2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-9423-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/HiPC56025.2022.00015

Low-latency Mini-batch GNN Inference on CPU-FPGA Heterogeneous Platform

Bingyi Zhang
University of Southern California
Los Angeles, CA, USA
bingyizh@usc.edu

Hanqing Zeng
Meta AI
Menlo Park, CA, USA
zengh@meta.com

Viktor Prasanna
University of Southern California
Los Angeles, CA, USA
prasanna@usc.edu

***Abstract*—Mini-batch inference of Graph Neural Networks (GNNs) is a key problem in many real-world applications. In this paper, we develop a computationally efficient mapping of GNNs onto CPU-FPGA heterogeneous platforms to achieve low-latency mini-batch inference. While the lightweight preprocessing algorithm of GNNs can be efficiently mapped onto the CPU platform, on the FPGA platform, we design a novel GNN hardware accelerator with an adaptive datapath denoted as Adaptive Computation Kernel (ACK) that can execute various computation kernels of GNNs with low-latency: (1) for dense computation kernels expressed as matrix multiplication, ACK works as a systolic array with fully localized connections, (2) for sparse computation kernels, ACK follows the scatter-gather paradigm and works as multiple parallel pipelines to support the irregular connectivity of graphs. The proposed task scheduling hides the CPU-FPGA data communication overhead to reduce the inference latency. We develop a fast design space exploration algorithm to generate a single accelerator for multiple target GNN models. We implement our accelerator on a state-of-the-art CPU-FPGA platform and evaluate the performance using three representative models (GCN, GraphSAGE, GAT). Results show that our CPU-FPGA implementation achieves $21.4-50.8\times$, $2.9-21.6\times$, $4.7\times$ latency reduction compared with state-of-the-art implementations on CPU-only, CPU-GPU and CPU-FPGA platforms.**

***Index Terms*—Graph Neural Network, Mini-batch inference, FPGA acceleration**

I. Introduction

Graph Neural Networks (GNNs) have become a revolutionary technique in graph-based machine learning. Many vendors have adopted GNNs in their commercial systems, such as recommendation systems [1], social media, knowledge databases, etc. In these systems, data are represented as graphs, where vertices correspond to entities and edges encode the relationship among entities. A fundamental task in these systems is mini-batch GNN inference: *given a batch of target vertices, infer their embeddings (vector representations) with low-latency.* For example, in recommendation systems of Alibaba [1] and Facebook [2], multiple users make a batch of requests at a time and inference latency directly determines the quality of service.

There are two major challenges for mini-batch inference. **(1) neighborhood explosion**: the widely used GNNs [3], [4], [5] follow the message-passing paradigm: in a L-layer model, a vertex recursively aggregates information from its L-hop neighbors. The *receptive field* is defined as the set of neighbors

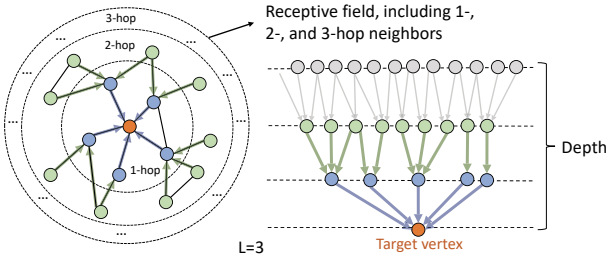

Fig. 1: Recursive message passing of GNNs results in exponential computation and communication cost, and low computation-to-communication (C2C) ratio

passing messages to the target vertex. In the example shown by Figure 1, the receptive field consists of all the vertices within L-hop. In a large graph, the size of the receptive field quickly explodes w.r.t. model depth L. Therefore, mini-batch GNN inference suffers from two issues. First, the computation and communication costs grow exponentially with the depth of GNN. This hinders the deployment of deeper GNNs on memory constrained accelerators. It has been proven [6], [7] that deeper GNNs have higher accuracy than shallower ones. Second, the computation-to-communication (C2C) ratio is low, thus making it not suitable for hardware acceleration. **(2) load imbalance**: GNN computation involves various kinds of kernels, including dense computation kernels and sparse computation kernels. To support various kernels, previous work [8], [9], [10] designed *hybrid* accelerators that for each kernel, a dedicated hardware module is designed and initialized independently. For example, in GraphACT [9], the Feature Aggregation Module executes the sparse kernel and Feature Transformation Module executes the dense kernel. However, it is challenging to achieve load balance among hardware modules in a hybrid accelerator. For example, the workload of feature aggregation is unpredictable and depends on the connectivity of the input graphs. The load imbalance leads to hardware under-utilization and extra latency.

Recently, *model depth-receptive field decoupling* [7] has been proposed to resolve the *neighborhood explosion* challenge. Under the decoupling principle, GNN depth is chosen to be independent of the receptive field. As shown in Figure 2, for a target vertex, it selects a small number of important

2640-0316/22/$31.00 ©2022 IEEE
DOI 10.1109/HiPC56025.2022.00015

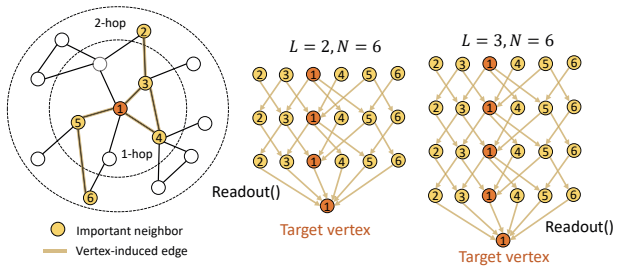

Fig. 2: An example of Decoupled GNN model

neighbors as the receptive field $\mathcal{N}$, and then performs message passing within $\mathcal{N}$. The key property of a Decoupled GNN is that the size of $\mathcal{N}$ remains fixed while the GNN becomes deeper, thus reducing the computation complexity from exponential to linear w.r.t. model depth. Each layer only propagates information within $\mathcal{N}$. Compared with original GNN models (Figure 1), the Decoupled GNN models theoretically lead to significantly less computation cost and memory bandwidth requirement and thus are well-suited for hardware acceleration.

While there have been many GNN accelerators [8], [11], [10], [12], [9] proposed, none of them is designed or optimized for low-latency mini-batch inference. Specifically, [8], [11], [10], [12] are for *full-graph* inference and [9] is for mini-batch *training*. Full-graph GNN inference has very different computation characteristics from mini-batch inference. In full-graph inference, all vertices in the graph are target vertices. Through careful vertex reordering [13], [14] and graph partitioning [8], [10], full-graph execution can achieve high data reuse by exploiting common neighbors. However, in mini-batch inference, it is more challenging to improve data reuse since the target vertices arrive randomly and rarely share common neighbors. In addition, GraphACT [9] is built on a specific training algorithm [15] to improve computation-to-communication ratio *only* during training. While the computation pipeline of GraphACT can be adapted to inference, its performance may be sub-optimal due to load imbalance (challenge (2) above).

In this paper, we accelerate mini-batch inference of Decoupled GNN models. Decoupling leads to high computational efficiency and makes it attractive for acceleration. Computations on the Decoupled model involves identifying important neighbors (Section III-B), and computation intensive GNN kernels (Section IV-A). Identifying important neighbors is lightweight, but requires irregular memory access and complex control flow. This can be efficiently executed on a CPU while the latter can be accelerated by FPGA which offers massive data parallelism. We show that a CPU-FPGA heterogeneous platform can effectively accelerate Decoupled GNNs. On FPGA platform, we design a unified hardware accelerator consisting of (1) adaptive datapath that can execute various computation kernels of GNNs with low-latency, thus overcoming the load-imbalance challenge, (2) memory organization that can hide data communication overhead to further reduce the inference latency. Our main contributions are:

- By analyzing the computation and communication cost of mini-batch GNN inference, we identify "model depth-receptive field" decoupling as a key model design technique towards low-latency accelerator design.
- We propose a system design on CPU-FPGA platforms to accelerate mini-batch inference of Decoupled GNN models:
  - We develop a novel hardware accelerator with both the sparse and dense computation modes to execute various GNN computation kernels with low latency.
  - We customize the memory organization with double/triple buffering techniques on FPGA to reduce data access latency and enable data prefetching.
  - We perform task scheduling to hide the CPU-FPGA data movement overhead.
- We develop a design space exploration algorithm that given 1) a specification of the target FPGA device, and 2) a set of target GNN models with various depths and receptive field sizes, generates a *single* hardware that achieves low-latency inference without reconfiguration.
- We implement our accelerator on a state-of-the-art CPU-FPGA platform and evaluate its performance using three representative models (GCN, GraphSAGE, GAT). We achieve $21.4-50.8\times$, $2.9-21.6\times$, $4.7\times$ latency reduction compared with state-of-the-art implementations on CPU-only, CPU-GPU and CPU-FPGA platforms.

