GN-Drive: Accelerating GNN Inference for Graph Prediction over Numerous Small Graphs

Min-Gwon Song*, Kwangrae Kim*, Soomin Rho*, Bohyeon Kang†, and Ki-Seok Chung*
*Department of Electronic Engineering, Hanyang University, Seoul, South Korea
†Department of Nanoscale Semiconductor Engineering, Hanyang University, Seoul, South Korea
Email: {mgsong1005, kksilver91, smrho, kbh2247, kchung}@hanyang.ac.kr

***Abstract*—Graph neural networks (GNNs) have achieved state-of-the-art performance on both node-level and graph-level prediction tasks. However, most existing GNN accelerators are optimized for node prediction workloads, where inference is performed on a single large-scale graph, and become inefficient for graph prediction workloads involving numerous small-scale graphs. To accelerate graph prediction workloads, GNN accelerators must exploit graph-level parallelism and should not suffer from a memory bottleneck in the encoding layer. Further, it is important to design hardware that is efficient to deal with diverse types of GNNs. To address these challenges, we propose GN-Drive, a graph prediction GNN accelerator that (i) designs a lightweight Graph Engine based on a hardware-friendly GNN abstraction for efficiently supporting diverse GNN models with high hardware utilization, (ii) instantiates multiple such Graph Engines to achieve high performance by exploiting graph-level parallelism, and (iii) adopts a cache for node and edge encodings called Encode Cache to reduce memory access overhead. Based on cycle-accurate simulation, the proposed design achieves average speedups of 1.64×, 2.61×, and 1.49× over an RTX 4090, HyGCN, and FlowGNN, respectively.**

***Index Terms*—Graph Neural Network, Hardware Accelerator, Graph Prediction, Graph Convolution Network**

I. Introduction

Graph neural networks (GNNs) have attracted significant attention due to their ability to process non-Euclidean graph data. Among various GNN tasks, node prediction and graph prediction are two representative task categories with distinct workload characteristics. While node prediction predicts labels or properties of individual nodes in a single large-scale graph, graph prediction predicts labels or properties of entire graphs across numerous small-scale graphs. The core operations of GNNs, aggregation and combination, involve irregular memory access patterns and sparse-matrix computations, which make them difficult to accelerate efficiently on CPUs or GPUs [1]. To tackle these challenges, previous GNN accelerators have focused on aggregation–combination dataflow, which can be formulated as sparse matrix–matrix multiplication (SpMM) and general matrix multiplication (GEMM) [1]–[3]. Although these efforts have been successful in accelerating GNNs, most existing GNN accelerators have focused on node prediction tasks, which target a single large-scale graph. Meanwhile, graph prediction tasks require efficient processing of numerous small graphs, such as real-time graph processing in Level-1 triggers for high-energy physics experiments [4] and high-throughput graph processing for molecular virtual screening [5]. This requirement necessitates new GNN accelerators that efficiently accelerate graph prediction tasks where numerous small-scale graphs are involved.

Hardware accelerators for graph prediction on numerous small graphs should satisfy the following two requirements: scale-out execution and avoidance of a memory bottleneck.

First, scale-out execution is required. Prior GNN accelerators for node prediction typically adopt a scale-up approach, allocating substantial computing resources to process a single large graph. However, this scale-up approach underutilizes hardware on small graphs, limiting performance gains from additional resources. This limitation can be addressed by assigning each small graph to a lightweight Graph Engine (GE) with fewer computing resources, thereby improving hardware utilization. Moreover, the workload comprises numerous graphs, so graph-level parallelism is essential for improving performance. Hence, we design a graph prediction accelerator with a scale-out approach that improves hardware utilization and performance by processing multiple graphs in parallel with multiple lightweight GEs.

Second, mitigating the memory bottleneck issue in the encoding layer is required. In molecule graph prediction, the encoding layer for node and edge features often suffers from a memory bottleneck issue, as each categorical attribute of a node or an edge is retrieved via a separate embedding table, and the retrieved embedding vectors are summed to produce the encoded nodes and edges. Profiling results show that the encoding layer accounts for up to 30% of GPU end-to-end latency. To alleviate this issue, we propose a novel cache called Encode Cache (EC), which caches the encoded outputs to reduce redundant memory accesses.

Beyond the challenges inherent to graph prediction, it is also crucial to support diverse GNN models. Recently, new GNNs with distinctive features have emerged as promising models. These emerging GNNs are often implemented using a method called Scatter-ApplyEdge-Gather-ApplyVertex (SAGA) abstraction [6], [7]. In SAGA, while the Scatter and Gather stages remain consistent across models, the ApplyEdge and ApplyVertex stages vary across models. However, our analysis reveals that the ApplyEdge and ApplyVertex stages are composed of GEMM and single instruction multiple data (SIMD) operations, allowing SAGA-based hardware to support diverse GNN models by instantiating both GEMM and SIMD units in each stage. However, depending on the GNN

model, Apply- stages may require only GEMM operations, only SIMD operations, both, or, in some cases, neither. As a result, SAGA-based hardware includes both GEMM and SIMD units in each stage, regardless of whether they are actually used or not. This inefficiency leaves many units idle, leading to resource under-utilization. To address this challenge, we propose a hardware-friendly GNN abstraction, HeavyApply–Scatter–LiteApply–Gather (HSLG), which improves hardware utilization while supporting diverse GNN models on a single hardware accelerator.

The main contributions of this work are summarized as follows. GN-Drive is an HSLG-based graph prediction GNN accelerator that (i) implements a lightweight GE by pipelining dedicated stage modules for each HSLG stage to efficiently support diverse GNN models with high hardware utilization, (ii) instantiates multiple such GEs to achieve high performance by exploiting graph-level parallelism, and (iii) integrates EC, which caches encoded results to alleviate the memory bottleneck of the encoding layer. Based on cycle-accurate simulation, the GN-Drive achieves average speedups of 1.64×, 2.61×, and 1.49× over an RTX 4090, HyGCN, and FlowGNN.

II. Background and Motivation

A. Graph Prediction Workload Characteristics

TABLE I: GNN dataset characteristics for node prediction and graph prediction.

Dataset	# Graphs	# Nodes	# Edges	Task
CiteSeer	1	3,327	4,732	node prediction
PubMed	1	19,717	44,338	node prediction
Reddit	1	232,965	114,615,892	node prediction
MolHIV	4,113	25.3	55.6	graph prediction
MolPCBA	43,773	27.0	59.3	graph prediction
HEP	10,000	49.1	785.3	graph prediction

Node prediction is a task to predict the properties of each node, and graph prediction is a task to predict the properties of a graph. Graph prediction tasks are often useful to solve problems in domains such as high-energy physics (HEP) and virtual screening of molecules, and they typically operate on numerous small-scale graphs. Table I summarizes the structural differences between node and graph prediction datasets. The datasets used for node prediction consist of a single large-scale graph with high sparsity, leading to memory-bound behavior during the aggregation. In contrast, the datasets used for graph prediction have small-scale graphs, so this issue is less pronounced. However, graph prediction introduces two challenges that have not been considered by prior node prediction GNN accelerators.

