

FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference

Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, Cong Hao
School of Electrical and Computer Engineering, Georgia Institute of Technology
{rishov.sarkar, stefanabikaram, yhe374, lsathidevi3, callie.hao}@gatech.edu

Abstract—Graph neural networks (GNNs) have recently exploded in popularity thanks to their broad applicability to graph-related problems such as quantum chemistry, drug discovery, and high energy physics. However, meeting demand for novel GNN models and fast inference simultaneously is challenging due to the gap between developing efficient accelerators and the rapid creation of new GNN models. Prior art focuses on accelerating specific classes of GNNs, such as Graph Convolutional Networks (GCN), but lacks generality to support a wide range of existing or new GNN models. Furthermore, most works rely on graph pre-processing to exploit data locality, making them unsuitable for real-time applications. To address these limitations, in this work, we propose a generic dataflow architecture for GNN acceleration, named FlowGNN, which is generalizable to the majority of *message-passing* GNNs. The contributions are three-fold. **First**, we propose a novel and scalable *dataflow* architecture, which generally supports a wide range of GNN models with message-passing mechanism. The architecture features a configurable dataflow optimized for simultaneous computation of node embedding, edge embedding, and message passing, which is generally applicable to all models. We also propose a rich library of model-specific components. **Second**, we deliver ultra-fast real-time GNN inference without *any* graph pre-processing, making it agnostic to dynamically changing graph structures. **Third**, we verify our architecture on the Xilinx Alveo U50 FPGA board and measure the *on-board end-to-end* performance. We achieve a speed-up of up to $24\text{--}254\times$ against CPU (6226R) and $1.3\text{--}477\times$ against GPU (A6000) (with batch sizes 1 through 1024); we also outperform the SOTA GNN accelerator I-GCN by $1.26\times$ speedup and $1.55\times$ energy efficiency over four datasets. Our implementation code and on-board measurement are publicly available on GitHub.¹

I. INTRODUCTION

Graph Neural Networks (GNNs) have become a powerful tool to apply deep learning to tasks involving graph structures. Representative applications of GNNs include analysis of social networks and citation networks, recommendation systems, traffic forecasting, LIDAR point cloud segmentation for autonomous driving, high energy particle physics, molecular representations, and drug discovery [1]–[4].

Real-time Applications. GNN inference acceleration is in huge demand, especially for **real-time** processing. For instance, point cloud segmentation and detection for autonomous driving using GNNs [5] require real-time computation. Another concrete example is in high energy physics (HEP): collision data from a particle collider are collected every 25ns and must be processed using GNNs within nanoseconds with raw input graphs to decide whether to save or discard

the data; overrunning latency targets will overflow memory buffers and lose precious data [6]–[8]. Consequently, we focus on applications with two distinct features neglected by previous work. **First**, *many small graphs are consecutively streamed in at batch size 1 with extremely low latency* [9]–[11], in contrast to other applications where a single graph is processed. **Second**, real-time computation must be delivered in a *workload-agnostic* manner, without time for graph pre-processing or graph-specific optimizations.

Platform. FPGAs are desirable for energy-efficient, low-latency inference and have already been used in HEP for GNN acceleration [11] (however, they only implement one GNN on one hard-coded graph). Given the rapidly evolving GNN architectures, FPGA provides reconfigurability to quickly adapt to newly proposed GNNs and thus is highly practical.

The **challenges** and the **limitations** of existing accelerators, however, are significant. **First**, GNN computation is both communication- and computation-intensive, as also noted by previous literature [12]–[14]. **Second**, novel GNN models are rapidly emerging while accelerator innovation lags behind. Most state-of-the-art (SOTA) GNN accelerators are tailored for Graph Convolutional Networks (GCNs) [12], [15]–[17], which can be conveniently expressed as sparse and general matrix multiplications (SpMM and GEMM). However, complicated operations such as edge embedding, attention, mixed neighborhood aggregation, etc. make the majority of GNNs *unsuitable* for SpMM/GEMM. Therefore, to rapidly adapt to evolving GNN models, *generic, extensible, and flexible acceleration frameworks* are needed. **Third**, some accelerators focus on one large input graph and propose optimizations relying on graph properties. Examples include graph preprocessing to enhance data locality [12], [18]–[21], and graph partitioning relying on a property of a fixed input graph (e.g., by analyzing adjacency matrix sparsity [16]). Such preprocessing or graph-specific techniques *are not feasible* for real-time applications with millions of input graphs with varied structures.

Motivated by the emerging requirements and existing limitations, we propose a generic and flexible architecture for GNN acceleration on FPGA, named **FlowGNN**, which supports a wide range of prevailing GNNs with message-passing mechanism and is easily extensible for new models. We summarize our contributions as follows:

- **Generic.** (1) *Model-generic.* FlowGNN is the first generic GNN accelerator able to process the majority of state-of-the-art GNNs by supporting the *message passing* mechanism.

¹<https://github.com/sharc-lab/FlowGNN>

TABLE I
GNN coverage and features of **FlowGNN** in comparison with prior works.

Accelerator	Models						Edge embeddings	Anisotropic aggregations	Attention	Flexible dataflow	Multi-level parallelism*	No pre-processing
	GCN	GIN	GAT	PNA	DGN	VN						
AWB-GCN [22]	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
HyGCN [23]	✓	⊙ ¹	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗
I-GCN [17]	✓	⊙ ¹	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓
Auten <i>et al.</i> [14]	✓	✗	⊙ ²	✗	✗	✗	✗	⊙ ³	⊙ ²	✗	✓	✓
GCoD [24]	✓	⊙ ¹	✓	✗	✗	✗	✗	⊙ ³	✓	✗	✓	✗
ReGNN [25]	✓	⊙ ¹	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗
ReFLIP [26]	✓	⊙ ¹	✗	✗	✗	✗	✗	⊙ ³	✓	✗	✓	✓
FlowGNN	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

✓ Full support ⊙¹ Without edge embeddings ⊙² No attention normalization ⊙³ Supports attention only ✗ No support
*Supports parallelism both within nodes and between nodes during aggregation (inter- and intra-node parallelism).

TABLE II
Supported representative GNNs by our framework **FlowGNN** with flexible extensions.

Model	Representativeness
GCN [27]	GNN family that can be represented as sparse matrix-matrix multiplications (SpMM)
GIN [28], [29]	GNN family with <i>edge embeddings</i> and transformations where SpMM <i>does not</i> apply
GAT [30]	Anisotropic GNN family with sophisticated message functions
PNA [31]	A popular GNN family arbitrarily using multiple aggregation methods
DGN [32]	A state-of-the-art GNN with a directional flow at each node and guided aggregation
VN [33]	A widely used GNN technique with a virtual node connecting to all other nodes

GCN: graph convolutional network; GIN: graph isomorphism network; GAT: graph attention network; PNA: principal neighbourhood aggregation; DGN: directional graph network; VN: GNN with virtual node.

Table II summarizes currently supported GNN models, each representative of a large GNN family. In particular, we emphasize FlowGNN’s support for *edge embeddings*, which are not considered by existing accelerators but are widely used in most GNN models. (2) *Workload-generic*. FlowGNN is also dataset and graph structure agnostic; its optimizations do not rely on analysis for specific input graphs, but can effectively process a series of graphs with arbitrary structure.

- **Real-time**. FlowGNN targets *real-time* applications with *zero preprocessing* and partitioning, where the graphs are streamed in consecutively and processed on-the-fly.
- **Dataflow architecture**. We propose a novel dataflow architecture, which can effectively overlap the two most time-consuming steps in GNN, node transformation and message passing, and significantly reduce processing units’ idle time. To boost the performance, FlowGNN also exploits multiple levels of parallelism: node, edge, scatter, and apply, via a novel multi-queue-based on-the-fly multicasting adapter.
- **Open-source and modularized**. The implementation of FlowGNN is publicly available, with on-FPGA measurement and guaranteed end-to-end functionality by cross-checking with PyTorch code. It has a rich library of GNN modules and an easy-to-use programming model for developing new GNN models inside FlowGNN framework.
- **Evaluation**. We verify the proposed architecture on Xilinx Alveo U50 FPGA by measuring its *on-board* performance. We use seven popular datasets totaling more than 57k

graphs being streamed into FlowGNN. FlowGNN achieves a speed-up of 54–254× against CPU (6226R) and 1.3–477× against GPU (A6000) with batch sizes from 1024 to 1 with 4× less power. Comparing with the SOTA accelerator I-GCN, we observe 1.03× and 1.25× better performance. The remarkable speedup suggests that *we did not sacrifice performance for generality*, given that the goal of FlowGNN is a general framework for advanced GNNs.

II. RELATED WORK AND MOTIVATIONS

A. Related Work

Recent GNN accelerators are summarized by a survey [34], including acceleration on CPU/GPUs, ASICs, FPGAs, and heterogeneous platforms. Auten *et al.* [14] and HyGCN [23] are the earliest ASIC GNN accelerators. AWB-GCN [22] is an FPGA accelerator that aims to combat workload imbalance in graph processing. EnGN [35] uses PEs connected in a ring and performs aggregations using a technique called Ring-Edge Reduce. GCNAX [36] addresses resource underutilization and excessive data movement using a flexible dataflow. Rubik [37] and GraphACT [38] aim to accelerate GCN training using ASIC and FPGA, respectively. BoostGCN [16] specifically optimizes GCN via sparsity analysis and graph partitioning. I-GCN [17] is the most recent GCN accelerator delivering the state-of-the-art performance, which uses an islandization approach to de-duplicate redundant GCN computations by merging nodes with shared neighbors.

