HitGNN: High-Throughput GNN Training Framework on CPU+Multi-FPGA Heterogeneous Platform

Yi-Chien Lin, *Graduate Student Member, IEEE*, Bingyi Zhang, and Viktor K. Prasanna

***Abstract*—As the size of real-world graphs increases, training Graph Neural Networks (GNNs) has become time-consuming and requires acceleration. While previous works have demonstrated the potential of utilizing FPGA for accelerating GNN training, few works have been carried out to accelerate GNN training with multiple FPGAs due to the necessity of hardware expertise and substantial development effort. To this end, we propose HitGNN, a framework that enables users to effortlessly map GNN training workloads onto a CPU+Multi-FPGA platform for acceleration. In particular, HitGNN takes the user-defined synchronous GNN training algorithm, GNN model, and platform metadata as input, determines the design parameters based on the platform metadata, and performs hardware mapping onto the CPU+Multi-FPGA platform, automatically. HitGNN consists of the following building blocks: (1) high-level application programming interfaces (APIs) that allow users to specify various synchronous GNN training algorithms and GNN models with only a handful of lines of code; (2) a software generator that generates a host program that performs mini-batch sampling, manages CPU-FPGA communication, and handles workload balancing among the FPGAs; (3) an accelerator generator that generates GNN kernels with optimized datapath and memory organization. We show that existing synchronous GNN training algorithms such as DistDGL and PaGraph can be easily deployed on a CPU+Multi-FPGA platform using our framework, while achieving high training throughput. Compared with the state-of-the-art frameworks that accelerate synchronous GNN training on a multi-GPU platform, HitGNN achieves up to 27.21× bandwidth efficiency, and up to 4.26× speedup using much less compute power and memory bandwidth than GPUs. In addition, HitGNN demonstrates good scalability to 16 FPGAs on a CPU+Multi-FPGA platform.**

***Index Terms*—CPU+Multi-FPGA, graph neural network, hardware acceleration.**

I. Introduction

GRAPH Neural Networks (GNNs) have become the state-of-the-art models for representation learning on graphs, facilitating many applications such as social recommendation system [1], [2], molecular property prediction [3], [4] and traffic prediction [5], etc. Initially, GNNs were computed on a CPU/GPU [6], [7], [8], [9] or an FPGA platform [10], [11], [12], [13]; however, as the size of the graph increases, computing GNNs on a single GPU or an FPGA platform becomes time-consuming. Thus, many works [14], [15], [16], [17] have proposed to accelerate GNN training on a multi-CPU or a multi-GPU platform as it provides more memory bandwidth and computation resources. To train GNN on multiple devices in parallel, these works perform *synchronous GNN training*; we describe synchronous GNN training in detail in Section II-C.

Compared with general-purpose processors like CPU and GPU, CPU+FPGA heterogeneous platform is promising for GNN training acceleration: the CPU can flexibly support various graph preprocessing and mini-batch sampling algorithms; and the FPGA can efficiently perform GNN operations because FPGA supports customized data access patterns and memory organization, which can effectively reduce the substantial memory traffic and random memory accesses in GNN training. Despite the various optimizations that can be deployed on a CPU+FPGA platform, training GNNs on a CPU+FPGA platform can still be time-consuming due to limited computation power and memory resources; thus, it is desirable to accelerate GNN training on a CPU+Multi-FPGA heterogeneous platform. However, accelerating GNN training on a CPU+Multi-FPGA platform is challenging. First, training GNNs on a CPU+Multi-FPGA platform suffers from workload imbalance and significant data communication overhead among FPGAs, which leads to low performance and poor scalability. Second, it requires hardware expertise to develop optimized kernels and considerable amount of time to explore the complex hardware design space of a CPU+Multi-FPGA platform.

Motivated by these challenges, we propose HitGNN, a generic framework for mapping synchronous GNN training algorithms on a CPU+Multi-FPGA heterogeneous platform. We first formulate the high-level abstraction of synchronous GNN training algorithms and then develop HitGNN based on the abstraction; this allows our framework to support various training algorithms that can be described with the formulated abstraction. To achieve high throughput and automate the implementation process, we develop a hardware Design Space Exploration (DSE) engine.

Given the platform metadata, the DSE engine determines the accelerator configurations that optimize the training throughput.

Manuscript received 2 March 2023; revised 16 February 2024; accepted 22 February 2024. Date of publication 28 February 2024; date of current version 18 March 2024. This work was supported in part by the U.S. National Science Foundation (NSF) under Grant CCF-1919289/SPX-2333009, Grant CNS-2009057, and Grant OAC-2209563, and in part by the Semiconductor Research Corporation (SRC). Recommended for acceptance by B. DiMartino. *(Corresponding author: Yi-Chien Lin.)*

The authors are with the Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089 USA (e-mail: yichienl@usc.edu; bingyizh@usc.edu; prasanna@usc.edu).

Digital Object Identifier 10.1109/TPDS.2024.3371332

To reduce the development effort, HitGNN features a kernel library, which consists of optimized GNN kernels that can be used off-the-shelf. To mask the hardware implementation details, HitGNN provides application programming interfaces (APIs) that allow developers to easily implement various training algorithms and GNN models with only a handful of lines of code. Lastly, the host program of HitGNN performs mini-batch sampling, handles workload imbalance, and reduces data communication among FPGAs.

We summarize our contributions as follows:

- We propose HitGNN, a generic framework that can automatically map various synchronous GNN training algorithms such as P^3, DistDGL and PaGraph, and various GNN models such as GCN and GraphSAGE on a CPU+Multi-FPGA heterogeneous platform for acceleration.
- To abstract away the hardware implementation details, HitGNN features easy-to-use programming APIs, allowing users to specify various synchronous GNN training algorithms and GNN models with a handful of lines of software code.
- To reduce the development effort, we design a GNN kernel library that consists of parameterized and optimized kernels of various well-known GNN models, and a DSE engine that can automatically determine the accelerator configuration.
- To realize a scalable design, we develop optimizations to reduce FPGA-to-FPGA communication overhead, and balance the workload among the FPGAs.
- Compared with state-of-the-art synchronous GNN training implementations on a multi-GPU platform, HitGNN achieves up to 27.21× bandwidth efficiency, and 4.26× speedup using much less compute power and memory bandwidth.

II. Background

A. GNN Models

Given an input graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$, where $\mathcal{V}$, $\mathcal{E}$, and $\boldsymbol{X}$ denote the set of vertices, the set of edges, and the feature matrix of the graph, respectively, a GNN model is specified by:

- L: number of layers.
- $\mathcal{V}^t$: a set of target vertices to be inferred.
- f^l: hidden dimension of layer l $(1 \leqslant l \leqslant L)$.
- A mechanism to construct mini-batches, including:
- The mechanism to construct $\mathcal{V}^l$: the set of vertices in layer l $(0 \leqslant l \leqslant L)$. $|\mathcal{V}^l|$ denotes the number of vertices in layer l.
- The mechanism to construct $\boldsymbol{A}^l \in \mathbb{R}^{|\mathcal{V}^{l-1}| \times |\mathcal{V}^l|}$: adjacency matrix for feature aggregation in layer l $(1 \leqslant l \leqslant L)$. $\boldsymbol{A}^l$ defines the inter-layer connectivity (edges) between $\mathcal{V}^{l-1}$ and $\mathcal{V}^l$.
- **Aggregate()** function that is used by each vertex to aggregate information from its neighbors.
- **Update()** function including a multi-layer perceptron (MLP) and an activation function $\sigma()$ that is used to perform feature update.

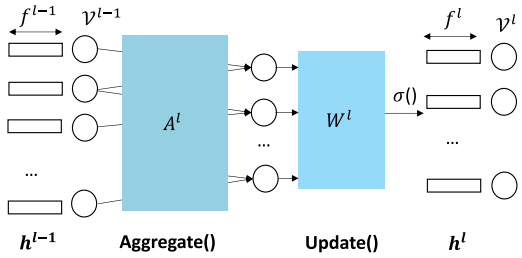

Fig. 1. Computation abstraction of a GNN layer.

Algorithm 1: GNN Computation Abstraction.

for $l = 1 \ldots L$ **do**
  for vertex $v \in \mathcal{V}^l$ **do**
    $\boldsymbol{a}_v^l = \mathbf{Aggregate}(\boldsymbol{h}_u^{l-1} : u \in \mathcal{N}(v) \text{ and } u \in \mathcal{V}^{l-1})$
    $\boldsymbol{h}_v^l = \mathbf{Update}(\boldsymbol{a}_i^l, \boldsymbol{W}^l, \sigma())$

- $\boldsymbol{W}^l \in \mathbb{R}^{f^{l-1} \times f^l}$: weight matrix of layer l $(1 \leqslant l \leqslant L)$ that is used in update function to perform linear transformation of vertex features.
- $\boldsymbol{X} \in \mathbb{R}^{|\mathcal{V}| \times f^0}$: feature matrix of the full graph, where each row represents the feature vector of a vertex.
- $\boldsymbol{h}^l \in \mathbb{R}^{|\mathcal{V}^l| \times f^l}$: the vertex feature matrix in layer l $(0 \leqslant l \leqslant L)$. Moreover, the feature matrix of the input layer is the input feature matrix, i.e., $\boldsymbol{h}^0 = \boldsymbol{X}$; and the feature matrix of the last layer $\boldsymbol{h}^L$ is the node embeddings of the target vertices $\mathcal{V}^t$.