Need for Scale-Out Approach. Existing node prediction GNN accelerators typically adopt a scale-up approach, allocating substantial hardware resources to maximize node-level parallelism within a single large graph, as illustrated in Fig. 1(a). However, when these architectures are applied to graph prediction tasks consisting of numerous small graphs, the abundant computing resources are often underutilized.

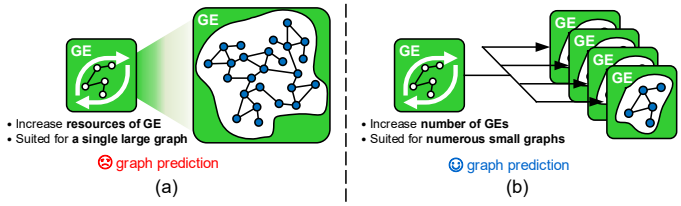


Fig. 1: Comparison of (a) scale-up approach for a single large graph and (b) scale-out approach for numerous small graphs.

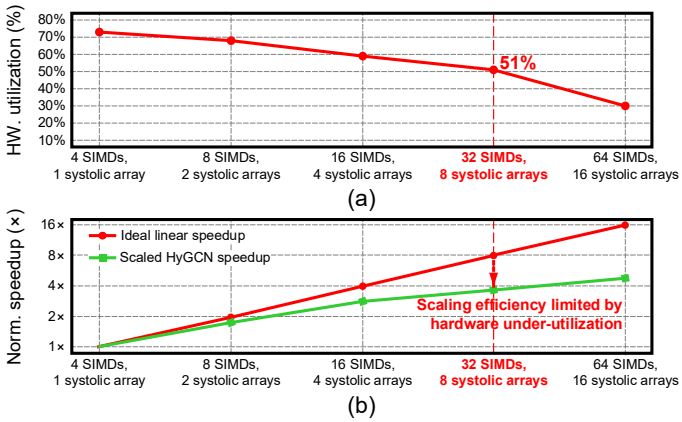
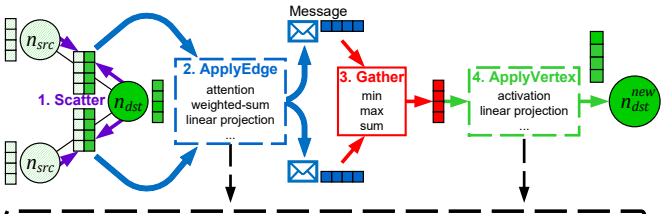


Fig. 2: Effect of computing resource scaling in HyGCN on the graph prediction workload. (a) Hardware utilization and (b) normalized speedup as computing resources increase. Each SIMD contains 16 cores, and the systolic array size is 4 × 128.

To illustrate this limitation, Fig. 2 evaluates HyGCN [1], a representative node prediction GNN accelerator, on the MolHIV [8] graph prediction workload using GCN [9] by scaling computing resources. Note that HyGCN's node prediction design point consists of 32 SIMD units and 8 systolic arrays. As shown in Fig. 2(a), hardware utilization decreases as the number of SIMD units and systolic arrays increases. This is mainly because allocating excessive hardware resources increases the penalty due to intra-tile underutilization, which is particularly severe in small graphs. At HyGCN's design point for node prediction, the hardware utilization drops to 51%. As a result, this under-utilization limits the performance gain obtained by excessive computing resources, as shown in Fig. 2(b). To address this mismatch, as shown in Fig. 1(b), we adopt a scale-out approach that allocates fewer computing resources to each small graph for higher hardware utilization and instantiates multiple lightweight GEs in parallel for higher performance through graph-level parallelism.

Memory Bottleneck of the Encoding Layer. Although executing multiple GEs in parallel will improve performance, as more graphs are processed in parallel, the required bandwidth for reading graph data increases accordingly. Moreover, it becomes more critical for molecular datasets, where encoding a single node or edge feature requires multiple lookups from memory. Therefore, avoiding a memory bottleneck while reading a large amount of data from DRAM is essential to fully exploit graph-level parallelism.

Encoding of nodes and edges should be done first before



Inconsistent behavior in ApplyEdge and ApplyVertex.

Stage	Function	GNN models			
		GCN	GIN	GAT	PNA
ApplyEdge	Operation	linear projection, sqrt, scale, ReLU	ReLU	attention	linear projection, power
	Hardware mapping	GEMM, SIMD	SIMD	GEMM, SIMD	GEMM, SIMD
ApplyVertex	Operation	-	MLP	ReLU	linear projection, log, scale, ReLU, sqrt, div
	Hardware mapping	-	GEMM, SIMD	SIMD	GEMM, SIMD

Fig. 3: SAGA abstraction for implementing the Message-Passing operation in GNNs, along with a comparison of the operation and the hardware mapping used in ApplyEdge and ApplyVertex across diverse GNN models.

the graph convolution (GConv) layer. Each input attribute (e.g., atomic number or chirality for nodes, and bond type for edges) has its own embedding table, and the embedding vectors retrieved by the attribute indices are summed to form the encoded node, edge results as follows:

$$y = \sum_{i=0}^{d-1} E_i[na[i]];\ y = \sum_{i=0}^{d-1} E_i^l[ea[i]] \tag{1}$$

where na is the node attribute feature and ea is the edge attribute feature with dimension d. E_i^l is an embedding table responsible for the i-th attribute at layer l. $E_i[na[i]]$ and $E_i^l[ea[i]]$ are retrieved embedding vectors. y is the encoded output. The encoding layer with redundant retrieval may lead to a memory bottleneck. Profiling results on GCN-MolHIV and GIN-MolHIV reveal that the node and edge encoding layers account for 27.85% and 30.34% of the end-to-end latency on a GPU, respectively. To avoid the memory bottleneck, we adopt EC, which leverages the highly biased distributions of $na[i]$ and $ea[i]$ values observed in our profiling results.

B. GNN Abstraction for Graph Engine Design

A GConv layer is the core computation of GNNs, where node features are updated, and GNN abstractions provide a representation of this layer across diverse GNN models.

Prior GNN abstractions. Most existing GNN accelerators have primarily focused on optimizing GEMM and SpMM computations. However, this approach cannot support emerging GNNs, which involve additional operations such as edge embedding and anisotropic aggregation [10]. To support the diverse emerging GNNs, a Message-Passing abstraction can be adopted. The Message-Passing equation is shown below.

$$n_v^{(l)} = f(n_v^{(l-1)},\ \psi(\{\phi(n_v^{(l-1)}, n_u^{(l-1)}, e_{vu}) \mid u \in N(v)\})) \tag{2}$$

where $n_v^{(l)}$ and $n_u^{(l)}$ denote the node features of the destination node v and the source node u at the l^{th} layer, and e_{vu} denotes the edge feature between v and u. The function $\phi(\cdot)$ defines the message function. $\psi(\cdot)$ is an aggregation function. $N(v)$ is the neighbor node set of v. $f(\cdot)$ is the update function.

The Message-Passing abstraction can be implemented based on the SAGA abstraction as shown in Fig. 3. Scatter routes the node feature to its adjacent edges using copy and concatenation operations. ApplyEdge uses the scattered node to generate a message. Gather aggregates all incoming messages via the aggregator. Finally, ApplyVertex updates the aggregated message to produce the new node features.