B. Limitations

Despite the great success of existing GNN accelerators and general graph processors, there are still limitations being overlooked. The most significant is that **advanced GNNs cannot be simplified as matrix multiplications, and edge embeddings cannot be ignored**. The majority of existing GNN accelerators focus mainly on GCNs and simplify the computation as a series of SpMM and GEMM. However, this simplification is *not valid* for many reasons. ① **Edge embedding**. Edge embeddings are used to represent important edge attributes, such as chemical bonds in a molecule [40], [41] or relationships in social networks, and thus are extensively used in advanced GNNs [31]–[33]. If there are multi-dimensional edge features, the SpMM/GEMM formulation no longer holds. Consider GIN [29] with edge embeddings:

$$x_i^{l+1} = \text{MLP}((1 + \epsilon) \cdot x_i^l + \sum_{j \in \mathcal{N}(i)} \text{ReLU}(x_j^l + e_{j,i}^l)) \quad (1)$$

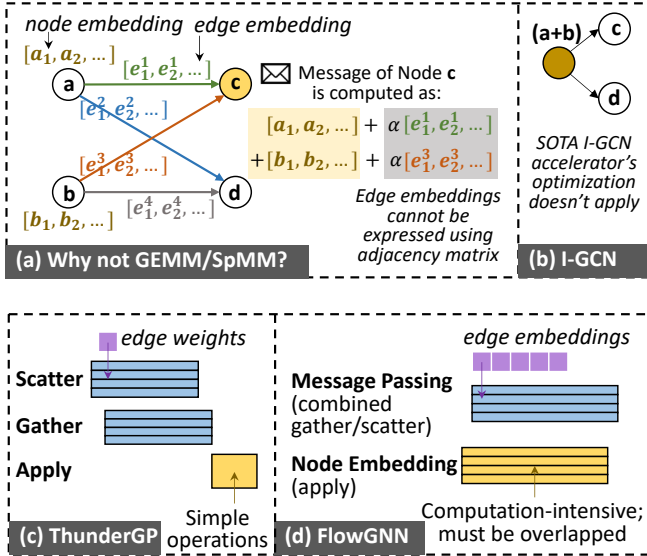


Fig. 1. (a) GNNs cannot be expressed as a sequence of GEMM and SpMM if there are edge embeddings. (b) With edge embeddings, redundancy removal by node merging in I-GCN [17] is not applicable. (c) and (d) shows the difference between ThunderGP [39] and FlowGNN.

where $x_i \in X$ is node embedding and $e_{j,i} \in E$ is edge embedding. Because the node feature matrix $X^{N \times F}$ (N nodes with embedding dimension F) and edge embedding matrix $E^{M \times D}$ (M edges with embedding dimension D) differ in size, the values have to be added differently. Specifically, the message transformation $\phi(x_j^i, e_{j,i}^l)$ must be computed once for each edge j, i , as Fig. 1(a) depicts. In contrast, without edge embeddings, $\phi(x_j^i)$ can be computed once for each j and reused for all i . **2 Invalid existing optimizations.** Edge embeddings make some GNN optimization techniques *unusable*. For example, SOTA GCN accelerator I-GCN [17] proposes redundancy removal by merging nodes with same neighbors to reduce computation, like nodes a and b in Fig. 1(b). However, with edge embeddings, messages from a to c and b to c now differ from those from a to d and b to d , and thus a and b cannot be merged. **3 Non-trivial aggregation.** GNNs using aggregations besides summation are inexpressible as SpMM. For instance, the Principal Neighborhood Aggregation (PNA) GNN in our framework uses mean, standard deviation, min, and max aggregations altogether in a weighted manner, whose coefficients must be computed on-the-fly and thus disrupts the simple GEMM or SpMM computation pattern. **4 Anisotropic GNNs.** Anisotropic GNNs require that a neighboring node transformation shall depend on the neighboring node and target node [42]. While for isotropic GNNs, propagation can be implemented using matrix multiplication-style approaches, for anisotropic ones, messages must be explicitly materialized [42]. One example is Graph Attention Networks (GAT) in our framework. In a GAT layer, a neighboring node is scaled by an attention coefficient α , which is dynamically computed based on all neighboring nodes. This dynamic nature prevents GAT from being expressed as matrix multiplications.

Therefore, existing accelerators, which heavily focus on GCN, rely on SpMM/GEMM formulation, and ignore edge

embeddings, can greatly limit their generalization to advanced and emerging new GNNs, such as GAT, GIN, PNA, etc. These advanced GNNs, listed in Table II, are **fundamentally different** in their computation because they **require explicit processing of each edge**; existing SpMM-/GEMM-based accelerators **are unusable** on such models since edge processing will disrupt the GEMM computation patterns; edge embeddings will also invalidate some optimization techniques, e.g., the node redundancy removal in I-GCN [17]. Meanwhile, many existing accelerators apply graph *pre-processing* to exploit data locality or partitioning on CPU, such as GraphACT [43], HyGCN [23], VersaGNN [44], and BoostGCN [16]. Such pre-processing is usually excluded from performance measurement and can be impractical for real-time applications.

C. Motivations and Innovations

Motivated by the above limitations, our goal is to develop a *generic architecture* that can support a wide range of GNNs and can rapidly adapt to emerging GNNs in the future. The main features of our proposed framework are two-fold: **1 an explicit message passing mechanism** that significantly improves the generality of the GNN accelerator for several reasons; **2 multi-level parallelism** (inter-node, intra-node, inter-edge across both message passing and node transformation) via multi-queue-based dataflow that can significantly improve performance without losing generality.

1 Explicit message passing for generality. First, message passing can express almost all GNN architectures at the theoretical formulation level. As a recent work [45] states, “any function of interest we want to compute over graphs can, in all likelihood, be expressed using pairwise message passing.” Second, explicit message passing can easily integrate edge embeddings and different aggregations without changing the fundamental architecture. We demonstrate the great generality of FlowGNN using six different GNN models (Table II).

Efficiently supporting message passing without losing generality is non-trivial, however, given the nature of irregular memory access since we can no longer exploit mature SpMM/GEMM optimization techniques. The *real-time* requirement elevates the challenge: we cannot apply any pre-processing to enforce data locality.

2 Multi-queue multi-level parallelism for performance. To address the challenges, our key innovation for high performance is a novel **multi-queue-based dataflow architecture**. It can effectively pipeline node transformation and edge embedding, the two most computation-intensive steps in GNN, with multiple levels of parallelism. In our experiments, our architectural innovation provides all the performance boost with no pre-processing or graph manipulation. Nevertheless, FlowGNN is orthogonal to certain optimization techniques such as on-the-fly partitioning and can be applied together.

These innovations make FlowGNN a feature-rich framework with excellent GNN model coverage, as shown in Table I. FlowGNN fully supports edge embeddings, unlike prior works, as well as anisotropic aggregations and self-attention, which are usually supported incompletely or not at

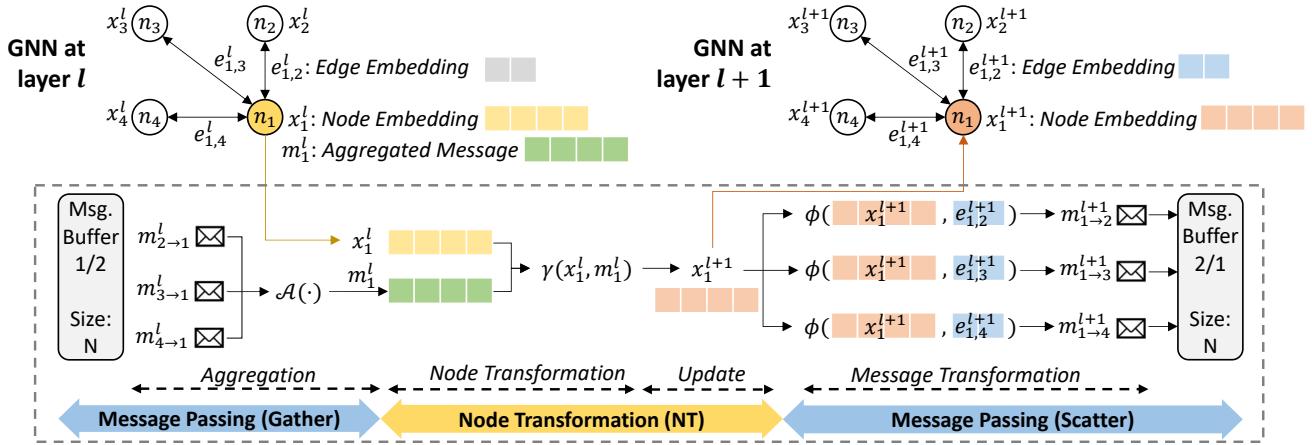


Fig. 2. The generic message passing mechanism of prevailing GNN models. It has two major stages: Node Transformation (NT) and Message Passing (MP). NT is analogous to *apply* and MP is analogous to *scatter/gather*. Both NT and MP are usually computation-intensive, especially NT, which typically uses linear or multi-layer perceptron (MLP).

all, with zero pre-processing required. FlowGNN also boasts both a flexible dataflow and multi-level parallelism. Existing GEMM-based accelerators exploit both inter- and intra-node parallelism but cannot support general message passing; the only gather/scatter-based accelerator HyGCN does not employ inter-node parallelism in aggregation. FlowGNN’s multi-queue-based multicasting enables both intra- and inter-node parallelism within a message passing skeleton.