GNN learns to generate low-dimensional vector representation (i.e., node embedding) for a set of target vertices $\mathcal{V}^t$ by iteratively aggregating and updating the vertex features from their $L-$hop neighbors. We depict the computation abstraction of a GNN layer in Fig. 1. Starting from layer 1, the GNN model computes the feature vector of each vertex in $\mathcal{V}^1$ by aggregating and updating the feature vectors of its neighbor vertices in $\mathcal{V}^0$; this process is repeated L times until the node embeddings of the target vertices $\mathcal{V}^t$ (which is $\mathcal{V}^L$) are derived. The computation process of a GNN model is shown in Algorithm 1, which is also known as the aggregate-update paradigm [18]. $\boldsymbol{a}_v^l \in \mathbb{R}^{f^l}$ is the intermediate result of $v \in \mathcal{V}^l$, and $\mathcal{N}_s(v)$ denotes sampled neighbors of v in $\mathcal{V}^{l-1}$.

B. Mini-Batch GNN Training

GNN models can be trained in mini-batch fashion, the training process consists of five stages [3], [6]: sampling, forward propagation, loss calculation, back propagation and weight update. In the sampling stage, a set of vertices and adjacency matrices are sampled from the input graph topology $\mathcal{G}(\mathcal{V}, \mathcal{E})$. We use $\mathcal{V}^l$ to denote the vertices sampled from $\mathcal{V}$ in layer l. $\boldsymbol{A}^l$ denotes the sampled adjacency matrix, which describes inter-layer connections (edges) between $\mathcal{V}^{l-1}$ and $\mathcal{V}^l$ within the mini-batch. A mini-batch consists of target vertices $\mathcal{V}^L$, sampled vertices for each layer $\{\mathcal{V}^l : 0 \leqslant l \leqslant L-1\}$, and sampled adjacency matrices (edges) $\{\boldsymbol{A}^l : 1 \leqslant l \leqslant L-1\}$.

Algorithm 2: Mini-Batch GNN Training Algorithm.

1: **for** each iteration **do**
2: **Sampling**($\mathcal{G}(\mathcal{V},\mathcal{E})$) ▷ Derive mini-batches
3: **for** $l = 1 \ldots L$ **do** ▷ Forward Propagation
4: **for** vertex $v \in \mathcal{V}^l$ **do**
5: $\boldsymbol{a}_v^l = \mathbf{Aggregate}(\boldsymbol{h}_u^{l-1} : u \in \mathcal{N}_s(v), u \in \mathcal{V}^{l-1})$
6: $\boldsymbol{h}_v^l = \mathbf{Update}(\boldsymbol{a}_i^l, \boldsymbol{W}^l, \sigma())$
7: **CalculateLoss**($\{\boldsymbol{h}_i^L : v_i \in \mathcal{V}^L\}$)
8: **BackPropagation**() ▷ Derive gradient of W^l
9: **WeightUpdate**()

In the forward propagation stage, the mini-batch is processed layer by layer; the output of the last layer is the node embeddings of the target vertices $\{\boldsymbol{h}_v^L : v \in \mathcal{V}^L\}$, which are then compared with the ground truth for loss calculation. The calculated loss served as the input for back propagation, which performs a similar computation as forward propagation but in the reverse direction. Finally, the gradients of $\boldsymbol{W}^l$ in each layer are derived for weight update. We show the steps of GNN training in Algorithm 2.

GNN models can also be trained using full-graph, this approach does not require the sampling stage; however, full-graph training causes large memory footprint [19], [20] that may not fit in a device memory (e.g., FPGA local DDR). Therefore, HitGNN focuses on accelerating mini-batch GNN training as it demonstrates advantages in accuracy, scalability on large graphs, and has been adopted by many state-of-the-art GNN frameworks [6], [16], [21].

C. Synchronous GNN Training

Existing works [14], [15], [16] utilize Synchronous Stochastic Gradient Descent (SGD) [22] to train GNN on a multi-CPU or multi-GPU platform. For the rest of the paper, we use *synchronous GNN training* to refer to training GNN on multiple devices in parallel using synchronous SGD.

Synchronous GNN training is similar to Algorithm 2, but with two additional stages: graph preprocessing and gradient synchronization. The first stage is graph preprocessing. In this stage, the input graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ is partitioned and distributed to each device such as GPU or FPGA for parallel training. In addition to graph partitioning, the graph preprocessing stage also performs feature storing which stores feature vectors in the device local memory (e.g., GPU global memory or FPGA local DDR). For devices like GPU or FPGA, the entire feature matrix $\boldsymbol{X}$ of a large-scale graph may be too large to fit in the local memory; thus, existing works [14], [15], [16] develop various feature storing strategies to store only part of the feature matrix in the device local memory.

We use $\boldsymbol{X}_i$ to denote the selected feature vectors stored in the local memory of device i. For works like DistDGL, the feature storing strategy is based on the result of graph partitioning; i.e., if v_j belongs to partition i, then the feature vector of $v_j \in X_i$. Other works like PaGraph develop caching strategies to store feature vectors of frequently accessed vertices, which is independent of the graph partitioning. When the graph preprocessing is done,

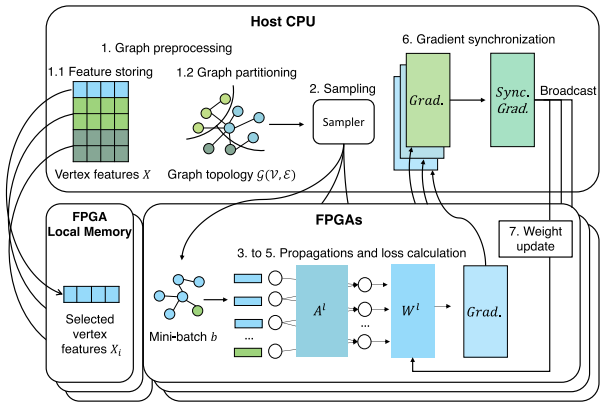

Fig. 2. Synchronous GNN training on a CPU+Multi-FPGA platform.

each device performs forward propagation, loss calculation, and back propagation in parallel. Then, a gradient synchronization is performed, which averages the gradients collected from each device. Then, the averaged gradient is broadcast to update the model weight within each device. We depict the workflow of synchronous GNN training on a CPU+Multi-FPGA platform in Fig. 2. The CPU+Multi-FPGA platform consists of a single CPU or multiple CPUs (depending on the number of sockets in the machine), and multiple FPGAs. The CPUs are connected to the FPGAs via PCIe and each FPGA is connected to a local DDR memory.

We list several representative synchronous GNN training algorithms in Table I. The differences among these algorithms are in graph partitioning and feature storing strategy. Other stages such as forward propagation, gradient synchronization, etc. are identical. Thus, we only show the two different stages for simplicity.

D. Related Work

DNN acceleration using FPGA cluster: [23], [24] accelerate deep neural network (DNN) training using an FPGA cluster. While these works show promising results in terms of performance and energy efficiency, DNN accelerators cannot be directly adapted to GNN training. This is because the computation characteristics of DNN and GNN are different: DNN models feature structured input data with high computation intensity, while GNN models feature unstructured input data with low computation intensity.

Hardware Design Space Exploration: [25], [26], [27] propose design space exploration (DSE) for DNN accelerators on FPGAs. These works formulate analytical models that predict the accelerator performance by considering DNN model meta-data and FPGA hardware meta-data. As mentioned above in the previous paragraph, the computation characteristics of DNN are very different from GNN; thus, existing DSE engines cannot be directly used for exploring GNN accelerators.

GNN training acceleration using a single accelerator: [28] accelerates GNN training on a CPU-FPGA platform and achieves high performance and energy efficiency. However, [28] is an accelerator specific for neighbor sampling GNNs [3],

TABLE I
SYNCHRONOUS GNN TRAINING ALGORITHMS

Synchronous GNN Training Algorithm	Graph Partitioning	Feature Storing Strategy
DistDGL [15]	METIS with multi-constraints	Based on the graph partitioning
PaGraph [16]	A greedy approach which aims to balance the number of training vertices among partitions	Store feature vectors of vertices with high out-degree
P^3 [14]	Partition along the feature dimension	Based on the graph partitioning

and cannot support different GNN models. [10] is the first GNN training framework on FPGA that supports different GNN models. However, [10] does not support synchronous GNN training, which is essential for GNN training on large-scale graphs using multiple accelerators. Similarly, several works accelerate using ASIC [18], [29], [30]; while these works are capable of supporting various GNN models, they can only run on a single accelerator. This limits the computation and memory resources that can be utilized, and thus limits the achievable performance.