II. BACKGROUND

A. GNN Acceleration on FPGA

Field Programmable Gate Array (FPGA) has been extensively studied for accelerating machine learning tasks [16], [17]. A high-end FPGA device has significant hardware resources, including Lookup Tables (LUTs), on-chip memories (BRAMs, URAMs), etc. Hardware programmability of FPGAs allows users to exploit the fine-grained data parallelism in a computation task. An FPGA is attractive for low-latency computations compared with GPU which is mainly optimized for coarse-grained thread-level parallelism.

GNN accelerators on FPGA: GraphACT [9] proposes a hybrid accelerator on FPGA for sub-graph sampling based GNN training. BoostGCN [10] accelerates the full-graph GNN inference through partition-centric feature aggregation. Deepburning-GL [12] is a design automation framework to generate FPGA accelerators for full-graph GNN inference. AWB-GCN [18] exploits data sparsity in various computation kernels of GNN. As discussed in Section I, previous GNN accelerators on FPGA [9], [10], [12], [18] are not suitable for mini-batch GNN inference. In this work, we develop an optimized FPGA accelerator to achieve low-latency mini-batch GNN inference (Section III).

B. Graph Neural Network

The related notations are defined in Table I. Graph Neural Networks (GNNs) [3], [4] are proposed for representation learning on graphs, facilitating tasks such as node classification [4], [3], link prediction [19] and graph classification [20]. The

TABLE I: Notations

Notation	Description	Notation	Description
$\mathcal{G}(\mathcal{V},\mathcal{E})$	input graph	v_i	i^{th} vertex
$\mathcal{V}$	set of vertices	e_{ij}	edge from v_i to v_j
$\mathcal{E}$	set of edges	L	number of GNN layers
N	# of vertices in the receptive field	$\mathcal{N}_L(i)$	L-hop neighbors of v_i
$\boldsymbol{h}_i^l$	feature vector of v_i at layer l	$\mathcal{N}$	receptive field
$\boldsymbol{H}$	vertex feature matrix	$\boldsymbol{W}^l$	weight matrix of layer l

input to an L-layer GNN is a graph $\mathcal{G}(\mathcal{V},\mathcal{E},\mathcal{X})$ where each vertex $v \in \mathcal{V}$ has a feature vector $\boldsymbol{x}_v \in \mathcal{X}$. The outputs are node representation vectors $\boldsymbol{h}^L$ for each vertex v. As shown in Algorithm 1, in layer l, v's neighbors $\{u|(u,v) \in \mathcal{E}\}$ perform *message passing* to generate the layer output $\boldsymbol{h}_v^l$, where each "message" is the $\boldsymbol{h}_u^{l-1}$ of the previous layer. Thus, performing message passing recursively for L layers means each vertex v aggregates information from all its L-hop neighbors.

Algorithm 1 Recursive message-passing paradigm of GNN

Input: Input graph: $\mathcal{G}(\mathcal{V},\mathcal{E})$; Initial vertex features of input graph: $\{\boldsymbol{h}_1^0, \boldsymbol{h}_2^0, \boldsymbol{h}_3^0, ..., \boldsymbol{h}_{|\mathcal{V}|}^0\}$; Set of target vertices: $\{v_1, v_2, ..., v_m\}$
Output: Output embeddings: $\{\boldsymbol{h}_{v_1}^L, \boldsymbol{h}_{v_2}^L, \boldsymbol{h}_{v_3}^L, ..., \boldsymbol{h}_{v_m}^L\}$
  for $v_i \in \{v_1, v_2, ..., v_m\}$ **do**
    for $l = 1$ to L **do**
      for $v_j \in \mathcal{N}_{L-l}(i)$ **do**
        $\boldsymbol{z}_j^l$ = aggregate $\left(\boldsymbol{h}_k^{l-1} : k \in \mathcal{N}_1(j) \bigcap \mathcal{N}_{L-l+1}(i)\right)$
        $\boldsymbol{h}_j^l$ = update $\left(\boldsymbol{z}_j^l, \boldsymbol{h}_j^{l-1}, \boldsymbol{W}^l\right)$

We define *receptive field* of v as the set of all vertices that pass messages to v. Thus, for an L-layer GNN following Algorithm 1, the receptive field of v includes all vertices which are up to L hops away from v. The size N of the receptive field grows exponentially with the depth L of the model: $N \approx \mathcal{O}(d^L)$, where d is the average degree of the graph. We denote GNNs following such recursive message-passing paradigm as *Coupled models* since the size of receptive field N depends on the model depth L. Note that the size of the GNN model, which equals the total size of all the weight matrices $\{|\boldsymbol{W}^l| : 1 \leqslant l \leqslant L\}$, is independent of the size of the graph.

Specification of a Coupled model: A Coupled GNN model is specified by: (1) number of layers L, (2) aggregate() function that defines operator for aggregating the neighbor information (e.g., aggregate() of GraphSAGE [4]: $\boldsymbol{z}_j^l = \text{Mean}\left(\boldsymbol{h}_i^{l-1} : v_i \in \mathcal{N}_1(j) \cup \{j\}\right)$), (3) hidden dimension of each layer f_l for $0 \leqslant l \leqslant L$, (4) update() function (e.g., $\boldsymbol{h}_j^l = ReLU(\boldsymbol{W}^l \boldsymbol{h}_j^{l-1})$) with learnable matrix $\boldsymbol{W}^l$ $(1 \leqslant l \leqslant L)$.

A main challenge for Coupled GNN models is the low Computation-to-Communication (C2C) ratio. We profile the execution of mini-batch inference of GraphSAGE [4] using a prior FPGA accelerator (GraphACT)[1] [9]. The CPU-FPGA platform and the graphs are specified in Section V-A. The graph is stored in the external memory of the host processor, since the sizes of realistic graphs are often much larger than the on-chip capacity of FPGAs. We further perform vertex sampling on the L-hop neighborhood (following the recommended parameters [4]) to optimize the inference latency. As shown in Figure 3-(1), the data transmission between CPU and FPGA incurs significant execution time overhead because the number of neighbors grows exponentially with the depth of GNN model. The execution time also increases exponentially with the GNN depth. Moreover, the hardware accelerator has low utilization $< 30\%$ (computed by $\frac{\text{achieved performance}}{\text{peak performance}} \times 100\%$). The roofline analysis (Figure 3) demonstrates that mini-batch inference of *Coupled GNN model* is memory-bound, and the overall performance is limited by the available PCIe bandwidth.

[1]GraphACT is an accelerator for training the GraphSAGE model, including forward propagation for inference and back propagation for calculating weight gradients. In Figure 3, we only perform the forward propagation of GraphACT for inference.