Since we support explicit message passing, which is similar to the *gather-apply-scatter* (GAS) model in general graph processing, a natural question is how FlowGNN differs from GAS processors. Fig. 1(c) and (d) depict the major difference between SOTA graph processor ThunderGP [39] and our proposed FlowGNN. In ThunderGP, the major performance improvement comes from the pipelined *scatter* and *gather*, followed by *apply*. This is acceptable because in graph processing, *apply* usually is a simple operation such as scalar summation. In GNN, however, both message passing (analogous to *gather/scatter*) and node transformation (analogous to *apply*) are computation-intensive, and thus must be effectively pipelined, as we emphasize in FlowGNN.

III. GENERIC ARCHITECTURE

In this section, we introduce FlowGNN’s generic dataflow architecture, including background of message passing mechanism of GNNs (Sec. III-B), a baseline simple dataflow design without node-/edge-level parallelism (Sec. III-C), and an improved FlowGNN with multi-level parallelism via data queues (Sec. III-D). Sec. IV introduces model-specific components for various GNNs. Sec. V introduces the programming model.

A. FlowGNN Framework Features and Supported GNNs

FlowGNN Features. Our goal is to provide a real-time GNN acceleration architecture with great generality to support a wide range and emerging GNNs with minimum modifications. Sec. I summarizes the key differentiating features from state-of-the-art accelerators: **generic**, **real-time**, and **open-source**. **Supported GNNs.** Table II summarizes currently supported GNNs, each being representative of a family of GNNs. Graph

Convolutional Network (GCN) [27] represents those which can be formulated as SpMM; simplified GCN [46] also falls into this category. Graph Isomorphism Network (GIN) [28] represents advanced GNNs with higher representation power, including edge embeddings and transformations where SpMM does not apply; GraphSage [47] falls into this category. Principal Neighborhood Aggregation (PNA) [31] represents a popular GNN family that uses multiple arbitrary aggregation methods simultaneously. Graph Attention Network (GAT) [30] represents the anisotropic GNN family with sophisticated message functions, in which edges must be materialized [42]. Directional Graph Network (DGN) [32] is a state-of-the-art GNN with directional flow at nodes with guided aggregation. GNN with virtual node (VN) [33] is a widely used GNN technique using virtual nodes connected to all other nodes.

B. Message Passing Mechanism

Most prevailing GNN architectures follow the message passing mechanism [27], [28], [30]–[33], [46]–[49]. The general computation of a message passing GNN can be expressed as:

$$x_i^{l+1} = \gamma \left(x_i^l, \mathcal{A}_{j \in \mathcal{N}(i)} (\phi(x_j^l, x_j^l, e_{i,j}^l)) \right) \quad (2)$$

where each node i ’s embedding x_i^{l+1} at layer $l+1$ is expressed in terms of its embedding x_i^l at layer l along with the embeddings x_j^l of its neighbors $j \in \mathcal{N}(i)$ and their respective edge embeddings $e_{i,j}^l$ through a differentiable message transformation $\phi(\cdot)$, a permutation-invariant aggregation function $\mathcal{A}(\cdot)$, and a differentiable node transformation $\gamma(\cdot)$.

Fig. 2 demonstrates the message passing procedure for a single node n_1 at layer l , which will be repeated for all nodes and layers. Highlighted at the bottom of the figure, there are two major steps for each node in each layer: **message passing (MP)** and **node transformation (NT)**. MP is divisible into *gather* and *scatter* phases, where *gather* consists of feature aggregation and *scatter* consists of message transformation and passing. Edge embeddings are incorporated into the message during *scatter* phase. NT is usually a linear layer or multi-layer perceptron (MLP), followed by node update.

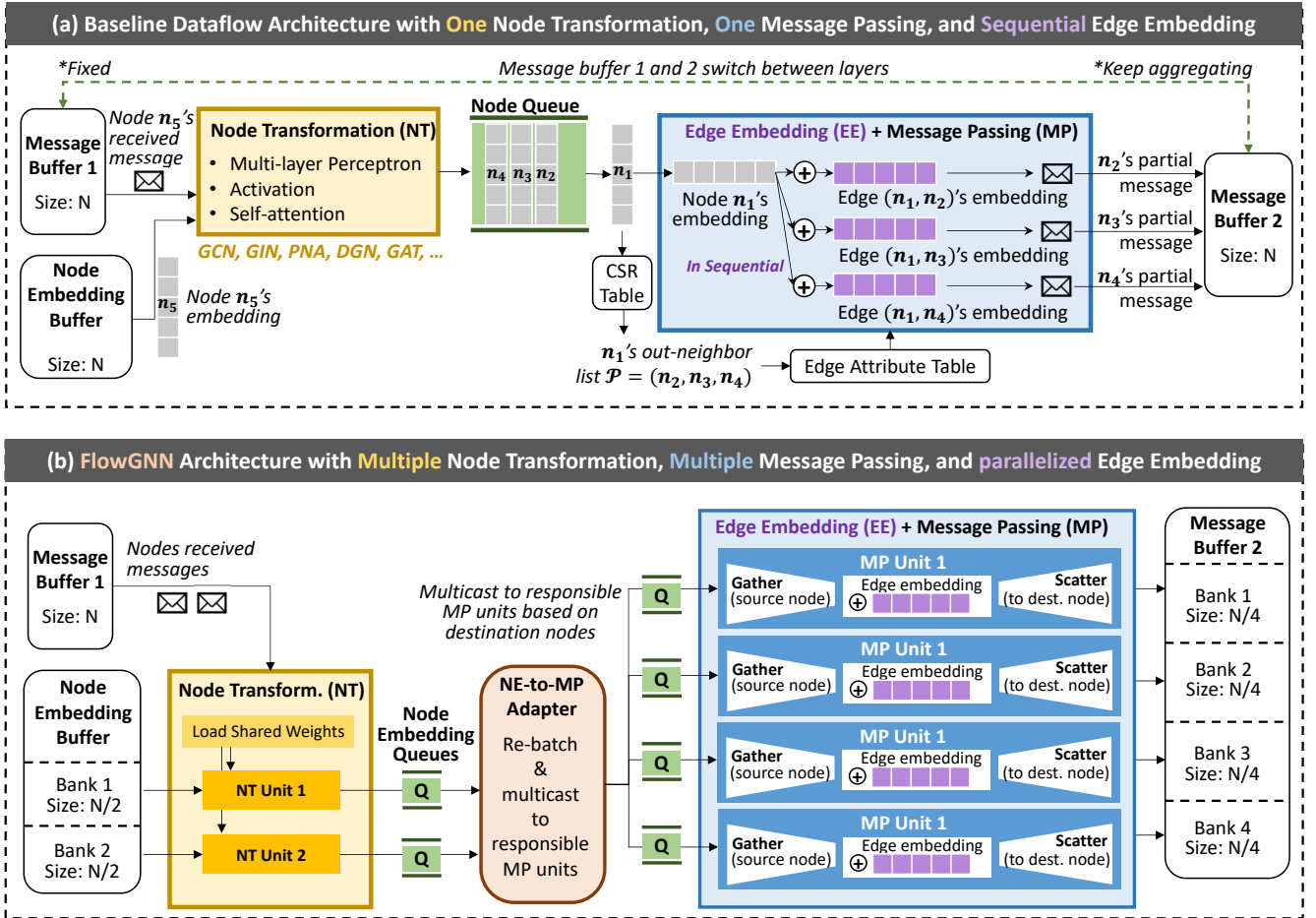


Fig. 3. Our proposed baseline dataflow architecture and the improved FlowGNN architecture. (a) The baseline dataflow architecture can effectively pipeline the Node Transformation (NT) and Message Passing (MP), but processes only one node and one edge at a time. (b) The improved FlowGNN architecture can process multiple nodes and multiple edges simultaneously, enabled by an NT-to-MP adapter via on-the-fly multicasting.

More specifically, the GNN computation flow has the following stages, as demonstrated in Fig. 2:

Message Passing (Gather). In the *gather* phase, a.k.a. aggregation, of a certain node n_1 , the messages from its neighbors obtained in the previous layer are retrieved from a message buffer. The messages are then aggregated in a permutation-invariant manner, denoted by $\mathcal{A}(\cdot)$ (e.g., sum, max, mean, std. dev.). In advanced GNNs such as PNA, multiple aggregators are used with learnable weights and scaled based on the degree of the target node. The aggregated message is denoted by m_1^l .

Node Transformation. After aggregation, m_1^l is processed together with node n_1 's current node embedding, denoted by x_1^l , via a node transformation function $\gamma(\cdot)$. This function, with inputs m_1^l and x_1^l , might be an identity, fully-connected layer, weighted sum, or an MLP. $\gamma(\cdot)$ produces a new node embedding of n_1 , denoted by x_1^{l+1} , and applies the update.

Message Passing (Scatter). After node transformation is the *scatter* phase of message passing. The new node embedding x_1^{l+1} will be transformed by a message transformation function $\phi(\cdot)$, usually together with an edge embedding $e_{src,dest}^{l+1}$, to generate the node's outgoing messages. Messages will be dispatched to all neighbors, which will eventually be collected by the *gather* stage of the next layer.

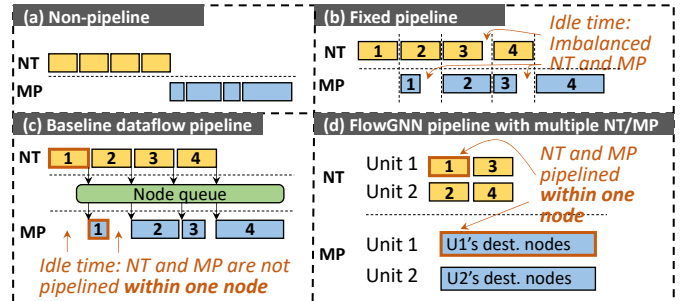


Fig. 4. Different strategies of pipelining of node transformation (NT) and message passing (MP). The proposed FlowGNN pipeline in (d) explores node/edge level parallelism and can pipeline NT and MP within one node.