GNN training acceleration using multiple FPGAs: [19] accelerates GNN training on a distributed platform, where the graph is stored in multiple nodes. On a distributed platform, the training performance is bottlenecked by the sampling stage. In this work, we focus on a single-node platform with multiple attached FPGAs. On such a platform, the performance is bottlenecked by the propagation stage. [31] accelerates GNN training on a CPU+Multi-FPGA platform; however, [31] can only accelerate a specific synchronous GNN training algorithm. It is necessary to support various algorithms because they outperform each other under different setup (e.g., GNN model, dataset) [32]. In this work, we propose a general framework that can accelerate various synchronous GNN training algorithms, while being able to scale to a CPU+Multi-FPGA platform.

III. CHALLENGES

We identify several challenges in accelerating synchronous GNN training on a CPU+Multi-FPGA platform.

1) Load imbalance among graph partitions: Unlike traditional Deep Neural Networks (DNNs), it is challenging to achieve load balance when accelerating GNN on a CPU+Multi-FPGA platform. DNNs perform computations on grid-structured data such as an image; the workload can be easily balanced among FPGAs by partitioning the images into equal-size chunks. However, it is non-trivial to partition the graph into sub-graphs with a balanced workload as the GNN workload depends on numerous factors (Section VI).

2) Cross-partition communication overhead: In addition to load balancing, reducing the expensive inter-FPGA communication overhead is challenging. This is because graph-structured data inherits complex data dependencies, and cross-partition edges exist after graph partitioning. The cross-partition edges incur inter-FPGA communication during GNN training, which limits the achievable speedup on the CPU+Multi-FPGA platform.

3) Development cost and barrier: Developing a GNN training accelerator on a CPU+Multi-FPGA platform is time-consuming, and requires hardware expertise. In particular, users need to be familiar with data layout, memory organization, and many more to accelerate the communication-intensive GNN training. Due to the large design space of CPU+Multi-FPGA platform, it is also non-trivial and time-consuming to perform the design space exploration.

To address Challenges 1 and 2, we develop several optimizations for the CPU+Multi-FPGA platform to increase scalability. In addition, these optimizations do not alter the semantics of the original GNN training algorithm; thus, the accuracy and convergence rate remains the same after applying the optimizations (Section V). For Challenge 3, HitGNN features optimized GNN kernels that can be used off-the-shelves; the kernels are parameterized and can be used to accelerate various GNN models. Furthermore, HitGNN automates the design steps via the Software Generator and the Accelerator Generator (Section IV), which includes automated design space exploration.

IV. FRAMEWORK

We describe the workflow of HitGNN in the following: in the design phase, user specifies the synchronous GNN training algorithm, GNN model, and the target platform metadata. Then, HitGNN automatically generates optimized accelerator designs and a host program (Section IV-A); during the runtime phase, user launches the GNN training on the CPU+Multi-FPGA platform. We show how the generated designs are mapped to the target platform in Section IV-B.

A. Framework Overview

We depict the framework overview of HitGNN in Fig. 3. HitGNN takes a synchronous GNN training algorithm, a GNN model, and platform metadata as input, and generates a high-throughput design to accelerate the training on a CPU+Multi-FPGA platform. In particular, the design consists of (1) a host program that manages task scheduling, data communication, and mini-batch sampling; and (2) accelerator designs which are optimized GNN kernels that run on the FPGA. In the input program, user specifies a synchronous GNN training algorithm via two sets of APIs:

- *Graph APIs:* specify the graph partitioning and feature storing strategy for the graph preprocessing stage mentioned in Section II-C.
- *GNN APIs:* parameters that define a GNN model mentioned in Section II-A.

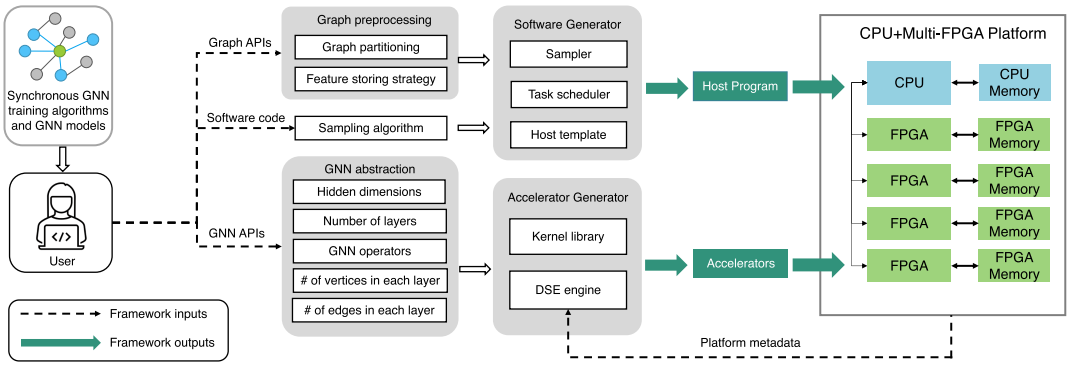

Fig. 3. Framework overview.

HitGNN then parses the input program, and extracts the abstraction of the synchronous GNN training algorithm, which serves as the intermediate representation for the software and accelerator generator to produce the design that runs on the CPU+Multi-FPGA platform. We describe each of the building blocks as follows:

Software generator: Given the user-specified inputs, the software generator produces a host program. During the preprocessing stage, the host program performs graph partitioning, and distributes the vertex features to each FPGA. During training, the host program performs mini-batch generation and distributes the mini-batches to each FPGA. The host program also consists of a runtime system that manages FPGA task scheduling and data communication.

Accelerator generator: The accelerator generator parses the user input and generates parameterized hardware design using the Kernel Library; the parameters (i.e., accelerator configuration) are determined by the DSE Engine. Then, the accelerator generator produces synthesizable hardware for the target FPGA.

- *Kernel Library:* HitGNN provides optimized hardware kernels written in high-level synthesis (HLS) for several widely-used GNN models. User can either implement existing models in the kernel library or customize their own model using the optimized kernel templates.
- *DSE Engine:* The DSE engine takes the platform metadata (e.g., PCIe bandwidth, number of FPGAs) as input, explores the hardware design space, and generates accelerator configurations that optimize the GNN training throughput (Section VI).

B. System Overview

Fig. 4 depicts the mapping of a synchronous GNN training algorithm onto a CPU+Multi-FPGA platform. During runtime, the host program on the CPU first performs graph preprocessing, and distributes selected vertex features to each FPGA. Then, the host launches GNN training. A sampler produces mini-batches, and the runtime system distributes the mini-batches to the FPGA local memory. In the meantime, the FPGAs perform GNN operations. If a vertex feature needed for computation is not present in the FPGA local memory, the FPGA sends a request to host CPU, and the runtime system reads the data from the

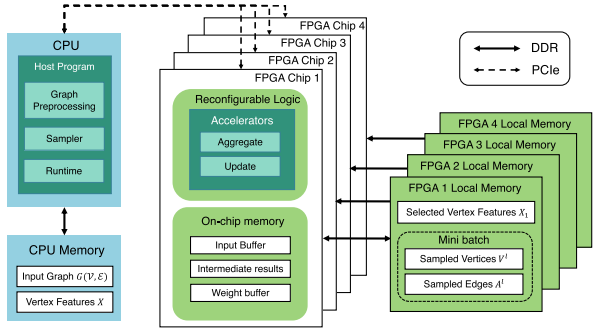

Fig. 4. System overview.

CPU memory and transfers it to the FPGA (Section V-B). After the backpropagation is performed and the gradients are derived, each FPGA sends the gradients back to the host for synchronization. The runtime system averages the gathered gradients, and then broadcasts the averaged gradients back to each FPGA to perform a global weight update.

HitGNN performs graph preprocessing and sampling on the host CPU, and performs GNN operations including feature aggregation and feature update on the FPGAs. Based on this task assignment, the input graph topology $\mathcal{G}(\mathcal{V},\mathcal{E})$ and vertex feature X are stored in the host memory for the host CPU to perform graph preprocessing and sampling; the mini-batch topology $\mathcal{V}^l$ and $\boldsymbol{A}^l$ and selected vertex features X_i are stored in the FPGA local memory for the FPGA to perform GNN operations.

C. High-Level APIs

Table II summarizes the high-level APIs provided to the users to program the synchronous GNN training using Python. Listing 1 is an example of mapping a synchronous GNN training algorithm onto the target CPU+Multi-FPGA platform using HitGNN.