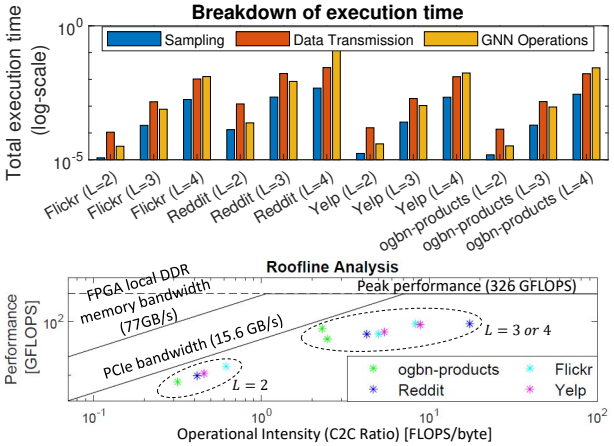

Fig. 3: Experimental analysis of mini-batch inference using Coupled GraphSAGE [3] model: (1) Breakdown of execution time, (2) Roofline analysis (vertical axis is in log-scale)

C. Decoupling of Model Depth and Receptive Field

Algorithm 2 Inference process of Decoupled GNN models

Input: $\mathcal{G}(\mathcal{V},\mathcal{E},\boldsymbol{X}^0)$; Number of layers L; Size of receptive field N; A batch of target vertices $\mathcal{V}_t$; GNN layer operators (aggregate(), update());
Output: Representation vectors of target vertices: $\{\boldsymbol{h}_v^L : v \in \mathcal{V}_t\}$
1: **for** $v \in \mathcal{V}_t$ **do**
2: Identify N important neighbors $\mathcal{N}_{imp}(v)$ for v.
3: Build vertex-induced subgraph $\mathcal{G}'(v)$ using $\mathcal{N}_{imp}(v) \bigcup \{v\}$
4: Extract the input vertex features $\mathcal{F}(v) = \{\boldsymbol{h}_u^0 : u \in \mathcal{G}'(v)\}$.
5: **for** $l \leftarrow 1$ to L **do**
6: Message passing within $\mathcal{G}'(v)$ using the layer-l operators
7: Obtain representation vector of vertex v through Readout().

Recently, [7] proposed a decoupling principle where the GNN depth L and the receptive field size N are specified independently. Decoupling is proposed based on the observation that in the Coupled GNN models, most neighbors involved in message-passing do *not* provide useful information.

Therefore, the key is to identify the important neighbors of the target vertex before applying message passing. As shown in Algorithm 2, for each target vertex v, we first define its receptive field $\mathcal{N}_{\text{imp}}$ as the important neighbors of v, where $\mathcal{N}_{\text{imp}}$ is independent of L. Then, we build a vertex-induced subgraph $\mathcal{G}'(v)$ from $\mathcal{N}_{\text{imp}}(v) \bigcup \{v\}$. Next, the GNN message passing is performed within $\mathcal{G}'(v)$ for L layers using the GNN layer operators. The representative vector $\boldsymbol{h}_v^{\text{emb}}$ is generated via applying the Readout() function (e.g., Max()) to the outputs of the last GNN layer. For example, $\boldsymbol{h}_v^{\text{emb}} = \text{Max}(\{\boldsymbol{h}_u^L : u \in \mathcal{N}_{\text{imp}}(v) \bigcup \{v\}\})$. Figure 2 (See Introduction Section) shows an example. Note that the decoupling principle can be applied to widely used models (e.g., GCN, GraphSAGE, GIN, GAT) since it does not change the GNN layer operators (e.g., aggregate and update). We define the GNNs constructed by the decoupling principle as the *Decoupled models*.

Specification of Decoupled model: A Decoupled model is specified by: (1) number of layers L, (2) number of important neighbors for the target vertex N (i.e., size of the receptive field), (3) the sampling algorithm to obtain the important neighbors, (4) aggregate() function, (5) hidden dimension of each layer f_l, $0 \leqslant l \leqslant L$, (6) update() function with learnable weight matrix $\boldsymbol{W}^l$, $1 \leqslant l \leqslant L$.

Accuracy of Decoupled model: When choosing appropriate neighbors $\mathcal{N}$ (see [7]), a Decoupled model in general achieves higher accuracy than the original Coupled model. See [7] for detailed theoretical and empirical evaluation.

III. Proposed Approach

A. Overview

As shown in Section II-B, Coupled GNN models are inherently memory-bound for mini-batch inference due to the exponential growth of the receptive field, making them hard to be accelerated even with well-optimized hardware pipelines. We identify that Decoupled GNN models are more suitable to be accelerated, due to their high C2C ratio (Section III-B). Therefore, the objective of our hardware design is to achieve low-latency mini-batch inference of Decoupled GNN models. For simplicity, in the rest of the paper, we use mini-batch and batch interchangeably. We define the performance metric as **latency of a batch** (Figure 9): given a batch of C target vertices and a pre-trained Decoupled GNN, **latency** is the time duration from receiving the C target vertex indices to obtaining the vertex representation vectors (See Figure 9). We consider a general scenario in which batches can come in intermittently with variable inter-batch latency, following the Facebook recommendation system [2].

To map Decoupled GNN models on CPU-FPGA platforms, we first identify and characterize the various computation kernels of GNNs (Section IV-A). Then, we design a novel *unified* architecture named Adaptive Computation Kernel (ACK, see Section IV-B), capable of executing both the sparse and dense computation without any runtime reconfiguration. Finally, we propose a design space exploration algorithm (Section IV-E) to generate a single hardware design point for various GNN models. Our design is thus advantageous compared with previous FPGA accelerators (e.g., BoostGCN [10], Deepburning-GL [12] , HP-GNN [18]) which require regenerating a hardware design for each GNN model.

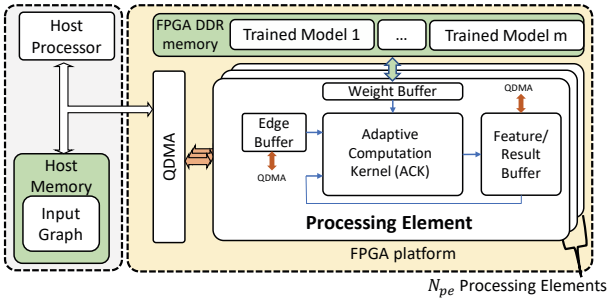

Fig. 4: System design

B. Analysis of Decoupled Models

We compare the computation and communication characteristics of Coupled and Decoupled GNN models. Using an L-layer Coupled GNN model to generate embedding for a target vertex, the information in the L-hop neighborhood is needed. In a Decoupled GNN model, N and L are specified independently. To simplify the analysis, we assume $f_i = f$ $(i = 0, 1, ..., L)$, and illustrate using the GraphSAGE [4] model. The comparison is shown in Table II. Usually, the receptive field of Coupled GNNs $\mathcal{O}(d^L)$ is much larger than that of Decoupled ones. To summarize, *Decoupled models* achieve small computation and communication cost, high C2C ratio and require small on-chip memory, making them attractive for hardware acceleration.

TABLE II: Comparison of Coupled and Decoupled GNNs

	Receptive Field Size	**Comp. Cost**	**Comm. Cost**	**C2C Ratio**
Coupled GNNs	$\mathcal{O}(d^L)$	$\mathcal{O}(d^L f^2)$	$\mathcal{O}(d^L f)$	$\mathcal{O}(f)$
Decoupled GNNs	$N \ll \mathcal{O}(d^L)$	$\mathcal{O}(NLf^2)$	$\mathcal{O}(Nf)$	$\mathcal{O}(Lf)$

Important Neighbor Identification (INI): INI (line 2 of Algorithm 2) is the key to achieve high accuracy with a Decoupled model. Following [7], we use the Personalized PageRank (PPR) [21] score as the metric to indicate the importance of neighbor vertices w.r.t. a given target vertex. We use the local-push algorithm [22] to compute approximate PPR scores. There are several benefits of using this approach: (1) As shown in [7], PPR score is a good metric to reflect neighbor importance. Empirically, Decoupled models based on PPR achieve high accuracy with a small number of neighbor vertices (e.g., $100 - 200$ vertices) [7]. (2) The computation complexity of the local-push algorithm is low and is independent of graph size [23]. (3) The local-push algorithm can be easily parallelized across multiple CPU cores.

C. System Design

Figure 4 depicts the proposed system.

Design Time: At design time, given the specification of the target FPGA platform and a set of Decoupled GNN models (see Section IV-E), we generate a single hardware accelerator and deploy it on the target FPGA platform. The overhead of generating the accelerator is a one-time cost. The trained GNN models are stored in the FPGA DDR memory. User can specify which model to use at runtime.