A complete GNN model may consist of multiple layers, each with message passing and node transformation steps. For graph-level tasks, a global pooling layer is needed, possibly followed by MLP layers for final prediction.

C. Baseline Dataflow Architecture

To explicitly support the message passing mechanism, we first propose the baseline dataflow architecture, shown in Fig. 3(a). It has two major processing components: one Node Transformation (NT) unit (yellow block), and one Message

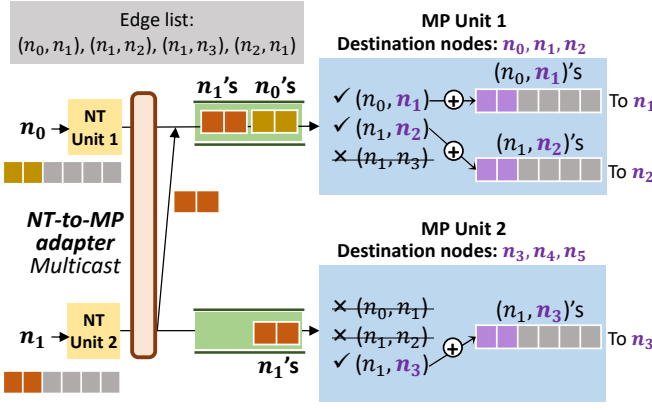


Fig. 5. MP units process edges and pass messages based on destination nodes. The NT-to-MP adapter multicasts the node embeddings to the correct MP unit. Note that the MP doesn't wait for the entire node embedding; it starts as soon as the first few elements arrive.

Passing (MP) unit (blue block), where edge embeddings are computed during MP. A node queue between NT and MP, implemented as a FIFO (first-in first-out), holds nodes (their embeddings) that are transformed and ready for message passing. This queue is the key to enable pipelined NT and MP: as long as the queue is not empty or full, NT and MP can run in parallel. We keep a high-level abstraction of NT and MP units for now and will introduce more details in Sec. III-D2.

Data Buffers. The architecture has three data buffers: one node embedding buffer and two message buffers, all with N entries. The two message buffers act alternately across layers: during layer 1, message buffer 1 is read-only while message buffer 2 is being updated; during layer 2, message buffer 2 becomes read-only and buffer 1 updates, and so on.

Execution Flow. Fig. 3(a) illustrates the execution flow of one GNN layer; for multiple layers, the same resources and dataflow will be reused. Within one layer, the NT unit accepts one node, e.g., n_1 , together with its received message, and applies node transformation and update, e.g., MLP, activation, and self-attention. This is the main component that distinguishes different GNN models. Then, the MP unit performs the subsequent scatter operation for n_1 , by sending out messages to all n_1 's neighboring nodes. Consider the graph in Fig. 2 as an example. Once n_1 's embedding is updated, the MP unit retrieves all its neighbors, i.e., n_2 , n_3 , and n_4 , computes the messages together with edge embeddings $e_{1,2}$, $e_{1,3}$, and $e_{1,4}$, and dispatches the messages. The receivers will instantly update their partially aggregated message in the message buffer. Thus the scatter and gather phases can be merged, since the aggregation function is permutation-invariant, so aggregation order does not matter. Such a merged fashion has two merits. First, it reduces overall process latency by fusing two stages into one. Second, it reduces memory cost from $O(E)$ to $O(N)$ where E is the number of edges and is typically much larger than N . This approach requires that graph data is stored in its compressed sparse row (CSR) format.

Equivalently, one can first perform gather, i.e., aggregation, via incoming edges, and then node transformation; in this case, no scatter is needed. FlowGNN also supports this dataflow,

which is more favorable for GAT. It requires two node embedding and one message buffer, all of size $O(N)$, and graph data stored in compressed sparse column (CSC) format.

Fig. 4 illustrates different strategies for NT and MP processing. **a Non-pipeline.** In Fig. 4(a), NT and MP are not pipelined; this apparently incurs a huge waste of idle cycles.

b Fixed pipeline. In Fig. 4(b), NT and MP are pipelined in a fixed manner: NT for the second node is pipelined with MP for the first node, etc. This reduces some latency but suffers from imbalanced NT and MP processing time. Specifically, if some nodes have larger degrees with longer MP latency than NT while others have shorter MP, there still will be idle cycles.

c Baseline dataflow pipeline. In Fig. 4(c), NT and MP are pipelined flexibly using a node queue: as soon as a node finishes its NT, i.e., is ready for message passing, its embeddings are pushed into the queue; meanwhile, the MP unit reads from the queue, incorporates edge embeddings, and then does scatter. This approach can greatly reduce idle cycles.

Limitations of Baseline Dataflow Architecture. Although the baseline dataflow architecture can effectively pipeline NT and MP, there are multiple limitations. Most obviously, it can process only one node and one edge at a time, restricting an otherwise parallel task. In addition, as shown in Fig. 4(c), NT and MP are not pipelined within one node, meaning that the node cannot start its MP until its NT completes.

D. Proposed FlowGNN Architecture

Inheriting the advantages of the baseline dataflow architecture, we propose our FlowGNN architecture to address the limitations and to greatly boost performance with multiple levels of parallelism. FlowGNN allows the following configurable parallelization parameters:

- Node parallelism (P_{node}): how many nodes can be processed simultaneously by NT.
- Edge parallelism (P_{edge}): how many edges can be processed simultaneously by MP.
- Apply parallelism (P_{apply}): how many node embedding dimensions can be processed simultaneously by one NT.
- Scatter parallelism (P_{scatter}): how many edge embedding dimensions can be processed simultaneously by one MP.

Fig. 3(b) depicts FlowGNN architecture. It has multiple NT units and multiple MP units (2 and 4 in this example, but configurable); between them is an NT-to-MP adapter, connecting NT units and MP units using multiple data queues. To allow parallel access, the node embedding and message buffers are partitioned into multiple banks. We introduce each component in detail in the following.

1) Node/Edge Parallelism: FlowGNN's key improvement over baseline is node and edge parallelism, which is, however, non-trivial. While parallelizing NT across multiple nodes is straightforward, it is challenging to parallelize MP across multiple edges, due to the random access of reading edge attributes and reading/writing target node embeddings. Specifically, to process message passing for a group of edges in parallel, those edges and their target nodes must be in different memory banks

or buffers; without graph pre-processing, such accesses are random and thus cannot be parallelized.

To address this problem, we propose a novel multi-queue dataflow to enable parallelized NT and MP via on-the-fly node distribution. The key idea is to instantiate multiple NT and MP units, and let each MP unit process its own bank of edges and read/write its own bank of node embeddings. This way, all NT and MP units can operate in parallel with no conflicts.

To achieve this, we design an **NT-to-MP adapter** and perform *on-the-fly multicasting*. Given P_{node} NT units, P_{node} nodes are processed in parallel as one batch. During this batch of NT, the NT-to-MP adapter distributes each node's transformed embeddings only to MP units that need to perform any subsequent scatter operations. Multicasting is based on *target node IDs*, and edge attributes and node embeddings are also stored in different memory banks based on target node IDs; this ensures that each MP will process only those edges and scatter to only those nodes within in its own bank. This addresses the first limitation of the baseline dataflow architecture. Worth noting, for a given node, MP need not wait for its node transformation to complete for all dimensions; as soon as embedding values are computed, they are streamed into the data queue, which the MP can then fetch. This addresses the second limitation of the baseline architecture.

Fig. 5 presents an example of multiple NT and MP units working in parallel via multicasting. Assume the edge list is $\{(n_0, n_1), (n_1, n_2), (n_1, n_3), (n_2, n_1)\}$. With two NT units, n_0 and n_1 are processed in parallel. As soon as their first P_{apply} embedding elements are computed (2 in this example), the NT-to-MP adapter will send them to the data queue belonging to the correct MP unit. Assume MP unit 1 is responsible for edges whose destination nodes are in $\{n_0, n_1, n_2\}$; MP unit 2 is responsible for edges with destinations in $\{n_3, n_4, n_5\}$. Since n_0 's neighbor is n_1 , its embeddings should be sent to MP unit 1 only; since n_1 's neighbors are n_2 and n_3 , its embeddings should be sent to both MP units 1 and 2. Then, each MP unit only processes its own edges in its own memory bank. When the entire embedding is finished, MP unit 1 scatters to nodes n_1 and n_2 , while MP unit 2 scatters to node n_3 .

Fig. 4(d) shows the **FlowGNN pipeline**, a significant improvement over **Baseline**, with multiple NT and MP units executing in parallel, and pipelined NT and MP within one node.

2) *Apply/Scatter Parallelism: Node Transformation (NT) Unit*. The NT unit handles any per-node computations required for a GNN. In most GNN algorithms, this is one or more learnable fully-connected layers and activation functions to transform each node's embedding for each GNN layer.

The canonical FlowGNN NT unit consists of two sequential processes, overlapped between nodes using ping-pong buffers to hide latency: *accumulate* and *output*. *Accumulate* reads each node's aggregated message and computes a fully-connected layer in an input-stationary fashion: each fetched element of the input vector is used to update the entire output vector. Then, once *accumulate* is finished for an entire node, *output* performs any finalization steps, such as an activation function, and sends the resulting embedding to the multicasting adapter.