Limitation of Existing GNN Accelerators. Although existing hardware accelerators have implemented these GNN abstractions in hardware, they have limitations in supporting diverse GNN models. FlowGNN [10] targets graph prediction and constructs its dataflow directly from the Message-Passing abstraction. This allows FlowGNN to support diverse GNN models by implementing model-specific message and aggregation functions. However, since each GNN has different message functions ($\phi(\cdot)$) or aggregation functions ($\psi(\cdot)$), FlowGNN requires FPGA reconfiguration to switch models, as each model's functions must be compiled into dedicated hardware kernels. QEGCN [11] implements the SAGA abstraction in a hardware accelerator. However, QEGCN hardwires the ApplyEdge and ApplyVertex functions for GCN, which limits the flexibility to support other GNN models.

Need for Hardware-Friendly GNN Abstraction. The main concern in processing GNN abstractions is the fact that Message-Passing's functions (message, update, aggregation) are model-specific, and SAGA's ApplyEdge and ApplyVertex also inherit model-specific behavior, making it difficult for fixed hardware logic to support diverse models. However, our analysis shows that although the ApplyEdge and ApplyVertex stages are model-dependent, their computations can be decomposed into GEMM and SIMD operations, as shown in Fig. 3. This observation suggests an opportunity for diverse GNN models to be supported by a unified GE architecture. Hence, we adopt the SAGA abstraction as our baseline architecture for supporting diverse models on a GE.

Hardware implementation of SAGA requires the GEMM and SIMD units in both the ApplyEdge and the ApplyVertex stages. However, some models may involve only GEMM operations, only SIMD operations, or neither. Even in cases where both operations are involved, the amount of computation between ApplyEdge and ApplyVertex can be heavily imbalanced. This imbalance leads to significant hardware underutilization. Hence, we propose HSLG, a hardware-friendly GNN abstraction that improves hardware utilization by reorganizing SAGA and separating GEMM and SIMD operations into distinct stages, while still preserving model coverage. In determining the order of these stages, HSLG places GEMM before Scatter. Since Scatter copies and multicasts the node or edge features, this ordering eliminates redundant GEMM executions on duplicated features [7].

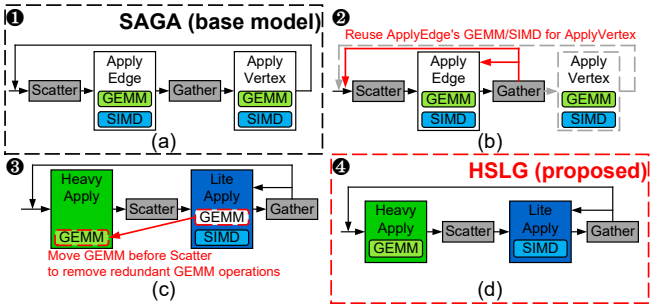


Fig. 4: Transforming SAGA into HSLG: (a) SAGA abstraction. (b) ApplyEdge and ApplyVertex are merged into a shared Apply-stage module. (c) GEMM is moved before Scatter to eliminate redundant GEMM operations. (d) HSLG abstraction.

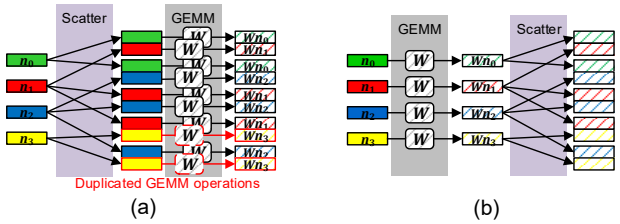


Fig. 5: Comparison of GEMM placement for redundant GEMM elimination: (a) After Scatter. (b) Before Scatter.

III. HSLG: Hardware-Friendly GNN Abstraction

Fig. 4 illustrates the transformation from SAGA to HSLG, which reorders SAGA's operations to decouple GEMM and SIMD into distinct stages, enabling computing resource reuse while eliminating redundant GEMM computations.

Fig. 4(a) shows the baseline SAGA architecture.

As shown in Fig. 4(b), ApplyEdge and ApplyVertex share the same GEMM/SIMD-based compute structure. Accordingly, the GEMM/SIMD-based compute structures of ApplyEdge and ApplyVertex can be merged into a shared Apply-stage module, avoiding separate hardware instantiation.

Fig. 4(c) further exploits a property of the Scatter stage to eliminate duplicated GEMM operations. As illustrated in Fig. 5(a), when GEMM is performed after Scatter, the same computation is repeatedly applied to duplicated feature values, leading to redundant GEMM operations. Since Scatter consists of copy and concatenation operations, it preserves the node and edge feature values without modification. Therefore, GEMM can be moved before Scatter while preserving computational equivalence, thereby eliminating redundant computations, as shown in Fig. 5(b).

For a linear projection over concatenated destination-node, source-node, and edge features, let S_{dst}, S_{src}, and S_{edge} denote the Scatter operators that route the corresponding features to each edge. Then, the projection can be decomposed as

$$[S_{\text{dst}}N \| S_{\text{src}}N \| S_{\text{edge}}E]W = S_{\text{dst}}(NW_{\text{dst}}) + S_{\text{src}}(NW_{\text{src}}) + S_{\text{edge}}(EW_{\text{edge}}) \tag{3}$$

where W_{dst}, W_{src}, and W_{edge} are the corresponding partitions of W. Thus, the destination-node, source-node, and edge projections can be computed independently in `HeavyApply` before Scatter, while Scatter only routes the resulting message pieces according to graph connectivity.

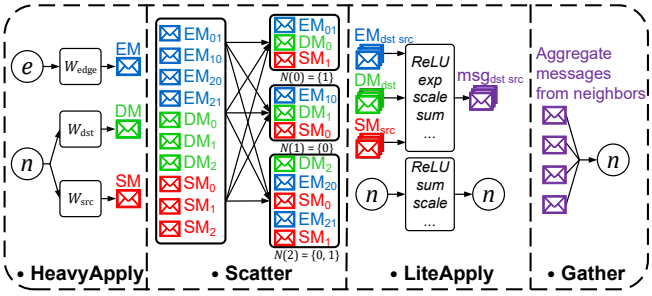


Fig. 6: Operation of each HSLG stage.

When a computation sequence requires an additional GEMM operation after the first GEMM within a layer, the intermediate result is written back to the Input Buffer and re-enters `HeavyApply` for an additional HSLG pass. This feedback mechanism supports both consecutive GEMM operations and GEMM–SIMD–GEMM sequences by reusing the same GEMM hardware without introducing an additional GEMM unit. When no additional edge-wise routing is required in the additional pass, Scatter can be bypassed. The execution penalty caused by partial HSLG pipeline disruption can be mitigated through double buffering.

Together, these transformations improve hardware efficiency through computing resource reuse while eliminating redundant GEMM operations, leading to the proposed HSLG architecture in Fig. 4(d), where the GEMM-only and SIMD-only stages are defined as `HeavyApply` and `LiteApply`, respectively.

Under the PNA [12] model on the MolHIV dataset, the HSLG-based GE improves hardware utilization by 28% over the SAGA-based GE through computing resource reuse and achieves a 1.96× speedup by reducing redundant GEMM operations. A detailed HSLG-based GE architecture will be described in the next section.

Fig. 6 shows the computation performed in HSLG.