V. CPU+Multi-FPGA Optimizations

Accelerating synchronous GNN training on a CPU+Multi-FPGA platform suffers from workload imbalance, and high data communication overhead. We describe the optimizations adopted to tackle these challenges in Section V-A and V-B. In

TABLE II
APPLICATION PROGRAMMING INTERFACES OF HITGNN

API Type	API Functions	Description
Graph APIs	Graph_Partition( )	Assign a set of vertices $\mathcal{V}$ and edges $\mathcal{E}$ to a FPGA
	Feature_Storing( )	Transfer selected vertex feature $\boldsymbol{X}_i$ to the local memory of a FPGA
GNN APIs	GNN_Parameters( )	Number of layer L, and feature length f^l
	GNN_Computation( )	The layer operators in GNN model. Specify an off-the-shelf GNN model or "customize"
	Scatter( )	User defined function, required only if customized layer operator is specified
	Gather( )	User defined function, required only if customized layer operator is specified
	Update( )	User defined function, required only if customized layer operator is specified
	GNN_Model( )	Build GNN model using GNN parameters
Host APIs	FPGA_Metadata( )	FPGA memory bandwidth, number of DSPs, LUTs, etc.
	Platform_Metadata( )	PCIe memory bandwidth, number of FPGAs, etc.
	Generate_Design( )	Generate hardware design and software design
	Load_Input_Graph( )	File path to read input graph
	Start_training( )	Run GNN training
	Save_model( )	Save trained GNN model

```
### Design Phase ###
'''
Run graph preprocessing programs to produce
V[p], E[p] and X[p]; p = # of FPGAs
'''
for i in range(p): #assign graph data to each FPGA
    Graph_Partition(V[i], E[i], i)
    Feature_Storing(X[i], i)

samp = open('sampling_program')
GNN_comp = GNN_Computation('GCN')
GNN_para = GNN_Parameters(L=2, hidden=[256])
Model = GNN_Model(GNN_comp, GNN_para)

#specify the resoruces of a single super logic
    region, using Xilinx-U250 as an example
for i in range(p):
    FPGApara[i] = FPGA_Metadata(SLR = 4, DSP=3072,
    LUT=423000, URAM=320, BRAM= , BW=19.25)
Platform = Platform_Metadata(BW = bw, FPGA =
    FPGApara, FPGA_connect = 16)
bitstream, runtime = Generate_Design(Model, samp,
    Platform) #generate host and accelerator design,
     return the pointers

### Runtime Phase ###
Graph = LoadInputGraph('ogbn-product', Path='')
Init(bitstream) # initialize the hardware platform
start_training(runtime, Graph, epochs=10)
save_model()
```

Listing 1. An example user program.

addition, we introduce the kernel library in HitGNN, which consists of GNN kernels optimized to achieve high training throughput (Section V-C).

A. Workload Balancing

In the graph preprocessing stage (Section II-C), the graph is partitioned into p partitions where p is the number of FPGAs on the target platform. During the sampling stage, the sampler samples mini-batches from each graph partition, and distributes the mini-batches to each FPGA. As introduced in Section III, the number of vertices and edges within each partition is different; thus, the number of mini-batches within each graph partition is also different and leads to workload imbalance. We propose a two-stage task scheduler to balance the workload among FPGAs.

Stage 1: In stage 1, the sampler is able to sample mini-batches from each graph partition. The task scheduler distributes the mini-batches to each FPGA based on the graph partition the mini-batches are sampled from.

Algorithm 3: Two-Stage Task Scheduling.

Input: partitioned graph topology $\mathcal{V}$[i], $\mathcal{E}$[i], where $0 \leq$ i $< p$
Output: mini-batches b and its FPGA assignment

```
# Stage 1
while # of mini-batches in each partition > 0 do
  for i in 0 to p do
    b = Sample(V[i], E[i])
    Distribute(b, i)                    ▷ Distribute to FPGA i
# Stage 2
cnt = 0                  ▷ Counter used for round-robin sampling
while # of mini-batches in any partition > 0 do
  for i in 0 to p-1 do
    if # of mini-batches in partition i > 0 then
      avail.append(i)          ▷ List of partitions to sample
    else
      idle.append(i)                    ▷ List of idle FPGAs
  for i in 0 to avail.length() do
    j = avail[i]
    b = Sample(V[j], E[j])
    Distribute(b, j)
  for i in 0 to idle.length() do
    j = avail[cnt % avail.length()]
    b = Sample(V[j], E[j])    ▷ Sample extra mini-batch
    Distribute(b, idle[i])          ▷ Assign to idle FPGA
    cnt++
  avail.clear()
  idle.clear()
```

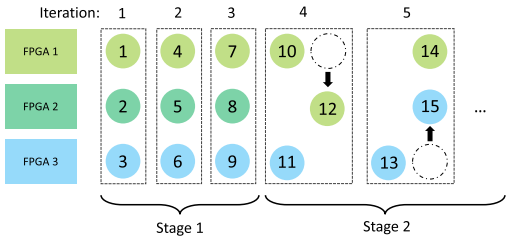

Fig. 5. Two-stage task scheduling.

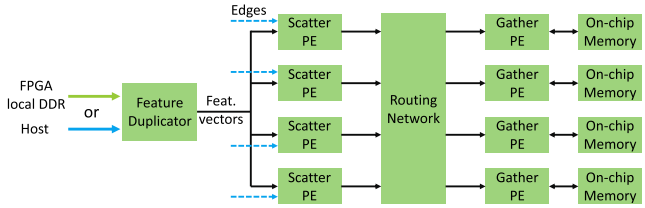

Fig. 6. Hardware design of the aggregate kernel.

Stage 2: In stage 2, the mini-batches within some graph partitions have all been executed. We show an example in Fig. 5 where all the mini-batches within partition 2 (which is assigned to FPGA 2) have been executed after iteration 3. Thus, for iteration 4, the sampler samples an extra mini-batch from partition 1 to produce 3 mini-batches to perform synchronous SGD. The extra mini-batch (i.e., mini-batch 12) is distributed to FPGA 1 by default, which leads to workload imbalance since FPGA 1 needs to execute 2 mini-batches in iteration 4. To balance the workload, the task scheduler distributes the extra mini-batch to an idle FPGA; in this example, the FPGA 2. The sampler samples the partitions in a round-robin fashion, so for iteration 5, the sampler moves on and samples a mini-batch from partition 3 and produces mini-batch 13. We show the general two-stage task scheduling with any number of FPGAs p in Algorithm 3.

B. Data Communication

In order to train GNN on multiple FPGAs in parallel, the input graph is partitioned and distributed to each FPGA. Because of the complex data dependency of graph-structured data, when an FPGA is performing GNN training, the data required may not reside in the local partition. In this case, the FPGA needs to fetch data from another FPGA which incurs FPGA-to-FPGA communication. With more FPGAs equipped, the graph is split into more partitions, and the data required is more likely to reside in a remote partition. Previous work has shown that the inter-device communication overhead can easily dominate the synchronous GNN training time when training on a multi-GPU platform [33].

We show the overview of a CPU+Multi-FPGA platform in Fig. 4; the FPGA-to-FPGA communication is done via a shared memory space in the CPU memory. In particular, data is first copied from the FPGA memory to the shared memory space and then transferred to another FPGA. Compared with CPU-FPGA communication, FPGA-to-FPGA communication is much slower because it requires additional data copying to the CPU memory [34]. Thus, we propose to fetch data directly from the CPU memory. In particular, whenever a vertex feature required for training is not presented in the FPGA local memory, the FPGA sends a request to the host CPU to fetch the data. Fetching vertex features from the CPU memory is feasible because the CPU memory holds the entire graph (Section IV-B). This optimization avoids reading vertex features from another FPGA and thus reduces FPGA-to-FPGA communications.

Though modern FPGAs support direct communication to another FPGA via Ethernet, this feature is not yet supported on most cloud-FPGA platforms such as Amazon Web Services (AWS) [35] and Microsoft Azure [36]. In addition, the controller logic to read data from another FPGA becomes complicated as the platform is equipped with more FPGAs since the controller needs to consider the network topology of FPGA-FPGA connections, and decides the routing to fetch the required data.

C. Optimized Kernel Library

GNN computation is time-consuming because it incurs substantial random memory access. We develop parameterized hardware templates with optimized data layout and memory organization that can effectively reduce the communication overhead, and therefore achieve high GNN training throughput. Fig. 6 shows an example of 4 scatter-gather processing elements (PEs). During pre-processing, the edges of the input graph are sorted by the source vertex; this ensures that edges with the same source vertex are processed together, and the kernel does not need to re-load the same source vertex later. Initially, a feature vector of the source vertex is loaded and broadcast to each scatter PE via the Feature Duplicator. In each iteration, an edge is loaded to each scatter PE. If the source vertex of the loaded edge matches the loaded feature vector, the scatter PE reuses the loaded feature vector for computation; otherwise, the scatter PE waits for the Feature Duplicator to broadcast the feature vector of the next source vertex to replace the current loaded feature vector. Note that during computation, the feature vector of the next source vertex is also prefetched to the Feature Duplicator. For the update kernel, we adopt a systolic-array-based design to perform block matrix multiplication. While the kernels exploit data parallelism, and increase data reuse, they do not alter the GNN training algorithm. The parameterized hardware template follows the computation paradigm mentioned in Algorithm 1; thus, it is able to support various GNN models.

VI. Hardware Design Space Exploration

HitGNN features a DSE engine that can explore the hardware design space of a CPU+Multi-FPGA platform, and decides the accelerator configurations automatically. In particular, the engine takes the configuration of a mini-batch ($\{|\mathcal{V}^l| : 0 \leqslant l \leqslant L\}$, $\{|\mathcal{A}^l| : 1 \leqslant l \leqslant L\}$), GNN hidden dimensions $\{f^l : 0 \leqslant l \leqslant L\}$, and platform metadata as input, and output a set of parameters that optimizes the GNN training throughput.