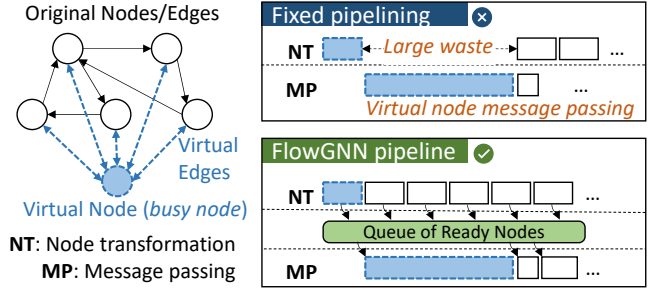


Fig. 6. FlowGNN's dataflow architecture is especially beneficial for models with virtual nodes.

The NT unit employs embedding-level parallelism using a configurable parameter P_{apply} , which determines the degree of parallelism across the node embedding dimension. Specifically, it controls how much of a node's message vector is read each cycle to be multiplied to update the output vector.

Message Passing (MP) Unit. The MP units handle per-edge computations. Two configurations are supported, depending on whether a GNN model is more suited to NT-to-MP (transform, then scatter) or MP-to-NT (gather, then transform) dataflow.

In the NT-to-MP dataflow, each MP unit handles an independent subset of *destination* nodes, as discussed in Sec. III-D1; analogously, in the MP-to-NT dataflow, each MP unit is assigned a subset of *source* nodes, gathering partial messages along edges from nodes within the assigned subset.

The MP unit also employs a configurable parameter P_{scatter} , determining the degree of parallelism across the edge embedding dimension. Notably, if P_{apply} and P_{scatter} are not the same, the NT-to-MP adapter will re-batch the embedding elements for alignment. For example, if $P_{\text{apply}} = 1$ and $P_{\text{scatter}} = 4$, the adapter will collect 4 elements from NT and then send to MP.

IV. MODEL-SPECIFIC COMPONENTS

In this section, we introduce model-specific optimizations on top of the general message passing architecture.

Graph Isomorphism Network. GIN is representative of GNNs where message passing involves edge embeddings, and where node transformation is computation-intensive using MLPs. Its message passing is within the framework using a customized message transformation $\phi(x, m) = x^l + \epsilon^l \cdot m^l$.

Graph Attention Network. GAT is representative of anisotropic GNNs because of its multi-head self-attention: incoming messages are weighted by attention coefficients based on the node and its neighbor's embeddings. GAT is fully compatible with FlowGNN through the MP-to-NT dataflow. GAT uses a custom message transformation that weights messages by an attention function $A(x_i, x_j)$ such as weighted sum or MLP.

Principled Neighbor Aggregation. PNA [31] uses multiple neighbor aggregations to increase the distinguishing power:

$$\oplus = \left[\frac{1}{\log(D_i + 1)/\tilde{D}} \right] \otimes \left[\frac{\mu}{\max} \right] \quad (3)$$

D_i is x_i 's degree, and $\tilde{D}$ is average node degree from training.

PNA falls into the message passing framework with a difference at the aggregation function. It has four aggregators, min, max, mean, and standard deviation, with scaling values.

```

1 GNN_compute(graph) {
2 // GNN compute skeleton; unchanged for all GNNs
3   for (layer in model) {
4 #pragma HLS dataflow // enable queues & pipelining
5     node_transformation(nodes_in, nt_out);
6     multicasting_adapter(nt_out, mp_in);
7     message_passing(mp_in, msg_buf);
8     aggregation(msg_buf, nodes_out);
9   }
10 node_transformation(nodes_in, nt_out) {
11   for (node in graph) {
12     nt_out << new_embedding;
13   }
14 message_passing(mp_in, msg_buf) {
15   for (edge in graph) {
16     mp_in >> node_data;
17   }
18 aggregation(msg_buf, nodes_out) {
19   for (node in graph) {
20   }

```

Listing 1: FlowGNN pseudocode showing which lines must be modified to create an accelerator for a new GNN.

Directional Graph Networks. DGN [32] uses vector fields to define directional flows at nodes for graph convolutions using anisotropic kernels. It accepts eigenvectors of the graph Laplacian as parameters to compute directional aggregation matrices during message passing. DGN uses the mean and directional derivative aggregators as $Y^l = \text{concat}\{D^{-1}AX^l, |B_{dx}^l X^l|\}$ where Y^l is aggregated messages, $X^{(t)}$ is node embeddings, D is the degree matrix, A is the adjacency matrix, and B_{dx}^1 is the directional derivative matrix. FlowGNN is trivially extensible to other DGN aggregations, e.g., directional smoothing B_{av} .

Virtual Node. Our dataflow architecture for pipelined node and edge processing especially benefits models with virtual nodes [33], artificial nodes connected to all other nodes in the graph. They provide shortcuts for message passing between node pairs and are demonstrated to be effective in many GNN models [50]–[52]. As shown in Fig. 6 left, a virtual node is *busy* with connections to all nodes, resulting in highly unbalanced workloads requiring special processing. Some models use multiple virtual nodes [51] which escalates the complexity.

Fortunately, the proposed dataflow pipelining easily resolves the imbalance introduced by virtual nodes without changing the framework. As shown in Fig. 6 right top, in fixed-pipeline or no-pipeline architectures, virtual node processing will take much longer than other nodes, causing a large waste. In contrast, Fig. 6 right bottom shows how the dataflow architecture can fully overlap virtual node processing with the node embedding computation for other nodes, with zero waste.

V. PROGRAMMING MODEL

We now describe the programming model of FlowGNN. Each GNN is compiled to its own kernel and deployed on FPGA. Using FPGA makes it easy to update the accelerator for new state-of-the-art GNN models.

We show the intended workflow for a hypothetical machine learning researcher Alice, who may have some experience with C++ coding but not much with High-Level Synthesis. We use pseudocode in Listing 1 to explain what needs to be changed to adapt to other GNNs in three cases.

TABLE III

Resource usage on Xilinx Alveo U50 FPGA. Clock frequency is 300 MHz.

Model	DSP	LUT	FF	BRAM
Available	5,952	872K	1,743K	1344
GIN	1,741	262,863	166,098	204
GCN	1,048	229,521	192,328	185
PNA	2,499	205,641	203,125	767
GAT	2,488	148,750	134,439	335
DGN	1,563	200,602	156,681	462

First, to try older GNNs such as GraphSage [47], SGC [46], PAN [53], UNet [54], or without edge embeddings, Alice can use our GIN, GAT, or PNA kernel without re-compiling and change only the kernel inputs. For example, she may set edge features to all zeros or set a specific aggregator weight to zero.

Second, suppose Alice finds a paper proposing a GNN beyond the current six models, NewGNN. It features an attention mechanism along with min, max, and mean aggregators. As the paper was just released, it has no hardware accelerator yet.

Luckily, FlowGNN already provides modular components with **extensive coverage for common GNN features**. The message passing skeleton remains unchanged (lines 1–10), and there are already components for attention computation (in our GAT model) and multiple aggregators (in our PNA model). Alice only needs to specify line 6 to be GAT transformation and line 9 to be PNA aggregator to build a NewGNN accelerator.

Third, shortly afterwards, Alice discovers a paper proposing NewerGNN, with novel aggregation and node transformation functions previously unseen in any GNN. Although FlowGNN does not come with built-in components for the new aggregator or node transformation functions, Alice can **only change a few lines of C++ code** to implement the new features, specifically, the highlighted lines in Listing 1. Alice is then able to deploy this accelerator just as quickly and easily as with NewGNN.

VI. EXPERIMENTS

A. Model and Implementation Details

We deploy FlowGNN for each of six models using High-Level Synthesis (HLS) by Vitis HLS and Vivado for the Xilinx FPGA Alveo U50 accelerator card, whose available resources are shown in Table III, targeting a 300 MHz clock frequency. Graphs are consecutively streamed into the accelerator in raw edge-list format (i.e., COO) with zero CPU intervention.

As listed in Table II, we implement six GNN models, each representative of a family of GNNs. Each model has a PyTorch implementation, to which we cross-check our on-board implementation to verify our end-to-end execution is correct. For GCN, GIN, and GIN-VN, we use 5 layers and node embedding dimension 100, mirroring the PyTorch models [55], along with global average pooling and an output head with one linear layer. For PNA, we use 4 layers with node embedding dimension 80, global average pooling, and an MLP-ReLU head of sizes (40, 20, 1). For DGN, we use 4 layers and node embedding dimension 100, global average pooling, and an MLP-ReLU head of sizes (50, 25, 1). For GAT, we use 5 layers with 4 heads and 16 features each, global average pooling, and an output head with one linear

TABLE IV

Number of graphs, (average) nodes, and (average) edges in each dataset used for evaluation. We also indicate datasets that include edge features (EF).

Dataset	Graphs	Nodes	Edges	EF
MolHIV	4113	25.3	55.6	✓
MolPCBA	43773	27.0	59.3	✓
HEP	10000	49.1	785.3	✓
Cora	1	2708	5429	✗
CiteSeer	1	3327	4732	✗
PubMed	1	19,717	44,338	✗
Reddit	1	232,965	114,615,892	✗

layer. We use two NT units and four MP units for all models. Table III reports each model’s resource utilization.

B. Dataset and Baseline

Datasets. Table IV lists the seven datasets used for evaluation. We first adopt MolHIV and MolPCBA, two molecular property prediction datasets from the Open Graph Benchmark [56], as well as 10k graphs from High Energy Physics (HEP) [57], generated according to the EdgeConv method [6], [10], [58] with $k = 16$. These datasets model our target use case of real-time inference on many small graphs (10–100 nodes) streamed in; similar applications have been demonstrated in other GNN works [10], [59]. Moreover, these datasets include edge features for comprehensive evaluation of GNNs with edge embeddings. We also use four prevailing single-graph benchmarks, CiteSeer, Cora, PubMed, and Reddit [60].