HeavyApply: HSLG first executes a GEMM operation to linearly project the node and edge features and obtains Message Pieces (MPs). MPs are categorized as Destination Message (DM), Source Message (SM), and Edge Message (EM). DM, SM, and EM are obtained by projecting the destination-node, source-node, and edge features with their corresponding weights, respectively. MPs are passed to Scatter as inputs. `HeavyApply` is mapped to a GEMM unit.

Scatter: In `Scatter`, MPs are routed to each destination node by grouping DM, SM, and EM along its connected edges.

LiteApply: `LiteApply` performs element-wise operations over the scattered MPs to form messages, generating one message for each edge connected to the destination node. `LiteApply` is mapped to a SIMD unit.

Gather: In `Gather`, messages are aggregated using a permutation invariant aggregator, such as min, max, or sum.

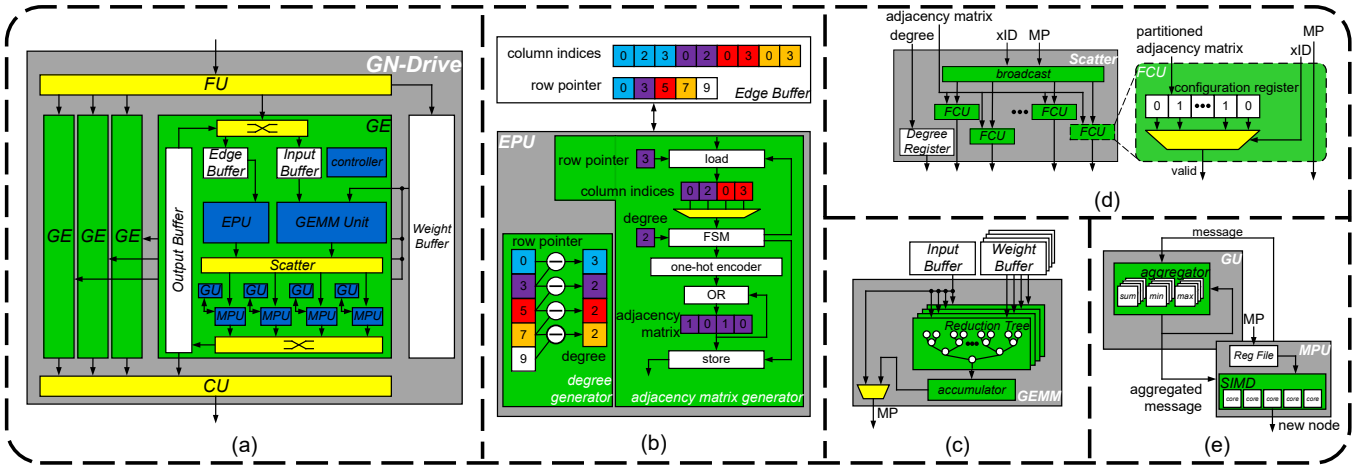


Fig. 7: GN-Drive architecture: (a) The top-level block diagram. (b) Edge Processing Unit (EPU). (c) The GEMM unit. (d) The Scatter unit. (e) Message Processing Unit (MPU) with Gather Unit (GU).

IV. HSLG-Based Hardware Accelerator

A. Architecture Overview

Fig. 7(a) presents the architecture of the proposed accelerator, GN-Drive. GN-Drive includes 6 GEs, each dedicated to a single graph and designed based on the HSLG abstraction. Each GE contains an Edge Processing Unit (EPU), a GEMM unit, a Scatter unit, Message Processing Units (MPUs), and Gather Units (GUs). Input Buffer and Output Buffer are double-buffered to hide the latency of data transfers to and from on-chip and off-chip memory. Fetch Unit (FU) reads edge data from DRAM in the compressed sparse row (CSR) format. We adopt CSR because degrees can be obtained by subtracting consecutive entries in the row pointer vector, and the format provides efficient compression for sparse matrices. The EPU converts a CSR representation into an adjacency matrix and writes it into the Scatter's configuration registers. Subsequently, FU fetches node and edge features from the Output Buffer or DRAM into the Input Buffer. The GEMM unit processes nodes and edges in the Input Buffer to generate MPs, which are then delivered to the Scatter unit. The Scatter unit then routes the MPs to the target MPUs according to the configuration register. Each MPU performs the LiteApply stage and generates messages for the corresponding destination nodes; the GU then immediately aggregates them. The aggregated message is returned to the MPU for the update step, and the final results are written to the Output Buffer. Results in the Output Buffer are written back to the DRAM through the Commit unit. All GEs share weights via the Weight Buffer and proceed in a synchronized manner, with progress aligned to the GE with the longest processing delay.

Multiple MPUs and GUs in a GE exploit node-level parallelism, and stages are executed in a pipelined manner. Also, we adopt GEs with fewer resources to improve hardware utilization, and instantiate them in parallel to exploit graph-level parallelism and thereby improve performance. The node features updated by GConv are stored in the Output Buffer, and because each GE is equipped with a dedicated output buffer, the stored features can be read in parallel across multiple GEs.

B. Microarchitecture

EPU: The EPU is shown in Fig. 7(b). The EPU receives graph data in the CSR format from the Edge Buffer. The `degree generator` computes the degree by subtracting consecutive entries in the row pointer. The `adjacency matrix generator` uses the row pointer as an offset to load column indices from the Edge Buffer. FSM then consumes column indices equal to the input degree, the one-hot encoder encodes them sequentially, and the `OR` unit accumulates them to build an adjacency matrix.

GEMM unit: The GEMM unit is shown in Fig. 7(c). The GEMM unit performs the HeavyApply stage. The unit consists of multiple Reduction Trees (RTs), and each RT uses a distinct Weight Buffer bank. Nodes and edges are broadcast from the Input Buffer so that all RTs share the same inputs. As a result, inputs can be reused across weights, and MPs produced by the RTs can be streamed to the Scatter unit. The RT can be used to perform a pooling operation for the virtual node, which connects to all nodes in the graph.

Scatter unit: As shown in Fig. 7(d), Scatter receives the degree value and the adjacency matrix from the EPU and the MP stream from the GEMM unit. The degree register can be shared across multiple MPUs. The adjacency matrix is partitioned into the configuration registers of the Flow Control Unit (FCU). Based on the xID (e.g., edge ID or node ID) accompanying each MP, the FCU asserts valid signals so that each MP is routed to its destination MPU.

MPU/GU: The structures of MPU and GU are shown in Fig. 7(e). The MPU performs the LiteApply stage using SIMD. Messages generated by the MPU are forwarded directly to the GU. The GU performs the Gather stage. Its aggregator supports sum, min, and max. Intermediate results are stored in a register file, enabling output-stationary behavior. When the aggregated message is returned to the MPU, the MPU executes the update step and produces the updated node.