A. Resource Utilization Model

The resource utilization model formulates the hardware resource consumption given a set of accelerator configurations. In

our kernel design, DSPs and LUTs are used the most among the various hardware resources. Thus, we model the usage of LUTs and DSPs as our constraints:

$$\lambda_1 \times m + \lambda_2 \times n \leq N_{DSP} \tag{1}$$

$$\rho_1 \times m + \rho_2 \times n + \rho_3 \times n\log(n) \leq N_{LUT} \tag{2}$$

N_{DSP} and N_{LUT} denote the available DSPs and LUTs on a single FPGA platform. For a multi-FPGA platform, each FPGA is constrained by (1) and 2 independently. All constraints of each FPGA on the target platform need to be satisfied to produce a valid design. We use m to denote the number of processing elements (PEs) in the update kernel, and n to denote the number of scatter-gather PEs in the aggregate kernel (Section V-C). The coefficients λ_i $(1 \leqslant i \leqslant 2)$ and ρ_i $(1 \leqslant i \leqslant 3)$ are constants that indicate the resource consumption for each PE. The utilization of DSPs grows linearly as we instantiate more PEs; as for LUTs, an additional $n\log(n)$ term is introduced to model the LUT overhead of the routing network in the aggregate kernel (Fig. 6).

B. Performance Model

We define the throughput of GNN training as Number of Vertices Traversed Per Second (NVTPS):

$$\text{Throughput} = \frac{\sum_{i=0}^{p}\sum_{l=0}^{L}|\mathcal{V}^l|}{t_{\text{parallel}}} \tag{3}$$

The numerator indicates the total amount of vertices traversed in one iteration. For a multi-FPGA platform with p FPGAs, p mini-batches are computed concurrently. The denominator t_{parallel} is the parallel execution time of one training iteration, which includes the time for p FPGAs to perform forward propagation, loss calculation, etc. (details are in Algorithm 2) and the time for gradient synchronization. t_{parallel} can be model as:

$$t_{\text{parallel}} = \max_{i \in p}\left(t^i_{\text{execution}}\right) + t_{\text{gradient\_}sync} \tag{4}$$

Since there are p FPGAs processing in parallel, the parallel execution time is limited by the slowest FPGA; an extra overhead $t_{\text{gradient\_sync}}$ is introduced to for synchronization.

We overlap sampling stage and the GNN computations, so the average execution time on a single FPGA $t_{\text{execution}}$ is estimated as:

$$t_{\text{execution}} = \max\left(t_{\text{sampling}}, t_{\text{GNN}}\right)$$
$$t_{\text{GNN}} = t_{\text{FP}} + t_{\text{LC}} + t_{\text{BP}} \tag{5}$$

where t_{GNN} consists of the execution time of forward propagation t_{FP}, loss calculation t_{LC}, and back propagation t_{BP}.

The total propagation time t_{FP} and t_{BP} is the sum of the execution time of each layer; the execution time of each layer is decided by the task that takes longer to complete since aggregation stage and update stage are pipelined. The aggregation stage consists of two tasks: (1) vertex feature loading, and (2) computation. Since the two tasks are pipelined, $t_{\text{aggregate}}$ can be modeled as:

$$t^l_{\text{aggregate}} = \max(t^l_{\text{load}}, t^l_{\text{compute}}) \tag{6}$$

$$t^l_{\text{load}} = \frac{|\mathcal{V}^{l-1}| \times \beta \times f^l \times S_{\text{feat}}}{BW_{\text{DDR}}} + \frac{|\mathcal{V}^{l-1}| \times (1-\beta) \times f^l \times S_{\text{feat}}}{BW_{\text{PCIe}}} \tag{7}$$

$$t^l_{\text{compute}} = \frac{|\boldsymbol{A}^l| \times f^l}{n \times \text{PE}_{\text{SIMD}} \times \text{Freq.}} \tag{8}$$

We model the vertex feature loading time t^l_{load} as $(\text{datatransferred})/(\text{effectivebandwidth})$. f^l is the feature length, and S_{feat} is the data size of each feature. β is the ratio of fetching data from a local graph partition stored in the local DDR memory (first term of (7)). If the data is in the remote partition, the data is fetched from host via PCIe (second term of (7)).

We model the compute time as (# of operations)/(# of PEs $\times$ kernel frequency). $|\boldsymbol{A}^l|$ is the number of edges in each layer. n denotes there are n scatter-gather PEs instantiated in the aggregation kernel. Each PE features vector parallelism (Section V-C), and can compute 512-bit of data each cycle; for single-precision floating point data, $\text{PE}_{SIMD} = 512/32 = 16$.

The feature update is modeled as:

$$t_{\text{update}} = \frac{|V^l| \times f^l \times f^{l+1}}{m \times freq} \tag{9}$$

We model the t_{update} as (# of operations)/(# of PEs $\times$ kernel frequency). The numerator is the complexity of the multi-layer perception, and m denotes the number of PEs instantiated in the update kernel.

C. Hardware DSE Engine

Exploring the design space of a CPU+Multi-FPGA platform is challenging since the design space can be large. For example, we can assign some FPGAs to perform feature aggregation, and the others to perform feature update; or, we can assign each FPGA to perform both tasks, but with less parallelism instantiated in both kernels. Our DSE engine adopts the latter approach for two reasons: (1) the bottleneck of GNN computations is vertex feature loading during the feature aggregation, so our design should utilize as much memory bandwidth as possible. That is, all the FPGAs (as opposed to several FPGAs) should utilize their memory bandwidth to perform feature aggregation; and (2) if the FPGAs are able to perform both feature aggregation and feature update independently, the intermediate results can be reused directly; this reduces high volume of data communication among FPGAs.

The DSE engine explores the design space on each FPGA, and decides the accelerator configurations that optimize the GNN training throughput. Furthermore, many modern FPGAs consist of multiple dies, and the available resources may vary across dies. Thus, the engine performs DSE for each die to explore the optimal configuration. We assume that each die is connected to one DDR channel (e.g., Xilinx Alveo U250) for simplicity. The DSE engine first constructs a search space by obtaining the maximum value of n and m separately using (1) and (2). Then, the DSE engine performs a parameter sweep through all the possible configurations. For each set of configurations, the

Algorithm 4: Hardware DSE Engine.

```
for each FPGA do
  for each die do
    Construct_Search_Space()        ▷ Derive n_max, m_max
    max_val = 0
    for n = 1...n_max do            ▷ Exhaustive search
      for m = 1...m_max do
        # Check resource availability using (1), (2)
        Valid ← Check_resource_availability(n, m)
        if Valid and Throughput(n, m) > max_val :
        then
          max_val ← Throughput(n, m)
          Save_configuration( n, m )
```

TABLE III
SPECIFICATIONS OF THE PLATFORMS

Platforms	CPU AMD EPYC 7763	GPU Nvidia RTX A5000	FPGA Xilinx Alveo U250
Technology	TSMC 7 nm+	Samsung 8 nm	TSMC 16 nm
Frequency	2.45 GHz	2000 MHz	300 MHz
Peak Performance	3.6 TFLOPS	27.8 TFLOPS	0.6 TFLOPS
On-chip Memory	256 MB L3 cache	6 MB L2 Cache	54 MB
Memory Bandwidth	205 GB/s	768 GB/s	77 GB/s

DSE engine evaluates its throughput using (3), and eventually obtains the optimal design. We show the steps performed by the DSE engine in Algorithm 4.

VII. EXPERIMENTS

A. Experimental Setup

Environments: We use our framework to generate GNN training implementations on a CPU+Multi-FPGA platform, and compare the training throughput with a multi-GPU platform. Both the multi-GPU and the CPU+Multi-FPGA platform are built on a dual-socket server. The multi-GPU platform is equipped with 4 GPUs, and the CPU+Multi-FPGA platform is equipped with 4 FPGAs. The GPUs or FPGAs are connected to the host CPU via PCIe. We list the data of the host CPU, GPUs, and FPGAs in Table III. Note that both the peak performance and memory bandwidth of the FPGA platform are much lower than the GPU platform; thus, the GNN training performance on the CPU+Multi-FPGA platform highly relies on the proposed optimizations. The multi-GPU baseline is implemented using Python v3.6, PyTorch v1.11, CUDA v11.3, and PyTorch-Geometric v2.0.3. To measure the power consumption on the CPU, GPU, and FPGA, we use PowerTop [37], Nvidia System Management Interface (SMI) [38], and Vitis Analyzer [39], respectively. The implementations on the CPU+Multi-FPGA platform are described in Section VII-B.

Synchronous GNN Training Algorithms: We evaluate our framework using three representative synchronous GNN training algorithms: DistDGL [15], PaGraph [16], and P^3 [14]. Note that we only follow how these algorithms perform graph preprocessing (details in Section II-C), and do not implement the optimizations in the original works since some optimizations are platform-dependent and cannot be applied to the CPU+Multi-FPGA platform. Furthermore, HitGNN already has its own optimizations for the CPU+Multi-FPGA platform.