Algorithm 3 Parallel Mini-batch Inference on CPU-FPGA

Input: A batch of target vertices $\mathcal{V}_t$; A Decoupled GNN model (already trained and stored in FPGA DDR memory);
Output: Representation vectors of target vertices: $\{\boldsymbol{h}_v^L : v \in \mathcal{V}_t\}$

1: **while** there is an idle CPU thread **do** ▷ CPU
2: Pick a target vertex v from $\mathcal{V}_t$ and remove v from $\mathcal{V}_t$
3: Extract important neighbors and build vertex-induced subgraph $\mathcal{G}'(v)$
4: Send vertex features and edges of $\mathcal{G}'(v)$ to FPGA
5: **while** there is an idle PE **do** ▷ FPGA
6: Load vertex features and edges of $\mathcal{G}'(v)$ for a target vertex v
7: **for** $l \leftarrow 1$ to L **do** ▷ Inference using ACK
8: **for** each $kernel$ from the $kernels$ of layer l **do**
9: Configure the execution mode of ACK for $kernel$
10: Execute $kernel$ on ACK
11: Send representation vector $\boldsymbol{h}_v^L$ back to CPU

Runtime: The overall execution process between CPU and FPGA is described in Algorithm 3. The input graph (including the edges and vertex features) is stored in the host memory. During runtime, the host processor receives the indices of a batch of target vertices and the GNN model specified by the user. **On the host platform**, the CPU performs important neighbor identification (line 2 of Algorithm 2) and constructs the vertex-induced subgraph for the target vertices. We use parallel threads on the CPU to execute the local-push algorithm [23] for multiple target vertices concurrently. Then, the CPU extracts the features of input vertices and the edges of the subgraph, and sends them to the FPGA accelerator through the PCIe interconnection. The CPU also performs task allocation for the accelerator based on the specification of the GNN model. For example, for inferring a target vertex using a L-layer model with 2 kernels (e.g., feature aggregation and feature transformation of a GCN [3] layer), the host program allocates $2L$ kernels for the accelerator to execute. **On the FPGA platform**, the input data from PCIe is directly sent to the accelerator through QDMA [24]. The FPGA accelerator consists of N_{pe} multiple parallel and independent processing elements (PEs) where each PE processes one target vertex at a time. Adaptive Computation Kernel (ACK) executes the L layers sequentially (see Algorithm 3). For each layer, ACK executes the kernels sequentially. The ACK execution mode corresponding to a kernel (see Section IV-B) is set by the control bits of the hardware multiplexers in ACK. The overhead of switching execution modes is just one clock cycle.

IV. Hardware Architecture

The proposed FPGA accelerator (Figure 4) consists of N_{pe} parallel and independent processing elements (PEs). Each PE contains an Adaptive Computation Kernel (ACK) to execute various computation kernels in GNNs, an Edge Buffer to store the edges, a Weight Buffer to store the Weight matrices and a Feature/Result Buffer to store the vertex features. An ACK contains a 2-D mesh of ALUs (Section IV-B).

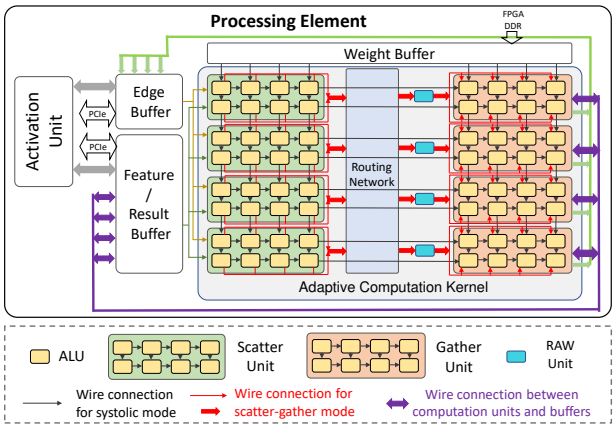

Fig. 5: The details of a Processing Element

A. Computation Kernels of GNNs

We summarize the various computation kernels in four widely used GNN models: GCN [3], GraphSAGE [4], GIN [25], GAT [5]:

Feature Aggregation (FA): FA can be implemented by a Scatter phase followed by a Gather phase. In Scatter phase, each vertex v_i sends its features $\boldsymbol{h}_i$ to its neighbors in the vertex-induced subgraph. The vertex features are multiplied by the edge weight to generate the intermediate results. In Gather phase, each vertex aggregates the incoming intermediate results through aggregate() function (e.g., element-wise Max, Mean) to generate the aggregated features $\boldsymbol{z}_i$.

Feature Transformation (FT): The aggregated features $\boldsymbol{z}_i$ are transformed through the update() function. In the widely used GNN models (e.g. GCN, GraphSAGE, GIN, GAT), the update() is a single-layer MLP with an element-wise activation function (e.g., ReLU, LeakyReLU).

Attention: Some GNN models (e.g., GAT) exploit the Attention mechanism to generate data-dependent edge weights. The weight of edge e_{ij} is calculated based on ($\boldsymbol{h}_i$, $\boldsymbol{h}_j$, $\boldsymbol{W}_{\text{att}}$, $\boldsymbol{a}$). $\boldsymbol{W}_{\text{att}}$ is the attention weight matrix that is multiplied with $\boldsymbol{h}_i$ and $\boldsymbol{h}_j$. $\boldsymbol{a}$ is the vector that is multiplied with $\boldsymbol{W}_{\text{att}}\boldsymbol{h}_i || \boldsymbol{W}_{\text{att}}\boldsymbol{h}_j$ to get the edge weight e_{ij}.

FT and Attention are dense computation kernels involving dense matrix multiplication, while FA is a sparse computation kernel due to the sparsity and irregularity of the graphs. If we execute the different kernels using different hardware modules, the load imbalance can lead to hardware under-utilization and increased latency (See Section IV-C).

B. Hardware Modules

To address the load imbalance challenge, we propose Adaptive Computation Kernel to execute various computation kernels of GNNs using the same set of computation resources.

Adaptive Computation Kernel (ACK): ACK contains an array of Arithmetic Logical Units (ALUs) of size $p_{\text{sys}} \times p_{\text{sys}}$. An ALU can execute various arithmetic operations including Multiplication, Addition, Multiply-Accumulation, Min, Max, etc. The proposed ACK has two execution modes – *Systolic Mode* and *Scatter-Gather Mode* – that can support FA, FT and Attention (Section IV-A).

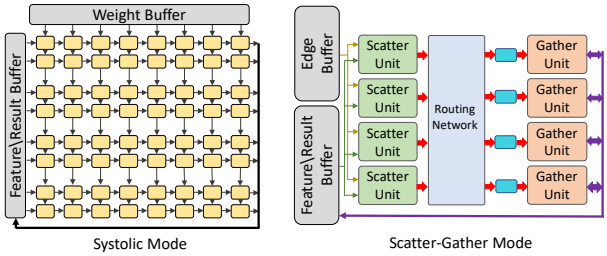

Fig. 6: Two data paths realizing the dense (left) and sparse (right) execution modes of ACK

Systolic Mode: The array of ALUs are organized as a two-dimension systolic array. Systolic array is an efficient architecture for dense matrix multiplication, which has localized interconnections as shown in Figure 6. Systolic Mode supports dense matrix multiplication in FT and Attention. In Systolic Mode, ACK can execute the multiplication of weight matrix $\boldsymbol{W}$ and Feature matrix $\boldsymbol{H}_{\text{in}}$ (See Table I, each row of $\boldsymbol{H}_{\text{in}}$ is a vertex feature vector $\boldsymbol{h}_i$) to obtain the output feature matrix $\boldsymbol{H}_{\text{out}}$. Weight Buffer streams the weight matrices of MLP (FT) or the attention weight matrix to the systolic array, and Feature/Result Buffer streams multiple vertex feature vectors into the systolic array. Systolic array of size $p_{\text{sys}} \times p_{\text{sys}}$ can execute p_{sys}^2 Multiply-Accumulation operations per cycle. Both the Weight Buffer and the Feature Buffer have port width of p_{sys} data, and can send p_{sys} data to the systolic array per cycle.