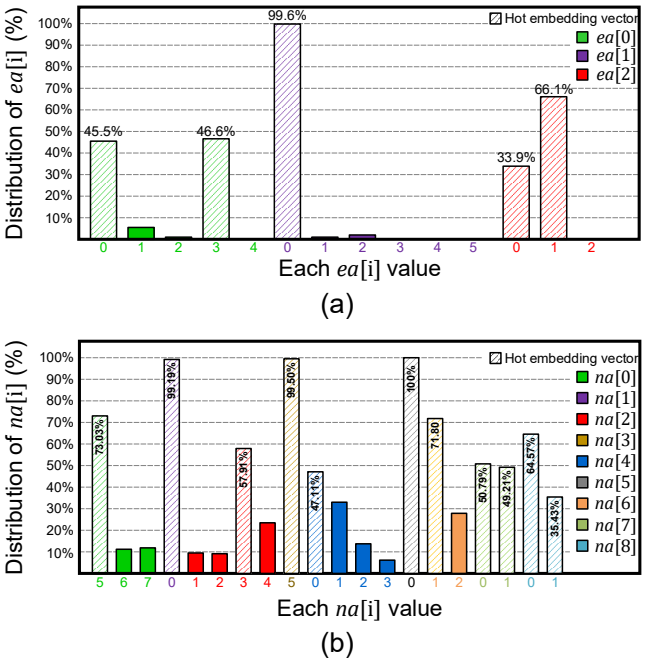


Fig. 8: Distribution of each (a) $ea[i]$, (b) $na[i]$ value.

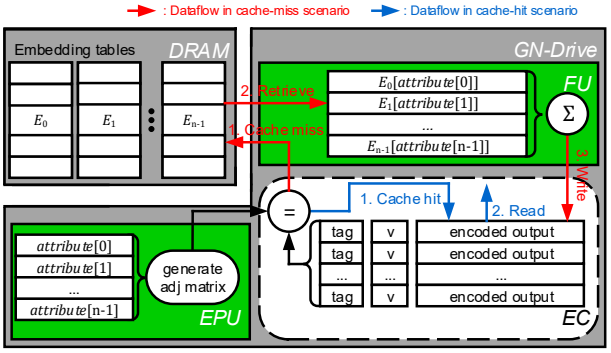


Fig. 9: Dataflow in Encode Cache (EC).

V. Encode Cache

The encoding operations for nodes and edges are defined in Eq. 1. For an encoded output y, embedding vectors ($E_i[na[i]]$ or $E_i^l[ea[i]]$) should be retrieved from distinct embedding tables (E_i) by an integer attribute value ($ea[i]$ or $na[i]$). This operation involves redundant retrievals, leading to a memory bottleneck. Furthermore, as described in the previous section, GN-Drive exploits graph-level parallelism by utilizing multiple GEs. Although this design can deliver high performance, increasing the number of GEs correspondingly increases the demand for data from DRAM. This issue is critical in molecular datasets due to multiple high-dimensional embedding vector lookups.

To resolve this concern, we analyzed the distribution of the attribute values. Fig. 8(a) shows the distribution of each $ea[i]$ value of MolHIV. The results show that, among the embedding vectors retrieved from the embedding tables, only a small subset is heavily used in practice. For example, even though

TABLE II: GN-Drive Configuration

GN-Drive Configuration	
Compute Unit	1 GHz @ 6 GEs (each with 16×5-depth RTs, 8×16-lane SIMDs)
On-chip Memory	Weight Buffer (1 MB), Input Buffer (6 KB), Output Buffer (200 KB), Edge Buffer (4 KB), Encode Cache (0.5 KB)
Off-chip Memory	HBM2 (256 GB/s, 8 GB)

$ea[1]$ can retrieve indices from 0 to 5 in the embedding table E_1, the access pattern is highly concentrated at index 0 of the embedding table, which accounts for 99.6%. Similarly, the distributions of other individual attributes for the edge are peaked at a few values, implying the existence of frequently referenced hot embedding vectors. Considering only these hot embedding vectors per attribute, the number of distinct encoded vectors formed by their combinations is also small. For instance, in the edge encoding, there are 2, 1, and 2 hot embedding vectors across three attributes, yielding only 4 encoded vectors. The node attribute values exhibit a similar trend as shown in Fig. 8(b). Motivated by this observation, we resolve the memory bottleneck issue of the encoding layer by caching the encoded output (y). Note that, for the node encoding, EC does not cache the encoded output for the full attribute combination. Instead, it partitions the node attributes into several groups, caches the partial encoded output for each group-wise combination, and accumulates the cached partial outputs to generate the final node encoded vector.

Fig. 9 shows how the EC operates. The EPU first forms an adjacency matrix defined over the input attribute vector. Note that the adjacency matrix represents attribute combinations rather than graph connectivity. This adjacency matrix serves as a tag that uniquely identifies the target encoded vector and is compared against the tag stored in the EC. On a cache miss, the FU retrieves the required embedding vectors from DRAM. Then, it aggregates them to produce the encoded result and updates the EC with the tag and the resulting output. On a cache hit, the EC directly returns the previously stored encoded vector. The EC is shared among all GEs. Since the EC tag does not track layer or weight versions, the EC is flushed whenever the embedding table changes. Although caching embedding vectors can also achieve a high hit ratio, this method still requires multiple embedding vector accesses from the cache and their accumulation to generate the encoded output, even on a cache hit. In contrast, EC directly caches the encoded output, requiring only a single cache access on a hit.

VI. Evaluation

A. Experiment Environment

Cycle-Accurate Simulation and RTL Evaluation: We evaluated GN-Drive using cycle-accurate simulation and Verilog RTL designs synthesized by Synopsys Design Compiler in SAMSUNG 28 nm FD-SOI technology operating at 1 GHz. As shown in Table II, the design integrates 6 GEs, each with 16 5-depth RTs, and 8 16-lane MPUs. We use the same 16-bit fixed-point precision as FlowGNN [10]. The synthesized

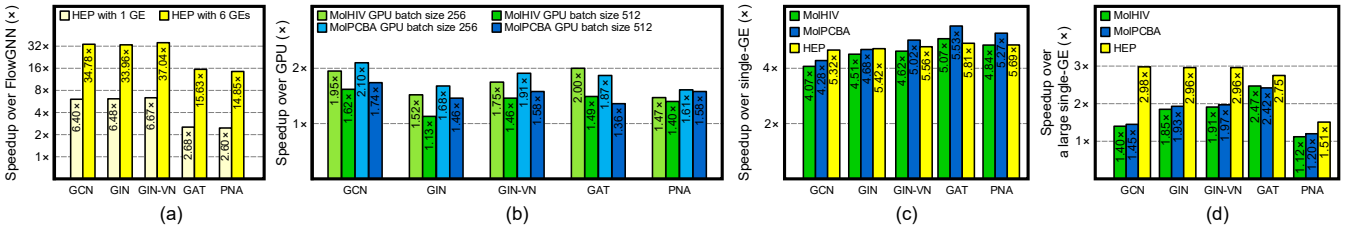


Fig. 10: Speedup of GN-Drive across different GNN models on the MolHIV, MolPCBA, and HEP datasets. The x-axis denotes the models, and the y-axis shows the speedup ratio. (a) Small-batch scenario comparing GN-Drive with the FlowGNN. (b) Large-batch scenario comparing GN-Drive with the GPU. (c) Speedup of GN-Drive with multiple GEs over the single GE configuration. (d) Speedup of GN-Drive with multiple lightweight GEs over a single large GE configuration.

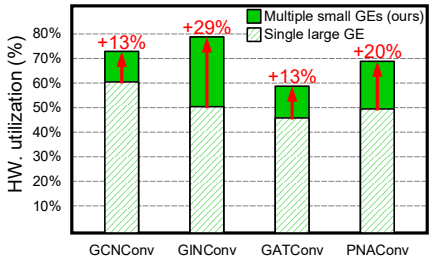


Fig. 11: Comparison of average hardware utilization between a single large GE and multiple small GEs across the GConv layers of different GNN models on MolHIV, MolPCBA, and HEP.