TABLE IV
DATASETS AND GNN-LAYER DIMENSIONS

Dataset	#Vertices	#Edges	f_0	f_1	f_2
Reddit (RD)	232,965	23,213,838	602	128	41
Yelp (YP)	716,847	13,954,819	300	128	100
Amazon (AM)	1,569,960	264,339,468	200	128	107
ogbn-products (PR)	2,449,029	61,859,140	100	128	47

```
### DistDGL preprocessing example ###
'''
Run multi-constraint METIS in DistDGL to produce
V[p], E[p] and X[p] first
'''
for i in range(p): #assign graph data to each FPGA
    Graph_Partition(V[i], E[i], i)
    Feature_Storing(X[i], i)

### PaGraph preprocessing example ###
'''
Run PaGraph's graph partitioning to produce V[p] and
     E[p], store vertex features with high out
    degree into array X
'''
for i in range(p): #assign graph data to each FPGA
    Graph_Partition(V[i], E[i], i)
    Feature_Storing(X, i) #same X for each FPGA

### P3 preprocessing example ###
for i in range(p):
    Graph_Partition(V, E, i) #entire graph topology
    Feature_Storing(X[i], i) #partitioned along
    feature dimension

samp = open('GraphSAGE_sampler')
GNN_comp = GNN_Computation('GraphSAGE')
GNN_para = GNN_Parameters(L=2, hidden=[128])
Model = GNN_Model(GNN_comp, GNN_para)
```

Listing 2. Example program of the three synchronous GNN training algorithms.

GNN Models and Datasets: We evaluate our framework on two well-known GNN models: GraphSAGE [3] and GCN [40]. We use a 2-layer model with hidden feature size 128, the size of target vertices $|\mathcal{V}^t|$ is set as 1024, and the neighbor sampling size of each layer are 25 and 10; the GNN parameters chosen for evaluation have been widely-used and have shown promising results in many works [3], [6], [7], [32], [40]. We choose four widely-used datasets with over ten million edges for evaluation: Reddit, Yelp, Amazon [6] and ogbn-products [32]. We list the details of the datasets and GNN-layer dimensions in Table IV.

B. Framework Implementation

HitGNN consists of several building blocks to generate a design that runs on a CPU+Multi-FPGA heterogeneous platform. The program parser, DSE engine, software and hardware generator are implemented using Python v3.6, and the accelerator templates are implemented using Xilinx Vitis HLS v2021.2. The host program template is programmed in C++14 with OpenCL library. User interface with HitGNN using the provided APIs (Section IV-C). In Listing 2, we provide an

```
1  //part of the Host Program
2  cl::Device devices = xcl::get_xil_devices();
3  //get all FPGAs on the platform
4  ...//set up and initialization
5
6  for(int iter = 0; iter < Layer; iter++) {
7  // variable "Layer" is specified by user
8      buffer_out =  cl::Buffer(CL_MEM_WRITE_ONLY);
9      aggregare_krnls.setArg(0, buffer_out);
10     ...//set arguments for kernel
11     q.enqueueTask(aggregare_krnls);
12     q.enqueueTask(update_krnls);
13     if(iter == 0){
14     //all-to-all broadcast for P^3, manually added
15         for(i=0;i<p;i++)
16             q.enqueueMigrateMemObjects(part_res[i]);
17         concat(part_res, layer2ip);
18         for(i=0;i<p;i++)
19             q.enqueueMigrateMemObjects(layer2ip[i])
       ;}}
20     q.finish();
21     q.enqueueMigrateMemObjects(buffer_out);
22     //copy result back to host}
23 //part of the generated Accelerator Design
24 read_from_stream(input, tmp_update);
25 if(tmpupdate.valid == 1){
26   index_type dst = tmp_update.dst - dst_offset;
27   reg_update = result_buffer[dst];
28
29   for (int j = 0; j < 16; j++){
30   #pragma HLS unroll = 8 //decide by DSE
31     regupdate.data[j] = regupdate.data[j] +
       tmpupdate.value.data[j];
32     }//user-specified aggregate function
33     resultbuffer[dst] = regupdate}
```

Listing 3. Example of generated code.

example of implementing the three synchronous GNN training algorithms in our experiments; and in Listing 3, we show part of the generated host program and synthesizable accelerator design.

P^3 requires an extra all-to-all broadcast step at the end of the first GNN layer; we regard this extra step as a special case, and do not provide any APIs to handle it for design simplicity. In Line 14-19 of Listing 3, we show that this extra step can be easily implemented using built-in functions of the OpenCL library.

C. DSE Engine Evaluation

Given a GNN model, sampling algorithm, and CPU+Multi-FPGA platform metadata, the DSE engine automatically decides the accelerator configurations. We use n to indicate the number of scatter-gather PEs in the aggregate kernel, and use m to indicate the number of PEs in the update kernel (Section VI). We perform a parameter sweep to explore the FPGA design space as in Algorithm 4. For each design point (n,m), the DSE engine estimates the GNN training throughput on the four datasets (Table IV), and reports the average throughput. We execute the DSE engine on the CPU, and the DSE engine is able to complete the parameter sweep within 10 seconds in all of our test cases. Since the DSE engine systematically searches through the hardware design space, it is able to find an optimal accelerator configuration. As shown in Table V, both accelerator configurations (8,2048) and (16,1024) saturate the available hardware resources. Since feature aggregation is usually the bottleneck of GNN training,

TABLE V
RESOURCE UTILIZATION AND PARALLELISM

Parallelism (n,m)	(8,2048)	(16,1024)
LUTs	72%	65%
DSPs	90%	56%
URAM	48%	34%
BRAM	40%	28%
Estimated Throughput (NVTPS)	97.0 M	92.6 M

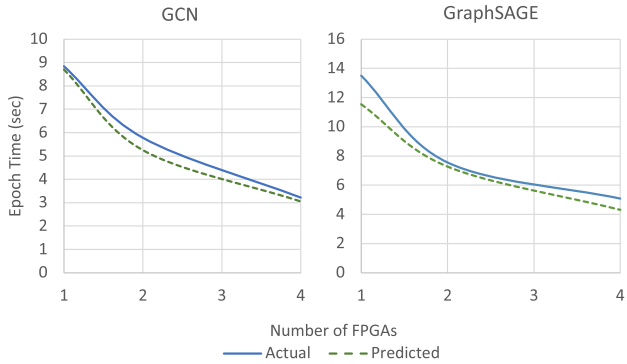

Fig. 7. Predicted performance versus actual performance.

one might choose (16,1024) over (8,2048) to maximize the parallelism of the aggregate kernel, intuitively. However, as the DSE engine suggests, the configuration (8,2048) leads to higher training throughput instead; this is because our optimized kernel effectively reduces the communication overhead of feature aggregation and shifts the bottleneck to the feature update phase. Thus, the configuration (8,2048), which invests more hardware resources in the update kernel delivers higher throughput.

We evaluate our DSE engine by comparing the predicted epoch time with the experimental results. Fig. 7 shows an example of the epoch time comparison on the ogbn-products dataset under various number of FPGAs using the DistDGL training algorithm. The prediction error ranges from 5% to 14% on the average. The error comes from extra latency that is not formulated in our model, such as the overhead of kernel launching and pipeline flushing.

D. Performance Metrics

- *Epoch time:* the time to train one epoch (seconds).
- *Throughput:* we define the throughput as the Number of Vertices Traversed Per Second (NVTPS).
- *Bandwidth efficiency:* throughput divided by the available DRAM memory bandwidth of the target platform (NVTPS/(GB/s)). GNN training throughput highly relies on the available memory bandwidth of the platform; since the memory bandwidth varies on different platforms, normalizing the throughput with the available bandwidth provides a fair comparison across different platforms.
- *Energy consumption:* we measure the energy consumption of training one epoch on the target platform (kJ/epoch).