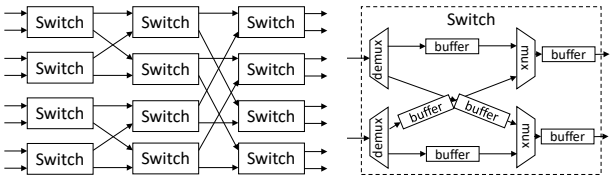

Fig. 7: The architecture of the routing network

Scatter-Gather Mode: The PE executes feature aggregation (FA) following the Scatter-Gather paradigm (Algorithm 4). The array of ALUs is partitioned into equal number of Scatter Units and Gather Units. In each Scatter Unit, the ALUs are organized as a vector multiplier that multiplies the vertex feature vector by the scalar edge weight. In each Gather Unit, the ALUs execute the aggregate() function. Suppose the feature vector has the format $\langle src, features \rangle$, where src denotes the index of the source vertex and the $features$ is the feature vector of the source vertex. Edge has the format $\langle src, dst, weight \rangle$, where $src, dst, weight$ denote the source vertex index, destination vertex index, edge weight respectively. The generated intermediate results (updates) by the Scatter Units have the format $\langle dst, features \rangle$. The N vertices in the receptive field are equally partitioned to the Gather Units. The routing network (Shown in Figure 7) performs all-to-all interconnection between Scatter Units and Gather Units. It routes the intermediate results $\langle dst, features \rangle$ generated by Scatter Units to the corresponding Gather Units based on the index dst. For example, suppose a Gather Unit is responsible for accumulating the results for vertices $v_1 - v_{64}$. All intermediate results that have dst ranging from 1 to 64 will be routed to this Gather Unit. This routing network is implemented as a butterfly network [26] which has close-to-optimal routing throughput for all-to-all communication (See Table 4 of [26] for the detailed evaluation of routing network).

Algorithm 4 FA using Scatter-Gather Paradigm

while not done **do**
  for each edge $e\langle src, dst, weight \rangle$ **do** ▷ Scatter Unit
    Produce update $u \leftarrow$Scatter($src.features, e.weight$)
  for each update $u\langle dst, features \rangle$ **do** ▷ Gather Unit
    Update vertex $dst \leftarrow$ Gather($u.features$)

When the execution of a kernel is completely finished, the ACK can start to execute the next kernel. In our design, the number of Scatter units and Gather Units both equals p_{sg}, where p_{sg} is decided by $p_{\text{sg}} = p_{\text{sys}}/2$. The Feature/Result Buffer have p_{sg} banks. Each bank stores the feature vectors of a partition of the vertex-induced subgraph (Algorithm 2). Each bank is connected to a Gather Unit. Each Scatter or Gather Unit has $2p_{\text{sg}}$ ALUs. Note that read-after-write (RAW) data hazard may occur when accumulators in the Gather Unit read the old feature vertex vector from the Feature/Result Buffer. To resolve the RAW data hazard, we implement a RAW Unit before Gather Unit as shown in Figure 8. In the RAW Unit, there is a RAW detector to detect the RAW data hazard and a small Reorder Buffer (implemented as a FIFO) to cache the input data when RAW is detected. The data in the Reorder Buffer will be sent to Gather Unit when there is no RAW data hazard.

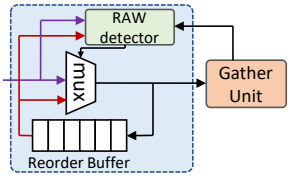

Fig. 8: RAW Unit (left)

Activation Unit: The Activation Unit executes the element-wise activation function in FT and the Softmax function in Attention. These functions are implemented using Xilinx High-Level Synthesis (HLS). For example, the Softmax function is implemented by using `hls::exp(x)` function as the building block.

Double/triple buffering: In a Processing Element, there are three Feature/Result Buffers for triple buffering. The first Buffer stores the vertex feature vectors of the current GNN layer. The second Buffer stores the vertex feature vectors of the next GNN layer. The third Buffer is used for prefetching

the input vertex feature vectors of the next target vertex. Similarly, Edge Buffer is also designed with triple buffering. Weight buffer is implemented using double buffering, where one buffer is used for storing the weight matrix of the current layer, and the other buffer is used to prefetch and store the weight matrix of the next layer. Through double/triple buffering, memory access and computation are overlapped to reduce the overall inference latency.

C. Load Balance

The key benefit of our design is that we use the same set of computation resources in a single hardware module (ACK) to execute various computation kernels with high efficiency. Therefore, we are able to assign all the on-chip computation resources to ACKs. In contrast, in the hybrid accelerators [8], [9], [10], the computation resources are divided among different hardware modules to execute different computation kernels. Suppose for a single GCN layer, feature aggregatgion (FA) has workload $\alpha_1 > 0$ and feature transformation (FT) has workload $\alpha_2 > 0$, and the total computation resource is β. In our design, we use β for ACKs. Therefore, the latency for executing this single GCN layer of our design is: $\frac{\alpha_1+\alpha_2}{\beta}$. In the hybrid accelerator, suppose the hardware module for FA uses β_1 resources and the hardware module for FT uses $\beta - \beta_1$ resources. The latency for executing this single GCN layer is $\max\left(\frac{\alpha_1}{\beta_1}, \frac{\alpha_2}{\beta-\beta_1}\right)$. It can be proved that:

$$\frac{\alpha_1+\alpha_2}{\beta} \leqslant \max\left(\frac{\alpha_1}{\beta_1}, \frac{\alpha_2}{\beta-\beta_1}\right) \quad (\beta>0, \beta_1>0, \beta-\beta_1>0) \tag{1}$$

where the equality is achieved when $\frac{\alpha_1}{\beta_1} = \frac{\alpha_2}{\beta-\beta_1}$. In the Decoupled GNN model, the workload of FA, α_1 is usually unpredictable because the number of edges in the receptive field varies with the target vertex as well as with the connectivity of the input graph. Moreover, varying the receptive field size can vary the workload α_1, α_2 at different rate. Therefore, in a fixed hybrid accelerator, it is hard to keep load balance for various input graphs and Decoupled models with various receptive field sizes. The load imbalance incurs increased latency. To execute GNN models with more than two computation kernels (e.g., GAT), load imbalance can be more severe in hybrid accelerators.

D. Task Scheduling on CPU-FPGA

Figure 9 shows the proposed task scheduling for mini-batch inference, based on Algorithm 3. The host processor performs Important Neighbor Identification and builds a vertex-induced subgraph for each target vertex. If there is an idle PE, it loads the input vertex feature vectors of the vertex-induced subgraph of a target vertex. The PE also prefetches the input data for an unprocessed target vertex. After loading the input data, the PE executes the L-layer GNN forward propagation for the target vertex. Finally, the PE sends the representation vector of the target vertex back to the host processor.

CPU-FPGA data communication: Using the proposed scheduling, the execution of the accelerator and the CPU-FPGA data movement are overlapped for all but the first vertex

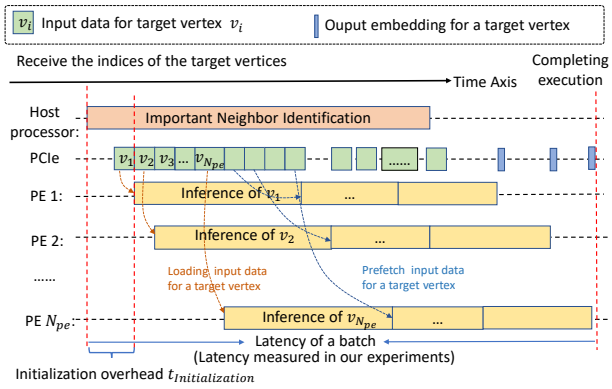

Fig. 9: Task scheduling for mini-batch GNN inference on CPU-FPGA platform

in a batch. Denote $t_{\text{initialization}} = t_{\text{load}} + t_{\text{INI}}$ as the initialization overhead of a batch, where t_{INI} is the latency of runing INI for a vertex using a single CPU thread on the host processor. t_{load} is the latency of loading the induced subgraph (vertex features; edges) for a target vertex. Section V-C shows that $t_{\text{initialization}}$ is negligible compared with total inference latency.

