then the GEMM is executed to complete the aggregation. In the final NFC stage, only the post-process Module needs to be called to execute the activation function on the aggregated results. The whole process is executed by the instructions generated in the model file to realize the collaboration between different modules.

V. IMPLEMENTATION AND RESULTS

A. Experiment Setup

We implement Graph-OPU in Verilog HDL on a Xilinx Alveo U50 FPGA, which integrates 8GB of HBM, offering a peak memory bandwidth of 316GB/s. Though Graph-OPU's theoretical maximum frequency is 330MHz, we operate it at 225MHz for optimal memory bandwidth utilization. Hardware resource usage is derived from synthesis and implementation reports in Vivado 2020.2. GNN models are defined using PyTorch with the Pytorch Geometric (PyG) library [33], and features and weights are quantized to 32-bit fixed-point, allowing PEs to utilize the built-in integer DSPs on the FPGA. Previous research demonstrates that INT32 achieves negligible accuracy loss compared to FP32 for GNNs [34].

Dataset. We comprehensively evaluate Graph-OPU using the mainstream GNN models: GCN, Light-GCN, GIN, GAT,

Authorized licensed use limited to: Vrije Universiteit Brussel. Downloaded on November 07,2025 at 12:42:58 UTC from IEEE Xplore. Restrictions apply.

TABLE V
COMPARISON OF THE MODEL SWITCHING TIME WITH OTHER OPEN-SOURCE ACCELERATORS.

Accelerators	Models	Model switching time (min)	Avg (min)
Graph-OPU	GCN	1.54	**2.02**
	Light-GCN	1.21	
	GAT	2.31	
	GIN	2.12	
	GraphSAGE	3.44	
FlowGNN[1]	GCN	517.41	569.15
	GAT	468.28	
	GIN	721.75	
Ref [18][2]	GCN	85.92	85.92
Ref [19][3]	GCN	120.47	120.47

[1] https://github.com/sharc-lab/FlowGNN
[2] https://github.com/GraphSAINT/GNN-ARCH
[3] https://github.com/jasonlin316/GCN-Inference-Acceleration-HLS

and GraphSAGE. All models have two layers. GCN, GAT, and GraphSAGE are evaluated using three typical datasets, CiteSeer, Cora, and PubMed. Light-GCN as a recommendation system is evaluated using Gowalla, LastFM, and Yelp2018. GIN is evaluated using MUTAG, PROTEINS, and PTC [35].

Comparison platform. To evaluate Graph-OPU, we compare it to SOTA end-to-end overlay accelerators (FlowGNN, FP-GNN, and DeepBurningGL) and some representative FPGA-based specific GCN accelerators. We also compare Graph-OPU with the Intel I7-12700KF CPU and Nvidia RTX A5000 GPU platforms. For the GPU, we use a batch size of 1, ensuring fair comparison for real-time streaming graph inference, as batching graphs delays processing. This approach is consistent with other works [11, 21, 36, 37], which commonly employ batch size 1 for comparisons.

B. Comparison with FPGA-based Accelerators

Table IV compares Graph-OPU with SOTA end-to-end overlay accelerators on the same GNN model. The computing resource consumption of different GNN models implemented in prior works vary significantly. The Efficiency metric (Throughput/DSP) is utilized to assess runtime computational resource efficiency [14], representing the proportion of valuable computation conducted by the DSP on average during execution. The Graph-OPU demonstrates better efficiency compared to existing custom designs.

1) Comparison in terms of model switching time: To evaluate the beniefits of Graph-OPU in reducing model switching time, we compare Graph-OPU with the only three available open-source works. These are FlowGNN [11], Ref [18], and Ref [19], as shown in Table V. Note that although Ref [18] and Ref [19] are FPGA-based specific GCN accelerators that cannot switch different GNN models, their FPGA implement processes are almost identical to model switching time. Therefore, we believe their data is still valuable for comparison. Graph-OPU has substantially reduced model switching time, resulting in an average acceleration of 128× compared to other approaches. This improvement highlights the effectiveness of the ISA design and software compiler. FlowGNN consumes a significant amount of time due to its development using HLS, which includes numerous matrix operations. These designs

TABLE VI
COMPARISON OF EXECUTION LATENCY AND ENERGY EFFICIENCY (EE) WITH SOTA END-TO-END OVERLAY ACCELERATOR ON GCN INFERENCE.