Baselines. As the baseline, we average five iterations of the time measured on CPU (Intel Xeon Gold 6226R) and GPU (NVIDIA RTX A6000), where each model is implemented in PyTorch Geometric [61]. On GPU, we use batch sizes from 1 to 1024. Note that batch size 1 is the *only fair comparison* for real-time inference of streaming graphs—batching graphs delays their processing. Other works [10], [62] also commonly compare only with batch size 1; we provide results at higher batch sizes only to affirm FlowGNN’s advantage.

We also compare with the state-of-the-art accelerator, I-GCN [17], following their experiment settings. Worth noting, as discussed in Sec. II-B, I-GCN employs a redundancy removal technique that skips explicit materialization of groups of edges, which limits I-GCN’s applicability to a narrow class of *isotropic* GNNs with *no edge embeddings*; thus this is *not a fair comparison* since I-GCN will fail in many cases where FlowGNN can manage. Nevertheless, we still make effort to make a comparison. For other GNN accelerators such as [12], [16], we find it hard to figure the GNN details such as the number of layers, node embedding dimensions, and whether graph/weight loading and graph-level processing are included. Therefore, our main comparison is with CPU and GPU using on-board measured performance with functionality guarantee.

C. End-to-end Evaluation

We fully evaluate FlowGNN implementations on six GNN models after place-and-route against CPU and GPU baselines. We measure latency **on-board end-to-end**, including weight

TABLE V

On-board latency (in ms) of FlowGNN compared against CPU and GPU at batch size 1, averaged from 10k graphs from high energy physics.

Model	CPU	GPU	FlowGNN (vs. GPU)
GIN	4.23	2.38	0.1799 (13.3×)
GIN+VN	5.02	3.51	0.2076 (16.9×)
GCN	4.59	3.01	0.1639 (18.4×)
GAT	2.24	1.96	0.0544 (36.0×)
PNA	9.66	5.37	0.1578 (34.0×)
DGN	30.20	61.26	0.1382 (443.4×)

TABLE VI

Energy efficiency (in graphs/kJ) of FlowGNN compared against CPU and GPU at batch size 1. The MolHIV dataset is used for energy evaluation.

Model	CPU	GPU	FlowGNN (vs. GPU)
GIN	4.48E3	4.50E3	7.34E5 (163×)
GIN+VN	3.16E3	2.99E3	6.46E5 (216×)
GCN	4.02E3	3.50E3	8.88E5 (254×)
GAT	6.29E3	5.41E3	2.29E6 (424×)
PNA	2.52E3	2.33E3	6.11E5 (262×)
DGN	1.40E3	7.96E2	1.39E6 (1748×)

loading and graph loading, and average across the test graphs in each dataset. We report speedups for each batch size.

Results are depicted in Fig. 7(a) for the MolHIV dataset and Fig. 7(b) for the MolPCBA dataset. In each figure, x -axis is GPU/CPU batch size; y -axis is the average latency per graph in milliseconds. GPU baseline is evaluated with batch sizes from 1 to 1024, and CPU is evaluated at batch size 1.

It clearly shows that, for six GNN models, on both datasets, FlowGNN achieves remarkable speedup over CPU and GPU. Compared with CPU, FlowGNN is 51.0–254.7× faster. Compared with GPU, FlowGNN is 53.4–477.6× faster at batch size 1. Additionally, FlowGNN consistently outperforms GPU at batch sizes up to 64, from 1.3× to 266.0×. The GAT and DGN models even surpass GPU at batch sizes 256 and 1024: GAT is 31.7–177.7× faster, and DGN is 3.7–4.4× faster.

Additionally, Table V lists results on our HEP dataset; all platforms use batch size 1 to simulate real-time high energy physics. FlowGNN is 13.3–443.4× faster than GPU and 23.5–218.6× faster than CPU, confirming its real-time advantage.

We also evaluated FlowGNN on six GNN models on two popular GNN datasets, Cora and CiteSeer; results are in Fig. 8. On both graphs, FlowGNN consistently outperforms CPU and GPU (batch size 1 since these are single graphs). GAT and DGN see especially massive speedups of 37.8× and 127.4× over GPU on Cora and 69.6× and 98.7× on CiteSeer.

Finally, we evaluate FlowGNN’s energy efficiency against CPU and GPU on the six models on the MolHIV dataset at batch size 1. Results in Table VI show 164–991× energy efficiency over CPU and 163–1748× over GPU.

The remarkable speedup and energy efficiency demonstrate the effectiveness of our proposed dataflow architecture, especially with no pre-processing. Although FlowGNN’s goal is to be a *generic* framework, the superior results suggest that we *did not sacrifice performance for generality*.

D. Ablation Study

We conduct an ablation study to quantitatively evaluate the baseline and FlowGNN architectures, using the GCN model on

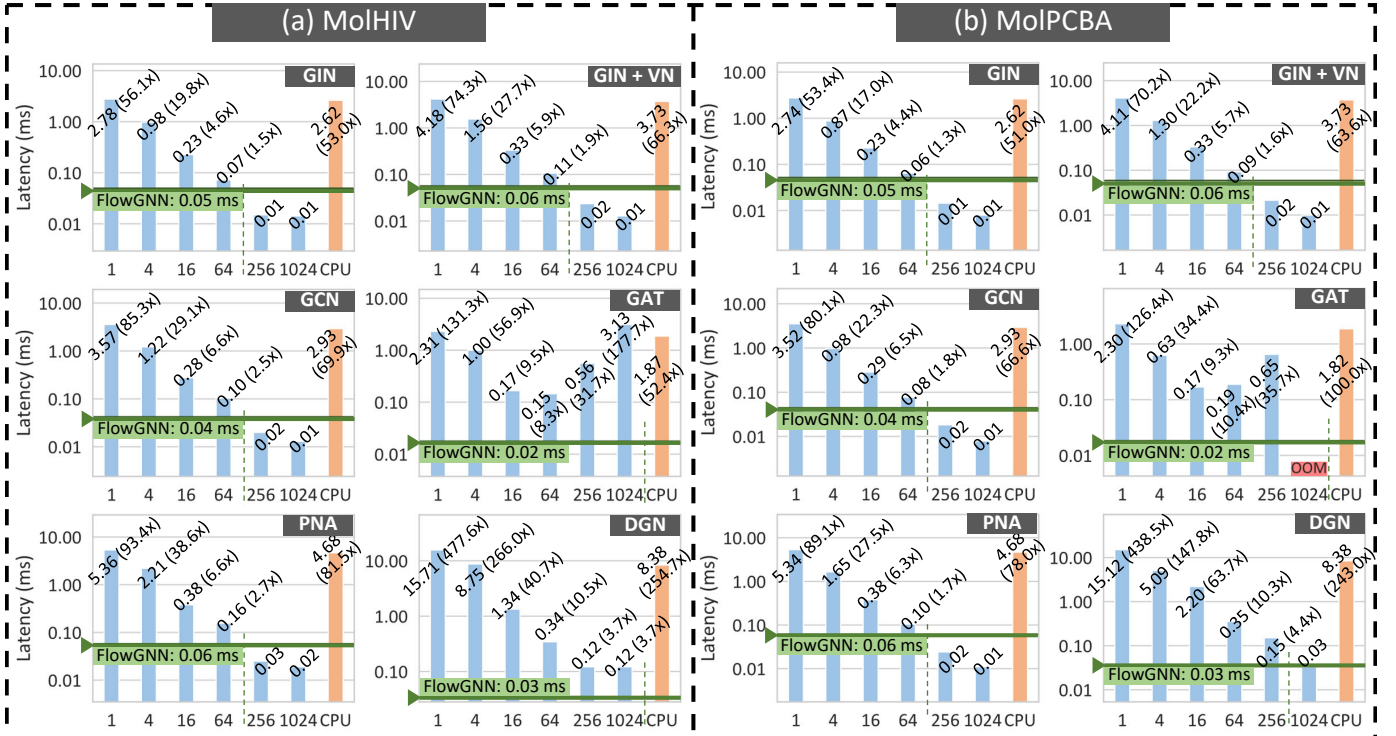


Fig. 7. FlowGNN latency measured on-board averaged from 40k test graphs. x -axis is GPU/CPU batch size; y -axis is average latency (ms) per graph. GPU baseline is evaluated with batch sizes from 1 to 1024. CPU is evaluated at batch size 1. (a) MolHIV results. (b) MolPCBA results. Our FlowGNN outperforms GPU by 3.58–477 $\times$ and is consistently faster than GPU with batch size ≤ 64 ; the GAT and DGN models even outperform GPU under batch size 1024.

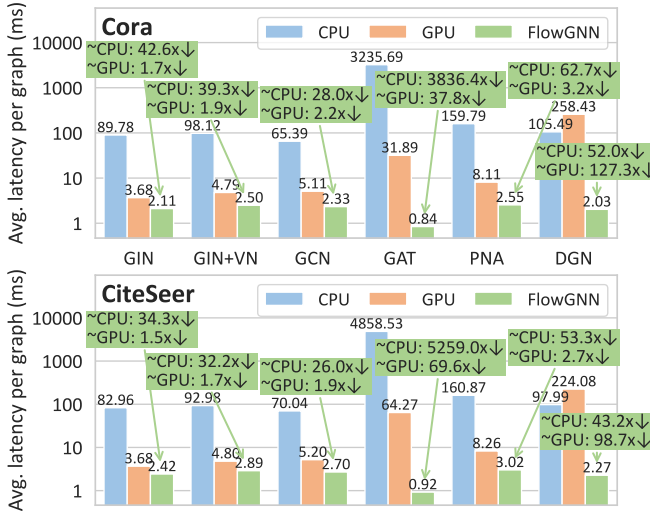


Fig. 8. FlowGNN latency on the Cora and CiteSeer datasets, compared against CPU and GPU baselines.