E. Design Space Exploration

We perform design space exploration (DSE) to determine the hardware parameters. The inputs to our DSE are (1) available hardware resources (N_{DSP}: number of DSPs) on FPGA, (2) arithmetic operations in the given set of Decoupled GNN models that needs to be supported. Given the inputs, the DSE determines the number of DSPs in an ALU N_{ALU}, the size of ACK in a PE $p_{\text{sys}} \times p_{\text{sys}}$, the number of PEs N_{pe} in the accelerator. The proposed design has the following properties:

- The proposed accelerator can execute a GNN model as long as the ALU can support all the arithmetic operations in this GNN model. N_{ALU} is determined based on the arithmetic operations of a given Decoupled GNN model.
- The size of the ACK $p_{\text{sys}} \times p_{\text{sys}}$ in a PE determines the latency of inferring a single target vertex, and the number of PEs N_{pe} decides how many target vertices can be inferred concurrently. Thus, the total on-chip computation resources should be exhausted by $N_{\text{pe}} \cdot p_{\text{sys}}^2$. The value of N_{pe} depends on the batch size: for large batch sizes, both large and small N_{pe} work well since sufficient parallelism is available across target vertices; for small batch sizes, it is desirable to set N_{pe} as small in order to still achieve low latency. Since batch sizes vary significantly in real-world applications, we minimizes N_{pe} by maximizing $p_{\text{sys}} \times p_{\text{sys}}$ in a PE.
- To efficiently implement Scatter Unit, Gather Unit and routing network, p_{sys} is chosen to be power of 2.

The above analysis leads to the following DSE algorithm:

1) Determine N_{ALU} based on all the arithmetic operations (given GNN models) to be supported.
2) Maximize the ALU array size: $p_{\text{sys}} = 2^{\left\lfloor \log_2 \sqrt{N_{\text{DSP}}/N_{\text{ALU}}} \right\rfloor}$

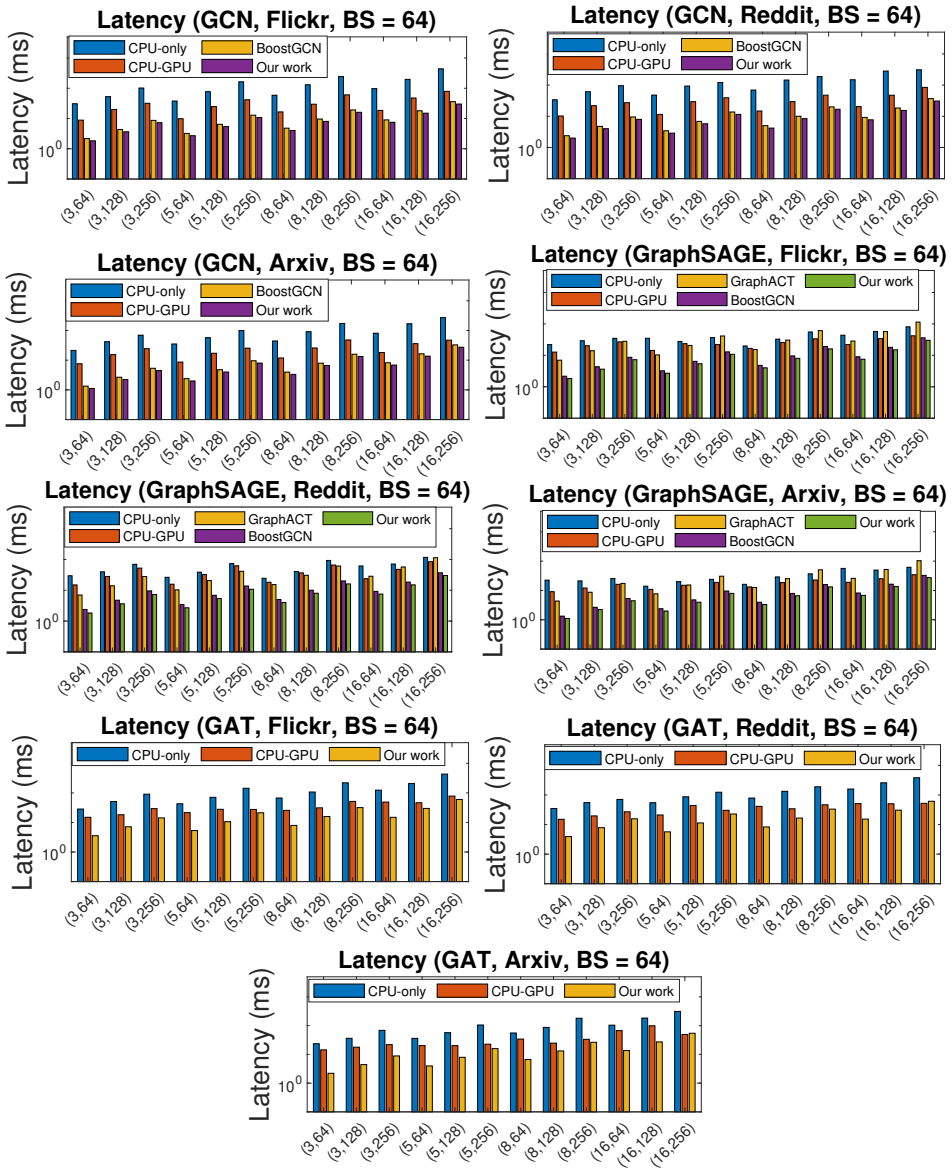

Fig. 10: Comparison of inference latency (Batch Size=64) for Decoupled GNN models with various depth and receptive field. Y-axis is in log-scale. X-axis denotes (number of layers L, size of receptive field N)

3) Determine the number of PEs: $N_{\text{pe}} = \left\lfloor \frac{N_{\text{DSP}}/N_{\text{ALU}}}{p_{\text{sys}} \times p_{\text{sys}}} \right\rfloor$

Many modern FPGAs have multiple Super Logic Regions (SLRs) with limited interconnection among SLRs. We perform the proposed DSE algorithm separately for each SLR. Note that the routing network has $p_{\text{sys}}/2$ input ports and $p_{\text{sys}}/2$ output ports with $(32 \times p_{\text{sys}})$-bit data width (since we use 32-bit data format). Its hardware cost is $\mathcal{O}(p_{\text{sys}}^2 \log p_{\text{sys}})$ which also increases with p_{sys}. Step 2 of maximizing p_{sys} in our DSE may incur additional hardware overhead due to expanding the routing network to be power of 2. Fortunately, as shown in [26], even a large-scale 512-bit 32-input-32-output routing network only consumes less than $189K$ LUTs, which is far smaller ($< 18\%$) than the total LUTs of state-of-the-art FPGA boards. Since all computation is performed with ALU, as long as LUT consumption is $< 100\%$ the latency will not be affected. In the large-scale FPGA device, such as Alveo U250, p_{sys} does not exceed 16. Therefore, the routing network is not the resource bottleneck in our design.

V. Experiments

A. Hardware Details and Baseline Platforms

We use High-level Synthesis (HLS) to develop the hardware templates. The obtained hardware parameters through DSE are annotated into the developed hardware templates, and we use the vendor's tool to synthesize the hardware design

and generate the accelerator bitstream. Then, the bitstream is deployed on the target FPGA platform. We perform DSE to generate a hardware accelerator on a state-of-the-art FPGA platform (Xilinx Alveo U250) for three widely used GNN models (GCN, GraphSAGE, GAT). The FPGA is hosted by an Intel Xeon Gold 5120 CPU. Figure 11 depicts the generated system design (Figure 4) on the CPU-FPGA platform. Based on the arithmetic operation in the three GNN models, each ALU consumes 5 DSPs. The ACK in each PE has an ALU array of size 16×16. In the ACK, there are 8 Scatter Units and 8 Gather Units. Each of the Scatter Units and Gather Units has an ALU array of size 2×8. The routing network is a butterfly network of 8 input ports and 8 output ports. Each input/output port has 512-bit width. On Alveo U250, there are 8 PEs in four Super Logic Regions (SLRs) with each SLR having 2 PEs. The hardware synthesis and Place&Route (P&R) are performed using Vitis 2021.1. The above accelerator on Alveo U250 consumes 762K LUTs, 10854 DSPs, 1853 BRAMs and 1050 URAMs. The resource utilization is reported after P&R. On the host processor, we use 8 threads to execute important neighbor identification. We deploy the host program on the host processor (Intel Xeon Gold 5120 CPU) and accelerator bitstream on the FPGA (Xilinx Alveo U250). The host processor and FPGA are connected through the PCIe 3.0×16 which form our target CPU-FPGA platform.