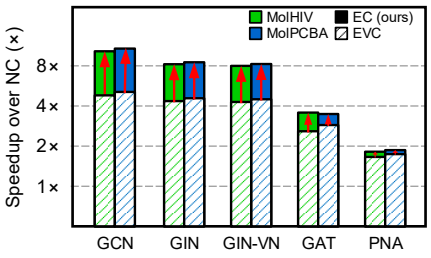


Fig. 12: Speedup of GN-Drive with different caching schemes across GNN models on MolHIV and MolPCBA. Speedup is normalized to the baseline implementation with NC.

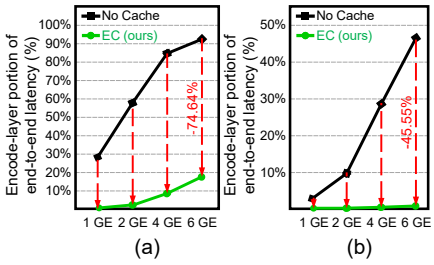


Fig. 13: Effect of EC on the encoding layer latency ratio as the number of GEs increases on MolHIV. (a) GCN, which shows the largest EC impact. (b) PNA, which shows the smallest EC impact.

design occupies 8.18 mm^2 and consumes 5.92 W of power. We model off-chip memory using an HBM2 configuration in DRAMsim3 [13] with 8 GB capacity and 256 GB/s bandwidth.

Models & Datasets: We used four representative GNNs: GCN [9], GIN [14], GAT [15], and PNA [12]. We additionally included GIN with virtual nodes, denoted as GIN-VN. For GCN, GIN, and GIN-VN, we use five layers with a node embedding dimension of 100; GAT uses five layers with four attention heads and 16 features per head; and PNA uses four layers with a node embedding dimension of 80. Experiments were conducted on the OGB molecular benchmarks [8], including MolHIV and MolPCBA, as well as the HEP dataset [16], whose graphs were constructed using EdgeConv with 16 nearest neighbors per node.

Baseline: As the software baseline, we used models implemented with the PyTorch Geometric (PyG) [17] framework and executed them on an NVIDIA RTX 4090 GPU. For the GPU baseline, we measured CUDA kernel execution time using Nsight Systems. Also, we compared GN-Drive with FlowGNN [10], a real-time graph prediction accelerator implemented on an Xilinx Alveo U50 FPGA at 300 MHz. In addition, we compared with HyGCN [1], a representative ASIC node prediction accelerator. HyGCN operates at 1 GHz, with 32 16-core SIMDs and 8 4×128 systolic modules.

B. Comparison under Various Batch Scenarios

For all speedup comparisons, we measured the total execution time over the entire test dataset for each platform.

TABLE III: Latency per graph (ms/graph) at batch size 1.

Dataset	models	FlowGNN	GPU	GN-Drive
HEP	GCN	0.16	0.29	0.025
	GIN	0.18	0.24	0.028
	GIN-VN	0.20	0.33	0.030
	GAT	0.05	0.28	0.019
	PNA	0.15	0.48	0.058

Small-batch real-time scenario: We compared GN-Drive with FlowGNN, a prior real-time graph prediction accelerator. We used the HEP dataset, where real-time processing is required. The evaluation results are illustrated in Fig. 10(a). When GN-Drive was evaluated with a single GE, it achieved an average speedup of $4.97\times$ over FlowGNN. This improvement is mainly due to the HSLG pipelining. When the batch size was increased to 6, enabling graph-level parallelism, GN-Drive fully utilized all 6 GEs, achieving an average speedup of $27.25\times$ over FlowGNN, demonstrating the effectiveness of its graph-level parallelism and the HSLG pipelining.

Table III further presents the per-graph latency of each platform. At a batch size of 1, the GPU exhibits relatively high latency due to constrained batch-level parallelism. GN-Drive uses only a single GE in this setting and achieves an average speedup of $4.97\times$ over FlowGNN operating at 300 MHz. Even after frequency-normalizing FlowGNN to 1 GHz, GN-Drive still achieves an average speedup of $1.49\times$.

Large-batch high-throughput scenario: Since GPUs provide

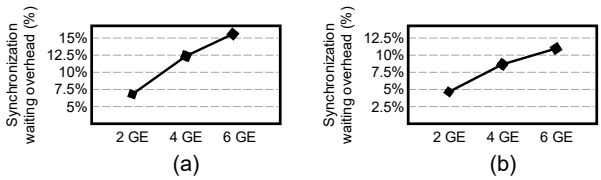


Fig. 14: Synchronization waiting overhead with varying numbers of GEs on (a) MolHIV and (b) MolPCBA. Results are averaged across all evaluated GNN models.

high throughput in large-batch settings, we evaluated GN-Drive on a GPU with large batch sizes. We used the MolHIV and MolPCBA datasets, where high-throughput molecular screening is required. The results of this evaluation are presented in Fig. 10(b). Compared with the GPU baseline, GN-Drive achieves an average speedup of $1.79\times$ when the GPU batch size is set to 256, and $1.48\times$ when the batch size is increased to 512, resulting in an overall average speedup of $1.64\times$. This performance gain mainly comes from EC alleviating the memory bottleneck of the encoding layer and the HSLG pipeline, improving GConv execution efficiency through higher hardware utilization and the elimination of redundant GEMM operations.

TABLE IV: Throughput (kgraphs/s) comparison on MolHIV, MolPCBA, and HEP.

Dataset	Model	FlowGNN	GPU	GN-Drive
MolHIV	GCN	25	403	**652**
	GIN	20	362	**482**
	GIN-VN	16	340	**460**
	GAT	50	792	**986**
	PNA	16	151	**211**
MolPCBA	GCN	25	369	**643**
	GIN	20	329	**475**
	GIN-VN	16	287	**453**
	GAT	50	657	**896**
	PNA	16	131	**207**
HEP	GCN	6	46	**217**
	GIN	6	49	**189**
	GIN-VN	5	44	**185**
	GAT	20	82	**313**
	PNA	7	23	**99**

Table IV further presents the throughput of each platform. FlowGNN exhibits relatively low throughput due to its focus on low-latency real-time inference. GN-Drive achieves an average throughput improvement of $1.47\times$ over the GPU on MolHIV and MolPCBA, while achieving $4.18\times$ higher throughput on HEP.

C. Effects of Scale-Out Approach

Graph-Level Parallelism: Fig. 10(c) shows the effect of graph-level parallelism by comparing a single GE with 512 MAC units and 128 SIMD cores versus 6 such GEs. To fully benefit from graph-level parallelism, the portion of memory access time must be kept low. The EC helps reduce the number of DRAM accesses in the encoding layer. As a result, GN-Drive effectively exploits graph-level parallelism and achieves an average speedup of $5.04\times$ over a single GE.

We further analyze the synchronization overhead caused by graph-size imbalance as the number of GEs increases. We define the synchronization waiting overhead as the fraction of cycles for which a GE remains idle while waiting for the slowest GE in the synchronized group. As shown in Fig. 14, the average waiting overhead increases from 6.82% to 15.59% on MolHIV and from 4.65% to 10.97% on MolPCBA as the number of GEs increases. Even with six GEs, the average waiting overhead remains below 16%.