TABLE VI
CROSS PLATFORM COMPARISON

		Dataset	Reddit		Yelp		Amazon		ogbn-products		Geo. Mean
		Model	GCN	GSG	GCN	GSG	GCN	GSG	GCN	GSG	
DistDGL [15]	Epoch time (s)	GPU	1.29	1.34	1.75	1.79	4.19	4.34	5.03	5.38	-
		Ours	0.62	0.77	0.63	0.87	1.14	1.72	3.06	4.31	-
	Throughput (NVTPS)	GPU	15.6 M	15.1 M	21.6 M	21.1 M	22.6 M	21.8 M	97.5 M	91.2 M	28.8 M (1.00×)
		Ours	32.5 M	26.2 M	59.9 M	43.4 M	83.1 M	55.1 M	160 M	114 M	60.7 M (2.11×)
	BW efficiency (NVTPS/(GB/s))	GPU	4.77 K	4.59 K	6.58 K	6.44 K	6.90 K	6.66 K	29.8 K	27.8 K	8.78 K (1.00×)
		Ours	63.4 K	51.1 K	117 K	84.6 K	162 K	107 K	313 K	222 K	118 K (13.4×)
	Energy consumption (kJ/epoch)	GPU	0.504	0.524	0.684	0.700	1.638	1.696	1.966	2.103	1.048 (1.00×)
		Ours	0.131	0.163	0.133	0.184	0.241	0.364	0.648	0.912	0.269 (3.90×)
PaGraph [16]	Epoch time (s)	GPU	1.21	1.25	1.66	1.69	4.04	4.16	4.61	4.89	-
		Ours	0.53	0.67	0.54	0.77	0.98	1.53	2.64	3.82	-
	Throughput (NVTPS)	GPU	16.7 M	16.1 M	22.7 M	22.3 M	23.4 M	22.8 M	106 M	100 M	30.6 M (1.00×)
		Ours	38.1 M	30.1 M	69.9 M	49.0 M	96.6 M	61.9 M	186 M	128 M	69.8 M (2.28×)
	BW efficiency (NVTPS/(GB/s))	GPU	5.09 K	4.92 K	6.94 K	6.82 K	7.15 K	6.95 K	32.5 K	30.6 K	9.35 K (1.00×)
		Ours	74.2 K	58.7 K	136 K	95.6 K	188 K	121 K	362 K	250 K	136 K (14.6×)
	Energy consumption (kJ/epoch)	GPU	0.473	0.489	0.649	0.661	1.579	1.626	1.802	1.911	0.984 (1.00×)
		Ours	0.112	0.142	0.114	0.163	0.207	0.324	0.559	0.809	0.234 (4.20×)
P^3 [14]	Epoch time (s)	GPU	1.24	1.32	1.82	1.79	4.43	4.41	5.21	5.43	-
		Ours	0.53	0.70	0.55	0.82	1.04	1.70	2.71	4.13	-
	Throughput (NVTPS)	GPU	16.3 M	15.3 M	20.7 M	21.1 M	21.4 M	21.5 M	94.2 M	90.4 M	28.4 M (1.00×)
		Ours	38.1 M	28.8 M	68.6 M	46.0 M	91.1 M	55.7 M	181 M	119 M	66.4 M (2.34×)
	BW efficiency (NVTPS/(GB/s))	GPU	4.96 K	4.66 K	6.33 K	6.44 K	6.52 K	6.55 K	28.7 K	27.6 K	8.67 K (1.00×)
		Ours	74.2 K	56.2 K	134 K	89.7 K	178 K	109 K	353 K	232 K	129 K (14.9×)
	Energy consumption (kJ/epoch)	GPU	0.485	0.516	0.711	0.700	1.731	1.724	2.036	2.122	1.061 (1.00×)
		Ours	0.112	0.148	0.116	0.174	0.220	0.360	0.574	0.874	0.246 (4.32×)

TABLE VII
THROUGHPUT IMPROVEMENT DUE TO OPTIMIZATIONS

Data-Model	Baseline	WB	WB+DC	Speedup
RD-GCN	19.9 M	22.7 M	32.5 M	63%
RD-GSG	16.9 M	19.2 M	26.2 M	55%
YP-GCN	36.4 M	41.9 M	59.9 M	65%
YP-GSG	28.5 M	32.8 M	43.4 M	52%
AM-GCN	50.8 M	59.6 M	84.1 M	64%
AM-GSG	36.5 M	42.9 M	55.1 M	51%
PR-GCN	96.7 M	113 M	160 M	66%
PR-GSG	73.8 M	86.5 M	114 M	54%

E. Cross Platform Comparison

We compare the performance of a set of designs generated by HitGNN that runs on a CPU+Multi-FPGA platform, with state-of-the-art GNN training implementations using PyTorch-Geometric [21] that runs on a multi-GPU platform. The CPU+Multi-FPGA platform has four FPGAs and the multi-GPU platform has four GPUs (Section VII-A). We show the results in Table VI, which uses the metrics defined in Section VII-D for comparison. We use *GPU* to indicate the multi-GPU baseline, and use *Ours* to indicate the designs generated by HitGNN; we use *GSG* to indicate the GraphSAGE [3] model, and *GCN* to indicate the GCN [40] model. Compared with the multi-GPU baseline, HitGNN achieves 2.11×, 2.28×, and 2.34× speedup for the DistDGL, PaGraph, and P^3, respectively. This is because HitGNN features optimizations that balance the workload among the FPGAs and reduce FPGA-to-FPGA communication overhead. To evaluate the effectiveness of our optimizations, we conduct an ablation study. We first evaluate the performance of a baseline design, and then gradually add the workload balancing (WB) optimization and the data communication (DC) optimization to the design. We show the evaluation in Table VII, which uses the DistDGL algorithm as an example, the two optimizations can deliver up to 66% throughput improvement in total. Our workload balancing optimization shows higher performance improvement (14% - 15%) than the workload optimization applied in the state-of-the-art [41] (2% - 5%). This is because HitGNN balances the workload in a more fine-grained manner. Specifically, HitGNN balances the mini-batches distributed to the FPGAs, and the mini-batches have similar workloads; in contrast, [41] balances graph partitions distributed to the GPUs, and the graph partitions have relatively diverse workloads. Our highly-optimized GNN kernels allow HitGNN to achieve 13.4×-14.9 × bandwidth efficiency than the multi-GPU platform w.r.t. the geometric mean. Due to the superior bandwidth efficiency, HitGNN achieves up to 4.26× speedup using only 0.16× memory bandwidth of the multi-GPU platform. Finally, HitGNN demonstrates a significant reduction in energy consumption, using 4.32× less energy compared with state-of-the-art implementations that run on multi-GPU platforms. This is because HitGNN utilizes FPGA for computation, which is a more energy-efficient platform than GPU; in addition, HitGNN requires less time to execute an epoch, which leads to less static power consumption.

F. Comparison With State-of-the-Art FPGA Design

We compare HitGNN with HP-GNN [10], a state-of-the-art framework for GNN training on a CPU-FPGA platform, in Table VIII. We choose the widely-used DistDGL algorithm as an example; other algorithms show similar results in terms of throughput, bandwidth efficiency, etc. (see Table VI). HitGNN achieves 2.20× speedup over HP-GNN. This is because HitGNN supports training with multiple FPGAs, while HP-GNN can train GNN using only a single FPGA. HitGNN achieves 79% bandwidth efficiency of HP-GNN. This is because training with a single FPGA does not incur inter-FPGA communication overhead or imbalanced workload, allowing HP-GNN to achieve a higher bandwidth efficiency than HitGNN; nevertheless, with

TABLE VIII
COMPARISON WITH STATE-OF-THE-ART FPGA DESIGN

		Reddit	Yelp	Geo. Mean
Epoch Time (s)	HP-GNN	1.54	1.83	-
	HitGNN	0.77	0.87	-
Throughput (NVTPS)	HP-GNN	13.0 M	18.1 M	15.3 M (1.00×)
	HitGNN	26.2 M	43.4 M	33.7 M (2.20×)
BW efficiency (NVTPS/(GB/s))	HP-GNN	71.1 K	98.3 K	83.3 K (1.00×)
	HitGNN	51.1 K	84.5 K	65.7 K (0.79×)
Energy consump. (kJ/epoch)	HP-GNN	0.08	0.10	0.09 (1.00×)
	HitGNN	0.16	0.18	0.17 (1.87×)

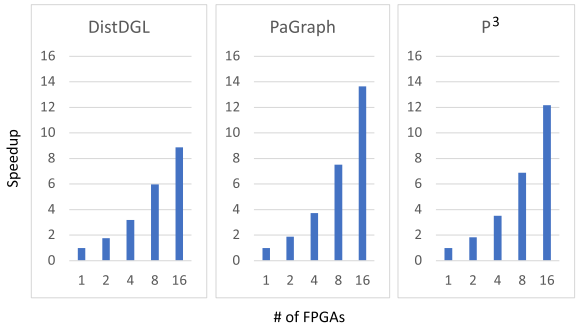

Fig. 8. Scalability evaluation.

our optimizations applied (see Section V), HitGNN effectively mitigates these overheads, and is able to achieve nearly 80% of the bandwidth efficiency of HP-GNN on a CPU+Multi-FPGA platform. More importantly, by supporting CPU+Multi-FPGA platform, HitGNN is capable of training on large graphs that do not fit in the DDR memory of a single FPGA. In contrast, the graph size for HP-GNN is strictly constrained by the available DDR memory of a single FPGA. Due to the utilization of more hardware resources, HitGNN consumes 1.87× more energy than HP-GNN for each training epoch. Note that the 1.87× increase in energy consumption also leads to a 2.20× speedup in terms of throughput.