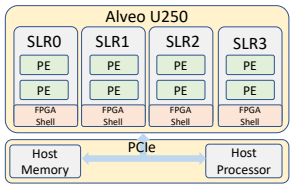

Fig. 11: Proposed CPU-FPGA implementation with the accelerator design on Xilinx Alveo U250

TABLE III: Specifications of platforms

Platforms	CPU AMD Ryzen 3990x	GPU Nvidia RTX3090	FPGA Alveo U250
Technology	TSMC 7 nm	TSMC 7 nm	TSMC 16 nm
Frequency	2.90 GHz	1.7 GHz	300 MHz
Peak Performance	3.7 TFLOPS	36 TFLOPS	0.72 TFLOPS
On-chip Memory	256 MB L3 cache	6 MB L2 cache	54 MB
Memory Bandwidth	107 GB/s	15.6 GB/s (PCIe)	15.6 GB/s (PCIe)

Baseline Platforms: We compare the following platforms in our experiments: (1) **Baseline 1**: CPU-only platform (AMD Ryzen 3990x), (2) **Baseline 2**: CPU-GPU platform (Intel Xeon Gold 5120 CPU + Nvidia RTX3090), (3) **Baseline 3**: CPU-GraphACT (Intel Xeon Gold 5120 CPU + GraphACT [9]), (4) **Baseline 4**: CPU-BoostGCN (Intel Xeon Gold 5120 CPU + BoostGCN [10]), (5) **Our work**: CPU-FPGA (Intel Xeon Gold 5120 CPU + proposed accelerator). The specifications of various platforms are shown in Table III and Table IV. To execute mini-batch inference, the CPU-only platform uses Pytorch with Intel MKL as the backend and the CPU-GPU plaform uses the Pytoch library with CUDA as the backend. Using PyTorch dataloader, the baseline CPU-GPU platform exploits data prefetching and double buffering to overlap loading data from host to GPU global memory and computations in the GPU streaming processor. Note that GraphACT supports GraphSAGE only. BoostGCN can support GCN and GraphSAGE. However, BoostGCN needs to generate a separate FPGA bitstream for each GNN model.

Latency measurement: In our experiments, we measure the *latency of a batch* defined in Section III-A and Figure 9. For all the baselines and our work, the measured *latency of a batch* is the duration from the time when host processor start receiving the indices of a batch of target vertices to the time the inference for all the vertices in the batch has been completed and stored in the CPU. The overheads of Important Neighbor Identification and the data movement between the CPU and GPU/FPGA through PCIe are included in our measured latency.

TABLE IV: Platform specifications of GNN accelerators

	GraphACT [9]	This paper	BoostGCN [10]
Platform	Xilinx Alveo U200	Xilinx Alveo U250	Intel Stratix 10 GX
Frequency	300 MHz	300 MHz	250 MHz
Data format	Float32	Float32	Float32
Peak Performance	249.6 GFLOPS	614 GFLOPS	640 GFLOPS
On-chip Memory	35.8 MB	45 MB	32 MB
Memory Bandwidth	15.6 GB/s (PCIe)	15.6 GB/s (PCIe)	15.6 GB/s (PCIe)

Benchmark: We evaluate various Decoupled Models (GCN, GraphSAGE, GAT) that can achieve superior accuracy. As shown in [7], the Decoupled Models ($N < 200$ and $L = 3$ or 5) can already achieve higher accuracy than the original Coupled GNN models (GCN, GraphSAGE, GAT). The Decoupled Models can achieve higher accuracy when L is increased. To evaluate Decoupled models with various L and N, we set the hidden dimension of each GNN layer as $f_l = 256, (1 \leqslant l \leqslant L)$ following [7]. We set the number of layers L as 3, 5, 8, 16 respectively. We specify the size of the receptive field N as 64, 128, 256. As shown in [2], the producation-scale recommendation systems in Facebook typically use batch size 64, 128, 256. We evaluate our design using a wider range of batch sizes 32, 64, 128, 256, 512. We use three representative graph datasets for evaluation as listed in Table V.

B. Comparison with State-of-the-art

We show the comparison results (latency of a batch) using various GNN models, L and N in Figure 10. Our CPU-FPGA implementation achieves $21.4 - 50.8\times$, $2.9 - 21.6\times$, $4.7\times$, $1.2\times$ speedup compared with CPU-only, CPU-GPU, CPU-GraphACT, and CPU-BoostGCN, respectively. Note that BoostGCN does not support GAT and needs to generate an accelerator for each GNN model.

On the CPU-only platform, the processor can directly (without PCIe overhead) access data from the host memory and

TABLE V: Dataset Statistics

Dataset	Vertices	Edges	Features f_{in}	Classes	Degree
Flickr (FL) [15]	89,250	899,756	500	7	10
Reddit (RE) [4]	232,965	116,069,191	602	41	50
ogbn-arxiv (OA) [27]	169,343	1,166,243	128	7	40

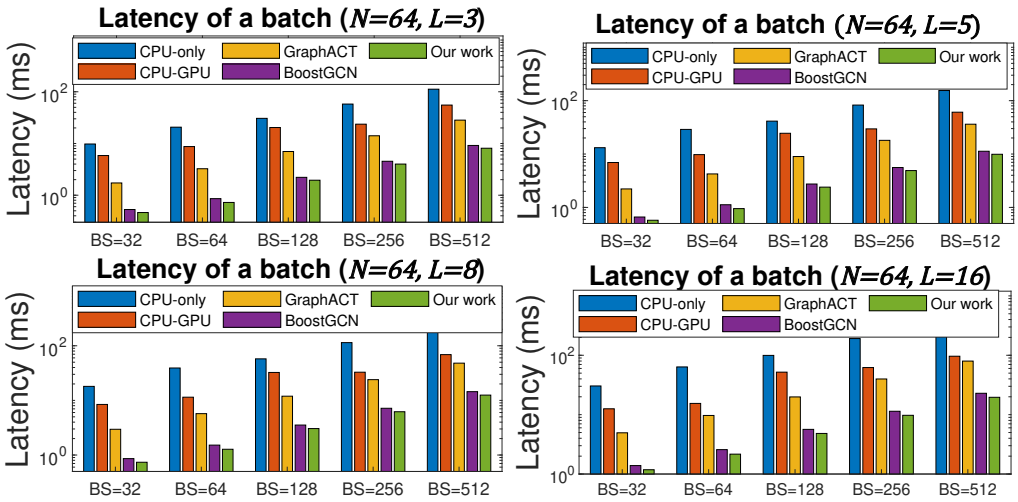

Fig. 12: Latency under various Batch Sizes (BS) for GraphSAGE and Flickr dataset

the processor has large shared L3 cache. However, the feature aggregation of GNN results in irregular memory access pattern and low data reuse. The processor has limited L1 (32 KB) and L2 (512 KB) cache. The data exchange (vertex features;weight matrices;edges) among L1 cache, L2 cache, and L3 cache becomes the performance bottleneck and results in reduced sustained performance. For example, on multi-core platform, loading data from L3 cache incurs latency of 32ns and loading data from L2 cache incurs latency of $5-12$ns. Compared with the CPU, the ACK in our accelerator can access data in one clock cycle during the inference execution.

For the CPU-GPU platform, although the GPU has higher peak performance, the GPU has higher latency than our CPU-FPGA platform because: (1) GPU has extra latency of loading data from host memory to GPU global memory and loading data from GPU global memory to GPU on-chip memory, while in our CPU-FPGA implementation, the FPGA accelerator can directly load data to the on-chip memory through QDMA from the host memory. (2) Similar to CPU, GPU has limited private L1 cache size (32 KB), therefore data exchange (vertex features;weight matrices;edges) between L2 cache and L1 cache becomes the performance bottleneck.