MolHIV dataset. Fig. 9 shows the incremental improvements over the most naive framework, non-pipelined NT and MP (Fig. 4(a), Sec. III-C). It shows that thanks to our customized NT and MP units with default parallelism (e.g., parallel linear and MLP), the non-pipelined NT and MP scheme is already 4.91 $\times$ faster than GPU. Next, it shows that fixed-pipeline is 1.66 $\times$ faster than non-pipeline, while the baseline dataflow achieves another 1.38 $\times$ speedup by reducing NT and MP idle cycles. Then, FlowGNN-1-1 ($P_{\text{apply}} = 1$, $P_{\text{scatter}} = 1$) is 1.45 $\times$ faster than the baseline dataflow, since it pipelines NT and MP

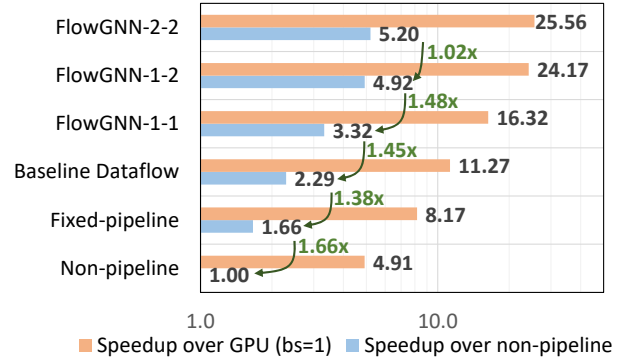


Fig. 9. Effectiveness of FlowGNN's dataflow architecture. Speedup plotted in log-scale. FlowGNN results are in the format of FlowGNN- P_{apply} - P_{scatter} .

for one node: MP can start before the entire NT is finished. We further increase the P_{scatter} from 1 to 2 and P_{apply} from 1 to 2, leading to 1.48 $\times$ and 1.02 $\times$ improvement, respectively. This ablation study shows that our proposed baseline dataflow architecture, along with pipelined NT/MP within one node and multiple NT/MP, are very effective in reducing the latency.

We then evaluate the impact of varying FlowGNN's configurable parallelism factors as a design space exploration (DSE). On top of the baseline architecture, we observe how varying P_{node} from 1 to 4, P_{edge} from 1 to 4, P_{apply} from 1 to 4, and P_{scatter} from 1 to 8 change the overall inference latency. We use GCN model on MolHIV dataset. We plot 108 data points in Fig. 10, showing the speedup over the baseline, where all four parameters are 1. First, each 3 $\times$ 3 block evaluates the impact of edge/node parallelism. Increasing P_{node} and P_{edge} usually

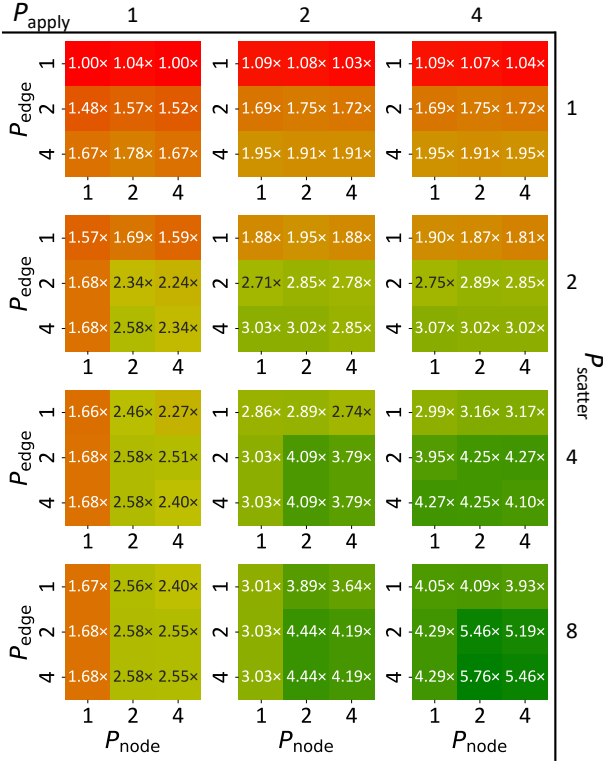


Fig. 10. Overall speedup resulting from different parallelization factors P_{node} , P_{edge} , P_{apply} , and P_{scatter} .

lead to speedup, from $1.33\times$ to $1.84\times$ when both parameters increase from 1 to 4. In some cases, increasing one parameter does not affect latency, depending on the bottleneck. When $P_{\text{apply}} = 1$, $P_{\text{scatter}} = 2$, and $P_{\text{node}} = 1$, increasing P_{edge} from 2 to 4 has no effect, but increasing P_{node} to 2 improves speedup from $1.68\times$ to $2.34\times$. Here, NT is the bottleneck, not MP. Second, increasing P_{apply} and P_{scatter} always seems effective, from $1.09\times$ to $2.99\times$ when both increase from 1 to 4. The largest speedup within the design space is $5.76\times$, when $P_{\text{edge}} = 4$, $P_{\text{node}} = 2$, $P_{\text{apply}} = 4$, and $P_{\text{scatter}} = 8$.

On one hand, the DSE study demonstrates that increasing the four parallelism parameters can effectively reduce latency. On the other hand, we recognize that the speedup is not linear, e.g., doubling one parameter does not result in $2\times$ speedup. One reason is the entangled effect of the four parameters, where some are bottlenecks while others are not; another reason is imbalanced MP workload due to imbalanced graphs.

E. Workload Imbalance Analysis

As edges are assigned to MP units by target node ID, workload imbalance may result. In Table VII, we analyze workload imbalance for varying values of P_{edge} , given by the largest difference in workloads between any two MP units as a percentage of the total workload. We see no more than 8.82% imbalance for all evaluated combinations of dataset and P_{edge} ; nevertheless, we will consider improvements in future work.

TABLE VII

Workload imbalance for varying values of P_{edge} . At 0%, all MP units have identical workloads; at 100%, one unit handles the entire workload.

P_{edge}	Datasets						
	MolHIV	MolPCBA	HEP	Cora	CiteSeer	PubMed	Reddit
2	6.41%	5.58%	2.47%	0.95%	0.40%	0.41%	0.04%
4	8.59%	7.78%	3.24%	3.83%	1.67%	2.21%	0.17%
8	8.82%	7.82%	3.30%	2.56%	2.69%	1.81%	0.28%
16	8.34%	7.62%	3.12%	2.72%	2.36%	1.23%	0.21%
32	7.37%	6.25%	3.75%	1.95%	1.68%	0.87%	0.21%
64	7.27%	6.28%	3.95%	1.82%	1.22%	0.82%	0.16%

TABLE VIII

Comparison with I-GCN [17] and AWB-GCN [22] on the Cora, CiteSeer, PubMed, and Reddit datasets using GCN. Although not a fair comparison, we still achieve an average of $1.26\times$ normalized speedup and an average of $1.55\times$ energy efficiency (EE) over I-GCN.

Dataset	Accelerator	Latency (μs)	DSPs	Normalized by DSPs	EE (graph/kJ)
Cora	AWB-GCN	2.3	4096	2.3	3.1E6
	I-GCN	1.3	4096	1.3	7.1E6
	FlowGNN	6.912	747	1.261 (1.03$\times$)	7.77E6 (1.09$\times$)
CiteSeer	AWB-GCN	4.0	4096	4.0	1.9E6
	I-GCN	1.9	4096	1.9	3.7E6
	FlowGNN	8.332	747	1.520 (1.25$\times$)	6.44E6 (1.74$\times$)
PubMed	AWB-GCN	30	4096	30	2.5E5
	I-GCN	15.1	4096	15.1	5.3E5
	FlowGNN	53.22	747	9.706 (1.56$\times$)	1.01E6 (1.91$\times$)
Reddit	AWB-GCN	3.2E4	4096	3.2E4	2.1E2
	I-GCN	3.0E4	4096	3.0E4	3.5E2
	FlowGNN	1.36E5	747	2.49E4 (1.20$\times$)	3.94E2 (1.13$\times$)

F. Comparison Against GCN Accelerators

As discussed earlier, most existing GNN accelerators formulate GNNs as GEMM/SpMM and cannot handle advanced features such as edge embeddings. I-GCN [17] and AWB-GCN [22] are two works of this type. In this section, we implement the GCN architecture of these existing works within the FlowGNN framework and compare its performance on the Cora, CiteSeer, PubMed, and Reddit datasets. We follow their model configuration, using a two-layer GCN, with node embedding dimension being 16 and no edge embedding.

Results are shown in Table VIII. Given the different hardware platform, we normalize by the number of DSPs, the major computation resource, and achieve average performance $1.26\times$ faster than I-GCN with $1.55\times$ energy efficiency, which is $2.21\times$ faster and $2.95\times$ more energy efficient than AWB-GCN. Although not a fair comparison because the optimizations in I-GCN and AWB-GCN do not generalize to edge embeddings, we still show superior performance.

VII. CONCLUSION

We proposed **FlowGNN**, the first generic and flexible accelerator framework for a wide range of GNNs. It uses a novel dataflow architecture with multiple levels of parallelism, including edge/node and apply/scatter parallelism. The enabling techniques are multiple data queues with on-the-fly node multicasting. Its noteworthy features include generality for future-proofing, real-time processing, and open-source. On-board evaluation with guaranteed functionality exhibited invariant speed-up comparing with CPU and GPU, up to 24–

254 $\times$ against CPU and 1.3–477 $\times$ against GPU with batch sizes from 1024 to 1. We are also 1.26 $\times$ faster and 1.55 $\times$ more energy efficient than the SOTA GNN accelerator. Remarkable speedup suggests that we did not sacrifice performance for generality and can deliver real-time performance.