Scale-Up vs. Scale-Out. Fig. 11 shows the comparison results in terms of the hardware utilization of the scale-up design with a single large GE with 3072 MAC units and 768 SIMD cores and that of the scale-out design with 6 GEs, each with 512 MAC units and 128 SIMD cores. To quantify the impact of using small multiple GEs on hardware utilization, we focus on the GConv layers, which dominate the end-to-end latency and largely determine the overall execution characteristics of the model. On average, the scale-out design improves hardware utilization by 18.75% compared to the scale-up design. Although GATConv shows lower utilization than the other models because its attention computation includes a GEMV operation, it still achieves a 13% higher utilization than the scale-up design. The higher hardware utilization rate results in better execution efficiency. That is, even the scale-out design with 6 small GEs uses a similar amount of computing resources as the scale-up design with a single large GE, it achieves higher hardware utilization, resulting in an average $2.06\times$ performance improvement compared to the scale-up design, as shown in Fig. 10(d).

TABLE V: Comparison of HyGCN and GN-Drive

Architecture	#MAC	#SIMD cores	HW util.	Speedup
HyGCN	4096	512	56%	1
GN-Drive	3072	768	**73%**	**2.61×**

Since HyGCN adopts a scale-up approach for node prediction by allocating substantial computing resources to a single large graph, we regard it as an architecture with a single large GE. We compare GN-Drive with HyGCN to confirm the advantage of the scale-out design with multiple lightweight GEs for graph prediction. Table V presents experimental results comparing HyGCN and GN-Drive in terms of computing resources, hardware utilization, and speedup. Speedup is normalized to HyGCN. The hardware utilization and speedup in Table V are averages on MolHIV, MolPCBA, and HEP. HyGCN employs substantially larger computing resources, but achieves only 56% hardware utilization. In contrast, GN-Drive achieves a hardware utilization of 73%, indicating that the smaller GE configuration is better for graph prediction workloads. Moreover, by combining EC for DRAM access reduction with a scale-out approach, GN-Drive achieves a $2.61\times$ speedup over HyGCN.

D. Effects of Encode Cache

We compare three cache types for the encoding layer: (i) No Cache (NC) with no optimization on the memory

TABLE VI: Sensitivity analysis of the number of EC entries.

Dataset		# EC Entries			
		0	2	4	8
MolHIV	Node hit ratio (%)	0.0	80.2	98.4	98.7
	Node encoding speedup ( )	1	5.63	85.93	109.98
	Edge hit ratio (%)	0.0	55.0	64.5	97.7
	Edge encoding speedup ( )	1	2.256	2.82	44.25
MolPCBA	Node hit ratio (%)	0.0	79.5	98.2	99.0
	Node encoding speedup ( )	1	5.48	26.48	188.83
	Edge hit ratio (%)	0.0	53.0	61.0	97.5
	Edge encoding speedup ( )	1	2.14	2.58	39.26

accesses of the encoding layer; (ii) Embedding Vector Cache (EVC), which caches the retrieved embedding vectors; and (iii) Encode Cache (EC) in our proposed design that caches the encoded outputs after accumulating retrieved embedding vectors. For simplicity, we refer to these designs as NC, EVC, and EC, respectively. We evaluate MolHIV and MolPCBA, excluding HEP because it does not perform embedding lookup for the encoding layer. EVC and EC use the same cache capacity and a direct-mapped organization for a fair comparison.

EC achieves an average cache hit ratio of 98%, while EVC achieves an average cache hit ratio of 99%. Although EVC achieves a slightly higher hit ratio, its latency benefit is limited because node and edge encoding still require multiple cache accesses for the encoded output even when the embedding vectors hit in the cache. In contrast, EC directly provides the encoded output with a single cache access. As shown in Fig. 12, the EVC-based design achieves an average speedup of $3.82\times$ over NC. The EC-based design achieves a higher average speedup of $6.68\times$. These results show that caching encoded outputs is more effective than caching individual embedding vectors for reducing encoding layer latency.

Fig. 13 shows the end-to-end latency ratio of the encoding layer on MolHIV as the number of GEs increases. We report GCN and PNA, where the impact of EC is the largest and the smallest, respectively. Without EC, the encoding layer latency ratio increases with more GEs because parallel GEs require more embedding table lookups from DRAM. With EC, this overhead is effectively suppressed by reducing DRAM accesses. In the 6 GE configuration, EC reduces the encoding layer portion of end-to-end latency by 74.64% for GCN, as shown in Fig. 13(a), and by 45.55% for PNA, as shown in Fig. 13(b), compared to the NC case.

Table VI shows the sensitivity of EC to the number of entries. Increasing the number of entries improves both the cache hit ratio and encoding performance. Four entries are sufficient to achieve over 98% hit ratio for node encoding, while edge encoding requires eight entries to achieve a comparable hit ratio. With eight entries, both node and edge hit ratios exceed 97% on MolHIV and MolPCBA, providing substantial encoding speedup over the NC configuration.

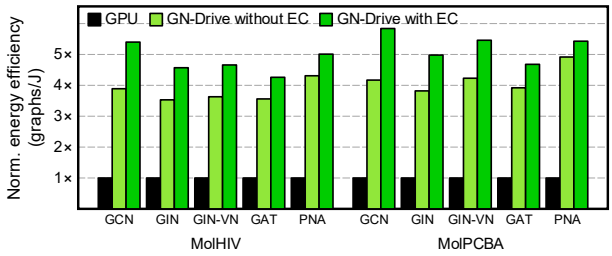


Fig. 15: Normalized energy efficiency of GN-Drive with and without EC across GNN models on MolHIV and MolPCBA. Energy efficiency is measured in graphs/J and normalized to the GPU baseline.

E. Hardware Cost and Energy Efficiency

TABLE VII: Area and power breakdown of GN-Drive.

Component	Area (mm^2)	Area (%)	Power (W)	Power (%)
MPU	3.92	47.92	3.45	58.24
GEMM	1.89	23.10	1.58	26.77
SRAM	1.80	22.04	0.57	9.67
Others	0.57	6.95	0.32	5.32
Total	8.18	100.00	5.92	100.00

Table VII summarizes the area and power breakdown of GN-Drive. The MPU and GEMM units together account for 71.02% of the total area and 85.01% of the total power, indicating that the compute units dominate both area and power consumption. SRAM accounts for 22.04% of the total and 9.67% of the total power. The complete GN-Drive accelerator, including six GEs and all on-chip buffers, occupies 8.18 mm^2 and consumes 5.92 W. The reported PPA excludes the external HBM2 device.

Fig. 15 compares the normalized energy efficiency of GN-Drive with and without EC. In this evaluation, the reported energy includes DRAM access energy. For GN-Drive, energy efficiency is calculated using the execution time obtained from cycle-accurate simulation, the synthesized power estimate, and the energy consumed by DRAM accesses. GN-Drive without EC achieves an average energy-efficiency improvement of $4.00\times$ over the GPU baseline, while GN-Drive with EC further improves this to $5.03\times$. Overall, EC improves the energy efficiency of GN-Drive by an average of $1.26\times$ by reducing DRAM access energy.