	Dataset	FlowGNN	FP-GNN	DeepBurning-GL	Graph-OPU
Latency (μs)	Cora	6.912	4.24	**3.3**	3.44
	CiteSeer	8.332	7	N/A	**5.13**
	PubMed	**53.22**	66.4	101.9	57.78
EE (graph/kJ)	Cora	7.77E6	**1.77E7**	N/A	**1.77E7**
	CiteSeer	6.44E6	1.07E7	N/A	**1.19E7**
	PubMed	**1.01E6**	1.13E6	N/A	1.05E6

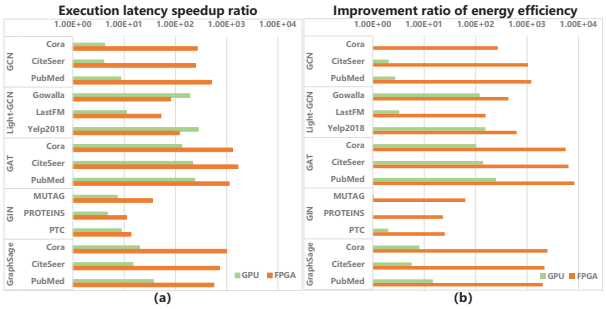

Fig. 4. Comparison with CPU and GPU implementations(Normalized over CPU Intel I7-12700KF). (a) Speedup ratio comparison; (b) Improvement ratio of energy efficiency comparison.

cause the HLS compilation to HDL files and subsequent synthesis to consume a considerable amount of time.

2) Comparison in terms of performance: Table VI lists the execution latency and energy efficiency of Graph-OPU and SOTA end-to-end overlay accelerators on the GCN inference. We compare their performance on the Cora, CiteSeer, and PubMed datasets. We set the same model configuration, using a two-layer GCN, with node embedding dimension being 16 and no edge embedding. While Graph-OPU may not exhibit the highest level of performance on some specific datasets, it arhieves average performance 1.36× faster and 1.41× more energy efficient than these SOTA accelerators. Deep Burning and FP-GNN achieves better performance by utilizing a large number of DSPs without employing specific sparse matrix multiplication optimizations. Flow-GNN is capable of achieving better performance on the lager matrix (PubMed) by utilizing multiple node transformation and multiple message passing through its message passing mechanism. Nevertheless, Graph-OPU exhibits competitive performance and energy efficiency with high flexibility.

C. Comparison with CPU and GPU

As shown in the Fig. 4 (a), we compare the speedup ratio to the CPU (Intel I7-12700KF) and GPU (Nvidia RTX A5000 GPU) implementations. All data is normalized with respect to Intel I7-12700KF. Graph-OPU performs 11-1654× faster than CPU and 0.4-63× faster than GPU for these GNN models with different datasets. In contrast, Graph-OPU performs well in most models but slightly less in Light-GCN models. Since it contains the biggest matrix in the dataset and the concat operation requires taking a series of accumulation for huge-size matrixes. We compare the improvement ratio of energy efficiency of Graph-OPU on different platforms in Fig. 4 (b). The plotted performance is also normalized with respect

Authorized licensed use limited to: Vrije Universiteit Brussel. Downloaded on November 07,2025 at 12:42:58 UTC from IEEE Xplore. Restrictions apply.

to CPU. Graph-OPU shows the average 2013× (23-5305×) and 109× (3-422×) better energy efficiency compared with CPU and GPU, respectively. The result demonstrates the architectural advantages of Graph-OPU.

VI. Conclusion

We present Graph-OPU, a highly integrated FPGA-based overlay processor for GNN. Graph-OPU allows users to execute and fast switch between different mainstream GNN modules. Specifically, the memory access module and data loading modules provide excellent data parallelism by efficiently using HBM's high bandwidth and multiple channels. The computational engine module performs SpMM and GEMM in GNN as a unified matrix multiplication with fully pipelined parallelization, while the post-process module completes other operations. Graph-OPU has both scalability and the potential to be compatible with the newly emerging GNN models. On average, Graph-OPU achieves a 128× speedup in model switching time compared to other works. Graph-OPU outperforms the SOTA overlay accelerators for GNN by 1.36× speedup and 1.41× energy efficiency improvement on average. Compared with CPU and GPU, Graph-OPU achieves up to 1654× and 63× speedup, as well as up to 5305× and 422× better energy efficiency, respectively.

Acknowledgment

This work was financially supported in part by National Key Research and Development Program of China under Grant 2021YFA1003602, in part by Shanghai Pujiang Program under Grant 22PJD003, and in part by the Natural Science Foundation for Distinguished Young Scholars of Jiangsu Province, China under Grant BK20200038.