We compare our CPU-FPGA implementation with GraphACT (Baseline 3) and BoostGCN (Baseline 4). GraphACT is optimized for subgraph-based mini-batch training which has similar computation pattern as the mini-batch inference of Decoupled GNN models. BoostGCN is the state-of-the-art FPGA accelerator for full-graph inference. Compared with CPU-GraphACT and CPU-BoostGCN, our CPU-FPGA implementation achieves lower latency because (1) our proposed ACK can efficiently execute various kernels in GNN. GraphACT and BoostGCN follow the hybrid design that two hardware modules are initialized for feature aggregation and feature transformation, respectively. The load imbalance of the two modules leads to hardware under-utilization on GraphACT and BoostGCN. (2) We adopt the Scatter-Gather paradigm to achieve massive computation parallelism for feature aggregation. GraphACT has limited computation parallelism in its Feature Aggregation Module.

Latency for various batch sizes: We compare the mini-batch inference latency with other platforms under various batch sizes. Figure 12 shows the experimental results using the Flickr dataset for the GraphSAGE model. We only show these results due to space limitation. The results under other experimental settings is similar. Under various batch sizes, our CPU-FPGA implementations still achieve significantly lower latency than the CPU-only platform, CPU-GPU platform, CPU-GraphACT and CPU-BoostGCN.

C. Analysis of Execution Time

We perform a detailed analysis of the total execution time using the Xilinx Runtime (XRT) profiler to analyze the execution time of host program, CPU-FPGA data transfer, and the execution time of the computation kernels on the FPGA.

Initialization overhead $t_{\textbf{initialization}}$: We measure the initialization overhead in our task scheduling (Figure 9). We use the results in Figure 13 for illustration since other experimental settings have similar results. The initialization overhead is $0.5\% - 6\%$ of the total execution time, which is negligible.

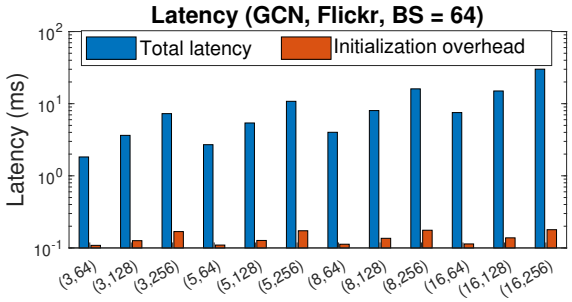

Fig. 13: Comparison of the initialization overhead and the total inference latency under various settings

TABLE VI: Average latency of loading the input data for a target vertex through PCIe interconnection

	Flickr	obgn-arxiv	Reddit
$N = 64$	12.6 μs	3.5 μs	15.1 μs
$N = 128$	29.1 μs	7.7 μs	32.3 μs
$N = 256$	72.5 μs	17.1 μs	72.7 μs

Overhead of CPU-FPGA data communication: To perform inference on a target vertex, the feature vectors of its N important neighbors and the edges in the induced subgraph are loaded from the host memory to the on-chip memory of a PE through PCIe. Note that the input data are directly sent to the on-chip memory through the QDMA. As shown in Table VI, we measure the average latency for loading the input data for a target vertex through PCIe. The above latency is hidden by our task scheduling for most target vertices (See Figure 9).

TABLE VII: Overhead of INI (t_{INI})

	Flickr	ogbn-arxiv	Reddit
Time per vertex (μs)	96	37.6	87.1

Overhead of INI (t_{INI}): On the host platform, we use 8 threads to execute INI. The measured overhead of INI t_{INI} is shown in Table VII. Note that the measured overhead t_{INI} is the time of INI for a vertex using single CPU thread on the host processor. The host processor can execute INI for 8 vertices concurrently. The average latency of INI is negligible compared with the total latency of mini-batch inference ($2 - 100$ ms). Moreover, The overhead t_{INI} for most vertices is hidden by our task scheduling (See Figure 9).

VI. Conclusion

In this paper, we proposed a novel hardware accelerator design to achieve low-latency mini-batch inference on CPU-FPGA heterogeneous platform. On various GNN models, we achieved load-balance and high hardware utilization via the novel Adaptive Computation Kernel design. As a result, our CPU-FPGA implementation achieves significant latency reduction under various GNN models and batch sizes, compared with state-of-the-art CPU, CPU-GPU and CPU-FPGA implementations.

Acknowledgment

This work is supported by the National Science Foundation (NSF) under grants CCF-1919289 and OAC-2209563.

References

[1] H. Yang, "Aligraph: A comprehensive graph neural network platform," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 3165–3166.

[2] U. Gupta, C.-J. Wu, X. Wang, M. Naumov, and B. Reagen, "The architectural implications of facebook's dnn-based personalized recommendation," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 488–501.

[3] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[4] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*.

[5] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," 2017.

[6] G. Li, M. Müller, G. Qian, and B. Perez, "Deepgcns: Making gcns go as deep as cnns," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2021.

[7] H. Zeng, M. Zhang, Y. Xia, A. Srivastava, R. Kannan, V. Prasanna, L. Jin, and R. Chen, "Decoupling the depth and scope of graph neural networks," *Advances in Neural Information Processing Systems*, 2021.

[8] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[9] H. Zeng and V. Prasanna, "Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms," in *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020.

[10] B. Zhang, R. Kannan, and V. Prasanna, "Boostgcn: A framework for optimizing gcn inference on fpga," in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2021, pp. 29–39.

[11] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2020.

[12] S. Liang, C. Liu, Y. Wang, H. Li, and X. Li, "Deepburning-gl: an automated framework for generating graph neural network accelerators," in *2020 ICCAD*. IEEE, 2020, pp. 1–9.

[13] B. Zhang, H. Zeng, and V. Prasanna, "Hardware acceleration of large scale gcn inference," in *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*.

[14] T. Geng, C. Wu, Y. Zhang, H. You, M. Herbordt, Y. Lin, and A. Li, "I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *MICRO-54*, 2021.

[15] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *International Conference on Learning Representations*, 2020.

[16] Z. Choudhury, S. Shrivastava, L. Ramapantulu, and S. Purini, "An fpga overlay for cnn inference with fine-grained flexible parallelism," *ACM Transactions on Architecture and Code Optimization (TACO)*.

[17] A. Sateesan, S. Sinha, and K. Smitha, "Dash: Design automation for synthesis and hardware generation for cnn," in *2020 International Conference on Field-Programmable Technology (ICFPT)*. IEEE, 2020.

[18] Y.-C. Lin, B. Zhang, and V. Prasanna, "Hp-gnn: Generating high throughput gnn training implementation on cpu-fpga heterogeneous platform," *arXiv preprint arXiv:2112.11684*, 2021.

[19] M. Zhang and Y. Chen, "Link prediction based on graph neural networks," *Advances in Neural Information Processing Systems*, 2018.

[20] R. Ying, J. You, C. Morris, X. Ren, W. L. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," *arXiv preprint arXiv:1806.08804*, 2018.

[21] B. Bahmani, A. Chowdhury, and A. Goel, "Fast incremental and personalized pagerank," *arXiv preprint arXiv:1006.2880*, 2010.

[22] R. Andersen, F. Chung, and K. Lang, "Local graph partitioning using pagerank vectors," in *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*. IEEE, 2006, pp. 475–486.

[23] M. Aggarwal, B. Zhang, and V. Prasanna, "Performance of local push algorithms for personalized pagerank on multi-core platforms," in *HiPC 2021*. IEEE, 2021, pp. 370–375.

[24] "Xilinx qdma ip." [Online]. Available: https://www.xilinx.com/products/intellectual-property/pcie-qdma.html

[25] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[26] Y.-k. Choi, Y. Chi, W. Qiao, N. Samardzic, and J. Cong, "Hbm connect: High-performance hls interconnect for fpga hbm," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[27] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," *arXiv preprint arXiv:2005.00687*, 2020.