VII. Conclusion

In this paper, we presented GN-Drive, a GNN accelerator for graph prediction of numerous small-scale graphs. To process graph prediction workloads effectively, GN-Drive adopts a proposed hardware-friendly abstraction that improves resource efficiency by separating GEMM and SIMD operations into distinct stages. Based on this abstraction, GN-Drive designs lightweight Graph Engines with fewer compute resources to improve hardware utilization for small-scale graphs, and instantiates them in parallel to exploit graph-level parallelism and achieve high performance. In addition, we introduced Encode Cache to avoid the memory bottleneck in the encoding layer by reducing the number of DRAM accesses. Evaluation

results show that the GN-Drive achieves average speedups of 1.64×, 2.61×, and 1.49× over an RTX 4090, HyGCN, and FlowGNN, respectively.

APPENDIX

This appendix summarizes the mapping of GCN, GIN, GAT, and PNA onto HeavyApply (H), Scatter (S), LiteApply (L), and Gather (G).

GCN: The GCN layer used in our evaluation is expressed as follows:

$$\mathbf{h}_v^{new} = \frac{\mathrm{ReLU}(W\mathbf{h}_v + \mathbf{e}_{\mathrm{root}})}{d_v} + \sum_{u\in\mathcal{N}(v)} \frac{\mathrm{ReLU}(W\mathbf{h}_u + \mathbf{e}_{uv})}{\sqrt{d_u d_v}}. \quad (4)$$

In HSLG, the node projection is executed in H before S, while normalization and ReLU are executed in L, and neighborhood summation is executed in G. Accordingly, HSLG mapping is

$$H_{\mathrm{proj}} \to S \to L_{\mathrm{norm/ReLU/div}} \to G_{\mathrm{sum}}. \quad (5)$$

GIN: The GIN layer used in our evaluation is expressed as follows:

$$\mathbf{h}_v^{new} = \mathrm{MLP}\Big((1+\epsilon)\mathbf{h}_v + \sum_{u\in\mathcal{N}(v)} \mathrm{ReLU}(\mathbf{h}_u + \mathbf{e}_{uv})\Big). \quad (6)$$

In HSLG, edge-wise addition and ReLU are executed in L after S, followed by neighborhood summation in G. The two linear layers of the MLP are executed in H, while BatchNorm and ReLU between them are executed in L. Accordingly, the HSLG mapping is

$$S \to L_{\mathrm{sum/ReLU}} \to G_{\mathrm{sum}} \to H_{\mathrm{FC1}} \to L_{\mathrm{BN/ReLU}} \to H_{\mathrm{FC2}}. \quad (7)$$

The intermediate MLP result is fed back to the Input Buffer and re-enters H. Since the MLP requires no topology-dependent routing, S is bypassed during this additional pass.

GAT: The GAT layer first computes the projected node feature and attention score for each edge (u, v) as follows:

$$\mathbf{z}_v = W\mathbf{h}_v, \qquad e_{uv} = \mathrm{LeakyReLU}\big(\mathbf{a}_d^T\mathbf{z}_v + \mathbf{a}_s^T\mathbf{z}_u\big). \quad (8)$$

In HSLG, the node and attention projections are executed in H before S, while the edge-wise addition and LeakyReLU are executed in L after S. Accordingly, this part is mapped as

$$H_{\mathrm{proj}} \to H_{\mathrm{attn}} \to S \to L_{\mathrm{sum}} \to L_{\mathrm{LeakyReLU}}. \quad (9)$$

The attention coefficient and updated node feature are expressed as follows:

$$\alpha_{uv} = \frac{\exp(e_{uv})}{\sum_{r\in\mathcal{N}(v)}\exp(e_{rv})}, \qquad \mathbf{h}_v^{new} = \mathrm{ReLU}\Big(\sum_{u\in\mathcal{N}(v)} \alpha_{uv}\mathbf{z}_u\Big). \quad (10)$$

In HSLG, exponentiation, normalization, weighted-message generation, and ReLU are executed in L, while neighborhood summations are executed in G. Accordingly, this part is mapped as

$$L_{\mathrm{exp}} \to G_{\mathrm{sum}} \to L_{\mathrm{div}} \to L_{\mathrm{mul}} \to G_{\mathrm{sum}} \to L_{\mathrm{ReLU}}. \quad (11)$$

PNA: The PNA layer is expressed following the original formulation as follows:

$$X_i^{new} = U\Big(X_i, \bigoplus_{(j,i)\in E} M(X_i, E_{j\to i}, X_j)\Big). \quad (12)$$

Here, $\bigoplus$ combines mean, standard deviation, maximum, and minimum aggregators with identity, amplification, and attenuation degree scalers. In HSLG, the linear transformation in M is decomposed into destination-node, source-node, and edge projections using Eq. (3) and executed in H before S. Aggregation and statistical operations are executed in G/L, degree-dependent scaling in L, and the update transformation U reuses H. Accordingly, the HSLG mapping is

$$H_{\mathrm{proj}} \to S \to G/L_{\mathrm{agg}} \to L_{\mathrm{scale}} \to H_{\mathrm{update}}. \quad (13)$$

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT) (RS-2024-00409492) The EDA tool was supported by the IC Design Education Center(IDEC), Korea.

REFERENCES

[1] M. Yan *et al.*, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[2] T. Geng *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.

[3] R. Hwang *et al.*, "Grow: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[4] Y. Iiyama *et al.*, "Distance-weighted graph neural networks on fpgas for real-time particle reconstruction in high energy physics," *Frontiers in Big Data*, vol. 3, p. 44, 2021.

[5] F. Sindt *et al.*, "Structure-based virtual screening of ultra-large chemical spaces: Advances and pitfalls," *European Journal of Medicinal Chemistry*, vol. 305, p. 118576, 2026.

[6] L. Ma *et al.*, "NeuGraph: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019.

[7] H. Zhang *et al.*, "Understanding GNN computational graph: A coordinated computation, io, and memory perspective," in *Proceedings of the Fifth Conference on Machine Learning and Systems, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022*, D. Marculescu, Y. Chi, and C. Wu, Eds., 2022.

[8] W. Hu *et al.*, "Open graph benchmark: Datasets for machine learning on graphs," in *Proc. NeurIPS*, 2020.

[9] T. N. Kipf *et al.*, "Semi-supervised classification with graph convolutional networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[10] R. Sarkar *et al.*, "Flowgnn: A dataflow architecture for real-time workload-agnostic graph neural network inference," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[11] W. Yuan *et al.*, "Qegcn: An fpga-based accelerator for quantized gcns with edge-level parallelism," *Journal of Systems Architecture*, vol. 129, 2022.

[12] G. Corso *et al.*, "Principal neighbourhood aggregation for graph nets," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.

[13] S. Li *et al.*, "Dramsim3: A cycle-accurate, thermal-capable dram simulator," *IEEE Computer Architecture Letters*, 2020.

[14] K. Xu *et al.*, "How powerful are graph neural networks?" in *International Conference on Learning Representations*, 2019.

[15] P. Velickovic *et al.*, "Graph attention networks," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[16] G. Kasieczka *et al.*, "Top quark tagging reference dataset," Mar 2019.

[17] M. Fey *et al.*, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.