References

[1] Atz, K. *et al.*, "Geometric Deep Learning on Molecular Representations," *Nat Mach Intell*, 2021.
[2] Zhang, C. *et al.*, "H-GCN: A Graph Convolutional Network Accelerator on Versal ACAP Architecture," in *FPL*, 2022.
[3] Deo, N. *et al.*, "Multimodal Trajectory Prediction Conditioned on Lane-Graph Traversals," in *PMLR*, 2022.
[4] Wang, Y. *et al.*, "Multi-View Graph Contrastive Representation Learning for Drug-Drug Interaction Prediction," in *WWW*, 2021.
[5] Geng, T. *et al.*, "AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing," in *MICRO*, 2020.
[6] Tao, Z. *et al.*, "LW-GCN: A Lightweight FPGA-based Graph Convolutional Network Accelerator," *ACM Trans. Reconfigurable Technol. Syst.*, 2022.
[7] Zhang, B. *et al.*, "BoostGCN: A Framework for Optimizing GCN Inference on FPGA," in *FCCM*, 2021.
[8] Geng, T. *et al.*, "I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement through Islandization," in *MICRO*, 2021.
[9] Liang, S. *et al.*, "DeepBurning-GL: an Automated Framework for Generating Graph Neural Network Accelerators," in *ICCAD*, 2020.
[10] Tian, T. *et al.*, "FP-GNN: Adaptive FPGA Accelerator for Graph Neural Networks," *Future Gener Comp Syst*, 2022.
[11] Sarkar, R. *et al.*, "FlowGNN: A Dataflow Architecture for Universal Graph Neural Network Inference via Multi-Queue Streaming," in *HPCA*, 2023.
[12] Seyoum, B. *et al.*, "Automating the design flow under dynamic partial reconfiguration for hardware-software co-design in FPGA SoC," in *SAC*, 2021.
[13] Bruna, J. *et al.*, "Spectral Networks and Locally Connected Networks on Graphs," *arXiv*, 2014.
[14] He, X. *et al.*, "LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation," in *ACM SIGIR*, 2020.
[15] Veličković, P. *et al.*, "Graph Attention Networks," *arXiv*, 2018.
[16] Xu, K. *et al.*, "How Powerful are Graph Neural Networks?" in *ICLR*, 2019.
[17] Hamilton, W. *et al.*, "Inductive Representation Learning on Large Graphs," in *NeurIPS*, 2017.
[18] Zhang, B. *et al.*, "Hardware Acceleration of Large Scale GCN Inference," in *ASAP*, 2020.
[19] Lin, Y. C. *et al.*, "GCN inference acceleration using high-level synthesis," in *HPEC*, 2021.
[20] Guo, K. *et al.*, "[DL] a survey of FPGA-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, 2019.
[21] Yu, Y. *et al.*, "OPU: An FPGA-Based Overlay Processor for Convolutional Neural Networks," *IEEE Trans. VLSI Syst*, 2020.
[22] Chen, T. *et al.*, "TVM: An automated End-to-End optimizing compiler for deep learning," in *OSDI*, 2018.
[23] Lattner, C. *et al.*, "LLVM: a Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, 2004.
[24] Yan, M. *et al.*, "HyGCN: A GCN Accelerator with Hybrid Architecture," in *HPCA*, 2020.
[25] Kiningham, K. *et al.*, "GRIP: A graph neural network accelerator architecture," *IEEE Trans. Comput*, 2022.
[26] Paszke, A. *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *NeurIPS*, 2019.
[27] Chen, X. *et al.*, "ReGraph: Scaling graph processing on HBM-enabled FPGAs with heterogeneous pipelines," in *MICRO*, 2022.
[28] Wu, C. *et al.*, "SkeletonGCN: A simple yet effective accelerator for GCN training," in *FPL*, 2022.
[29] Gao, Y. *et al.*, "SDMA: An efficient and flexible sparse-dense matrix-multiplication architecture for GNNs," in *FPL*, 2022.
[30] Song, L. *et al.*, "Sextans: A streaming accelerator for general-purpose sparse-matrix dense-matrix multiplication," in *FPGA*, 2022.
[31] Wang, H. *et al.*, "Spatten: Efficient sparse attention architecture with cascade token and head pruning," in *HPCA*, 2021.
[32] Xu, Z. *et al.*, "CNN-based Feature-point Extraction for Real-time Visual SLAM on Embedded FPGA," in *FCCM*, 2020.
[33] Fey, M. *et al.*, "Fast graph representation learning with PyTorch geometric," *arXiv*, 2019.
[34] Yuan, W. *et al.*, "QEGCN: An FPGA-based accelerator for quantized GCNs with edge-level parallelism," *J. Syst. Archit*, 2022.
[35] Wei, S. *et al.*, "Graph learning and its applications: A holistic survey," *arXiv*, 2023.
[36] GroqInc, "The challenge of batch size 1: Groq adds responsiveness to inference performance," 2020.
[37] Qu, H. *et al.*, "Jet tagging via particle clouds," *Phys. Rev. D*, 2020.

Authorized licensed use limited to: Vrije Universiteit Brussel. Downloaded on November 07,2025 at 12:42:58 UTC from IEEE Xplore. Restrictions apply.