ACKNOWLEDGEMENTS

The authors would like to thank Zihang Qiao for his contributions to the development of PNA and DGN, as well as Parima Mehta for her contribution to the virtual node model. The authors would also like to thank Dr. Pan Li for his insightful discussions.

REFERENCES

- [1] Damian Szklarczyk, Annika L Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda T Doncheva, John H Morris, Peer Bork, et al. STRING v11: protein-protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic acids research*, 47(D1):D607–D613, 2019.
- [2] David S Wishart, Yannick D Feunang, An C Guo, Elvis J Lo, Ana Marcu, Jason R Grant, Tanvir Sajed, Daniel Johnson, Carin Li, Zinat Sayeeda, et al. DrugBank 5.0: a major update to the DrugBank database for 2018. *Nucleic acids research*, 46(D1):D1074–D1082, 2018.
- [3] Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. MoleculeNet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.
- [4] Kenneth Atz, Francesca Grisoni, and Gisbert Schneider. Geometric deep learning on molecular representations. *Nature Machine Intelligence*, pages 1–10, 2021.
- [5] Weijing Shi and Raj Rajkumar. Point-GNN: Graph neural network for 3D object detection in a point cloud. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1711–1719, 2020.
- [6] Jonathan Shlomi, Peter Battaglia, and Jean-Roch Vlimant. Graph neural networks in particle physics. *Machine Learning: Science and Technology*, 2(2):021001, 2020.
- [7] Yutaro Iiyama, Gianluca Cerminara, Abhijay Gupta, Jan Kieseler, Vladimir Loncar, Maurizio Pierini, Shah Rukh Qasim, Marcel Rieger, Sioni Summers, Gerrit Van Onsem, et al. Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics. *Frontiers in big Data*, page 44, 2021.
- [8] Gianluca Cerminara, Abhijay Gupta, Yutaro Iiyama, Jan Kieseler, Vladimir Loncar, Jennifer Ngadiuba, Maurizio Pierini, Marcel Rieger, Sioni Summers, Gerrit Van Onsem, et al. Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction at the Large Hadron Collider. 2020.
- [9] V. Mikuni and F. Canelli. ABCNet: An attention-based method for particle tagging. *The European Physical Journal Plus*, 135(6):463, June 2020.
- [10] Huilin Qu and Loukas Gouskos. Jet tagging via particle clouds. *Physical Review D*, 101(5):056019, March 2020.
- [11] Abdelrahman Elabd, Vesal Razavimaleki, Shi-Yu Huang, Javier Duarte, Markus Atkinson, Gage DeZoort, Peter Elmer, Scott Hauck, Jin-Xuan Hu, Shih-Chieh Hsu, Bo-Cheng Lai, Mark Neubauer, Isobel Ojalvo, Savannah Thais, and Matthew Trahms. Graph neural networks for charged particle tracking on FPGAs. *Frontiers in Big Data*, 5:828666, March 2022.
- [12] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. Hardware acceleration of large scale GCN inference. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 61–68. IEEE, 2020.
- [13] Lixiang Li, Yao Chen, Zacharie Zirnheld, Pan Li, and Cong Hao. MELOPPR: Software/hardware co-design for memory-efficient low-latency personalized PageRank. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 601–606, 2021.
- [14] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.
- [15] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936. IEEE, 2020.
- [16] Bingyi Zhang, Rajgopal Kannan, and Viktor Prasanna. BoostGCN: A framework for optimizing GCN inference on FPGA. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 29–39. IEEE, 2021.
- [17] Tong Geng, Chunshu Wu, Yongang Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin Herbordt, Yingyan Lin, and Ang Li. I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1051–1063, Virtual Event Greece, October 2021. ACM.
- [18] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. Redundancy-free computation for graph neural networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 997–1005, 2020.
- [19] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. GNNAdvisor: An efficient runtime system for GNN acceleration on GPUs. *arXiv preprint arXiv:2006.06608*, 2020.
- [20] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. Rubik: A hierarchical architecture for efficient graph neural network training. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.
- [21] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. HyGCN: A GCN accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.
- [22] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 922–936, Athens, Greece, October 2020. IEEE.
- [23] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. HyGCN: A GCN accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29, San Diego, CA, USA, February 2020. IEEE.
- [24] Haoran You, Tong Geng, Yongang Zhang, Ang Li, and Yingyan Lin. GCoD: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 460–474. IEEE Computer Society, April 2022.
- [25] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. ReGNN: A redundancy-eliminated graph neural networks accelerator. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 429–443, April 2022.
- [26] Yu Huang, Long Zheng, Pengcheng Yao, Qinggang Wang, Xiaofei Liao, Hai Jin, and Jingling Xue. Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1029–1042, April 2022.
- [27] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2016.
- [28] Keyulu Xu et al. How powerful are graph neural networks? In *ICLR*, 2019.
- [29] Weihua Hu, Bowen Liu, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. Strategies for pre-training graph neural networks, 2019.
- [30] Petar Veličković et al. Graph attention networks. In *arXiv preprint arXiv:1710.10903*, 2017.
- [31] Gabriele Corso et al. Principal neighbourhood aggregation for graph nets. In *NeurIPS*, 2020.
- [32] Dominique Beani, Saro Passaro, Vincent Létourneau, Will Hamilton, Gabriele Corso, and Pietro Liò. Directional graph networks. In *International Conference on Machine Learning*, pages 748–758. PMLR, 2021.
- [33] Justin Gilmer et al. Neural message passing for quantum chemistry. In *ICML*, 2017.

- [34] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. Computing graph neural networks: A survey from algorithms to accelerators. *ACM Computing Surveys*, 54(9):1–38, December 2022.
- [35] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, Huawei Li, Dawen Xu, and Xiaowei Li. EnGN: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 70(9):1511–1525, September 2021.
- [36] Jiajun Li, Ahmed Lour, Avinash Karanth, and Razvan Bunescu. GC-NAX: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 775–788, Seoul, Korea (South), February 2021. IEEE.
- [37] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, and Yuan Xie. Rubik: A hierarchical architecture for efficient graph neural network training. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(4):936–949, April 2022.
- [38] Hanqing Zeng and Viktor Prasanna. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 255–265, Seaside CA USA, February 2020. ACM.
- [39] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. Thundergp: Hls-based graph processing framework on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 69–80, 2021.
- [40] Zhuoning Yuan, Yan Yan, Milan Sonka, and Tianbao Yang. Large-scale robust deep auc maximization: A new surrogate loss and empirical studies on medical image classification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 3040–3049, 2021.
- [41] Liyu Gong and Qiang Cheng. Exploiting edge features for graph neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9211–9219, 2019.
- [42] Shyam A Tailor, Felix Opolka, Pietro Lio, and Nicholas Donald Lane. Do we need anisotropic graph neural networks? In *International Conference on Learning Representations*, 2021.
- [43] Hanqing Zeng and Viktor Prasanna. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '20, page 255–265, New York, NY, USA, 2020. Association for Computing Machinery.
- [44] Feng Shi, Ahren Yiqiao Jin, and Song-Chun Zhu. Versagmn: a versatile accelerator for graph neural networks. *arXiv preprint arXiv:2105.01280*, 2021.
- [45] Petar Veličković. Message passing all the way up, 2022.
- [46] Felix Wu et al. Simplifying graph convolutional networks. In *ICML*, 2019.
- [47] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1025–1035, 2017.
- [48] Hanrui Wang et al. GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.
- [49] Hongyang Gao and Shuiwang Ji. Graph u-nets. In *ICML*, 2019.
- [50] Katsuhiko Ishiguro, Shin-ichi Maeda, and Masanori Koyama. Graph warp module: an auxiliary module for boosting the power of graph neural networks. *arXiv preprint arXiv:1902.01020*, 2019.
- [51] Yifan Xue, Yixuan Liao, Xiaoxin Chen, and Jingwei Zhao. Node augmentation methods for graph neural network based object classification. In *2021 2nd International Conference on Computing and Data Science (CDS)*, pages 556–561. IEEE, 2021.
- [52] Trang Pham, Truyen Tran, Hoa Dam, and Svetha Venkatesh. Graph classification via deep learning with virtual nodes. *arXiv preprint arXiv:1708.04357*, 2017.
- [53] Zheng Ma, Ming Li, and Yuguang Wang. Pan: Path integral based convolution for deep graph neural networks. *arXiv preprint arXiv:1904.10996*, 2019.
- [54] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
- [55] GNN models from Open Graph Benchmark. <https://github.com/snap-stanford/ogb/tree/master/examples/graphproppred/mol>. [Online; accessed 16-Jan-2022].
- [56] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 22118–22133. Curran Associates, Inc., 2020.
- [57] Gregor Kasieczka, Tilman Plehn, Jennifer Thompson, and Michael Russel. Top quark tagging reference dataset, March 2019.
- [58] Javier Duarte and Jean-Roch Vlimant. Graph neural networks for particle tracking and reconstruction. In *Artificial Intelligence for High Energy Physics*, pages 387–436. World Scientific, December 2020.
- [59] Atefeh Sohrabizadeh, Yuze Chi, and Jason Cong. StreamGCN: Accelerating graph convolutional networks with streaming processing. In *2022 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–8, April 2022.
- [60] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.
- [61] PyTorch Geometric. <https://pytorch-geometric.readthedocs.io/en/latest/>. [Online; accessed 16-Jan-2022].
- [62] Groq, Inc. The challenge of batch size 1: Groq adds responsiveness to inference performance. page 7, 2020.