G. Scalability

We build a CPU+Multi-FPGA platform simulator to evaluate the scalability of HitGNN. The simulator estimates the training performance given the synchronous GNN training algorithm, GNN model, and platform metadata. To verify the simulator, we first implement the host program and the hardware kernels. Then, we measure the host program execution time, and the post-synthesis kernel execution time to fine-tune the simulator. We show the speedup compared with a single FPGA for each algorithm in Fig. 8. By reducing the FPGA-to-FPGA communication overhead, and balancing the workload, HitGNN achieves scalable speedup. The scalability of HitGNN is limited by the CPU memory bandwidth. This is because FPGAs fetch data from the CPU memory if the data required is not presented in the FPGA local memory (Section V-B). Using the EPYC 7763 CPU as an example, the CPU memory (bandwidth = 205 GB/sec.) can serve up to 205/16 = 12.8 FPGAs without saturating the CPU memory bandwidth, where the denominator is the bandwidth of a single CPU-FPGA connection via PCIe; if more FPGAs are added to the platform, the scalability of the speedup starts to decrease since the CPU memory bandwidth is gradually saturated. In Fig. 8, we show that the throughput of HitGNN scales up to 16 FPGAs.

VIII. CONCLUSION

In this paper, we proposed HitGNN, a general framework to generate high-throughput GNN training implementation on a given CPU+Multi-FPGA heterogeneous platform. HitGNN features various optimizations to accelerate synchronous GNN training on the CPU+Multi-FPGA platform. The optimizations of HitGNN can be applied to various synchronous GNN training algorithms; in addition, these optimizations preserves the GNN training semantics; thus, they do not affect the model accuracy or the convergence rate. The implementations generated by HitGNN achieved up to 27× bandwidth efficiency compared with the multi-GPU baseline, and thus achieved up to 4.26× throughput using less compute power and memory bandwidth.

We discuss the limitations of HitGNN. First, HitGNN cannot be directly extended to a distributed platform as it assumes that all the data is stored on a single machine. Second, on a single machine, the scalability of HitGNN is limited by the CPU memory bandwidth. In the future, we plan to improve the system design of HitGNN by handling data communication across multiple nodes; this allows HitGNN to scale to distributed platforms with multiple nodes. We will also improve the scalability of HitGNN on a single machine by exploiting high-speed FPGA interconnection network (e.g., SmartNIC) to relieve the stress on the CPU memory bandwidth.

REFERENCES

[1] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proc. 24th ACM Int. Conf. Knowl. Discov. Data Mining*, 2018, pp. 974–983.
[2] R. Zhu et al., "AliGraph: A comprehensive graph neural network platform," *Proc. VLDB Endowment*, vol. 12, no. 12, pp. 2094–2105, 2019.
[3] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1025–1035.
[4] C.-I. Yang and Y.-P. Li, "Explainable uncertainty quantifications for deep learning-based molecular property prediction," *J. Cheminformatics*, vol. 15, no. 13, 2023. [Online]. Available: https://doi.org/10.1186/s13321-023-00682-3
[5] W. Jiang and J. Luo, "Graph neural network for traffic forecasting: A survey," 2021, *arXiv:2101.11174*.
[6] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Graph-SAINT: Graph sampling based inductive learning method," in *Proc. Int. Conf. Learn. Representations*, 2020.
[7] H. Zeng et al., "Decoupling the depth and scope of graph neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2021, pp. 19665–19679.
[8] Y.-C. Lin, Y. Chen, S. Gobriel, N. Jain, G. K. Jhaand, and V. Prasanna, "Argo: An auto-tuning runtime system for scalable gnn training on multi-core processor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2024.
[9] Y.-C. Lin, G. Deng, and V. Prasanna, "A unified CPU-GPU protocol for GNN training," in *Proc. ACM Int. Conf. Comput. Front.*, 2024, pp. 392–404.
[10] Y.-C. Lin, B. Zhang, and V. Prasanna, "HP-GNN: Generating high throughput GNN training implementation on CPU-FPGA heterogeneous platform," in *Proc. ACM/SIGDA Int. Symp. Field- Program. Gate Arrays*, 2022, pp. 123–133.

[11] Z. Que et al., "Ll-GNN: Low latency graph neural networks on FPGAs for particle detectors," 2022, *arXiv:2209.14065*.
[12] Y. C. Lin, B. Zhang, and V. Prasanna, "GCN inference acceleration using high-level synthesis," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2021, pp. 1–6.
[13] P. Chen, P. Manjunath, S. Wijeratne, B. Zhang, and V. Prasanna, "Exploiting on-chip heterogeneity of versal architecture for GNN inference acceleration," in *Proc. 33rd Int. Conf. Field- Program. Log. Appl.*, 2023, pp. 219–227.
[14] S. Gandhi and A. P. Iyer, "P3: Distributed deep graph learning at scale," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 551–568.
[15] D. Zheng et al., "DistDGL: Distributed graph neural network training for billion-scale graphs," in *Proc. IEEE/ACM Workshop Irregular Appl.: Architectures Algorithms*, 2020, pp. 36–44.
[16] Z. Lin, C. Li, Y. Miao, Y. Liu, and Y. Xu, "PaGraph: Scaling GNN training on large graphs via computation-aware caching," in *Proc. 11th ACM Symp. Cloud Comput.*, 2020, pp. 401–415.
[17] Y.-C. Lin and V. Prasanna, "Hyscale-GNN: A scalable hybrid GNN training system on single-node heterogeneous architecture," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2023, pp. 557–567.
[18] M. Yan et al., "HyGCN: A GCN accelerator with hybrid architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 15–19.
[19] S. Li et al., "Hyperscale FPGA-as-a-service architecture for large-scale distributed graph neural network," in *Proc. Proc. 49th Annu. Int. Symp. Comput. Archit.*, 2022, pp. 946–961.
[20] H. Zhang et al., "Understanding GNN computational graph: A coordinated computation, IO, and memory perspective," in *Proc. Mach. Learn. Syst.*, 2022, pp. 467–484.
[21] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *Proc. ICLR Workshop Representation Learn. Graphs Manifolds*, 2019.
[22] J. Chen, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," in *Proc. Int. Conf. Learn. Representations Workshop*, 2016.
[23] T. Geng et al., "FPDeep: Acceleration and load balancing of CNN training on FPGA clusters," in *Proc. IEEE 26th Annu. Int. Symp. Field- Program. Custom Comput. Mach.*, 2018, pp. 81–84.
[24] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of CNN training on deeply-pipelined FPGA clusters with work and weight load balancing," in *Proc. Int. Conf. Field-Program. Log. Appl.*, 2018, pp. 394–3944.
[25] X. Zhang et al., "DNNExplorer: A framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator," in *Proc. Proc. 39th Int. Conf. Comput.-Aided Des.*, 2020, pp. 1–9.
[26] Y. Zhao, C. Li, Y. Wang, P. Xu, Y. Zhang, and Y. Lin, "DNN-chip predictor: An analytical performance predictor for DNN accelerators with various dataflows and hardware architectures," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2020, pp. 1593–1597.
[27] Y. Yu, Y. Li, S. Che, N. K. Jha, and W. Zhang, "Software-defined design space exploration for an efficient DNN accelerator architecture," *IEEE Trans. Comput.*, vol. 70, no. 1, pp. 45–56, Jan. 2021.
[28] B. Zhang, S. R. Kuppannagari, R. Kannan, and V. Prasanna, "Efficient neighbor-sampling-based GNN training on CPU-FPGA heterogeneous platform," in *Proc. IEEE High Perform. Extreme Comput. Conf.*, 2021, pp. 1–7.
[29] J. R. Stevens, D. Das, S. Avancha, B. Kaul, and A. Raghunathan, "GNNerator: A hardware/software framework for accelerating graph neural networks," in *Proc. 58th ACM/IEEE Des. Automat. Conf.*, 2021, pp. 955–960.
[30] T. Geng et al., "I-GCN: A graph convolutional network accelerator with runtime locality enhancement through islandization," in *Proc. MICRO-54: 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2021, pp. 1051–1063.
[31] Y.-C. Lin, B. Zhang, and V. Prasanna, "Accelerating GNN training on cpu multi-FPGA heterogeneous platform," in *High Performance Computing*. Berlin, Germany: Springer, 2022.
[32] W. Hu et al., "Open graph benchmark: Datasets for machine learning on graphs," 2020, *arXiv: 2005.00687*.
[33] Z. Cai, X. Yan, Y. Wu, K. Ma, J. Cheng, and F. Yu, "DGCL: An efficient communication library for distributed GNN training," in *Proc. 16th Eur. Conf. Comput. Syst.*, 2021, pp. 130–144.
[34] Y.-K. Choi, J. Cong, Z. Fang, Y. Hao, G. Reinman, and P. Wei, "In-depth analysis on microarchitectures of modern heterogeneous CPU-FPGA platforms," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, pp. 1–20, 2019.
[35] "Amazon EC2 F1 [online]," Accessed: Jun. 23, 2022. [Online]. Available: https://aws.amazon.com/tw/ec2/instance-types/f1/
[36] "Azure np-series [online]," Accessed: Jun. 23, 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/virtual-machines/np-series
[37] "Powertop [online]," Accessed: Jun. 23, 2022. [Online]. Available: https://github.com/fenrus75/powertop
[38] "Nvidia system management interface [online]," Sep. 05, 2023. [Online]. Available: https://developer.nvidia.com/nvidia-system-management-interface
[39] V. Kathail, "Xilinx vitis unified software platform," in *Proc. ACM/SIGDA Int. Symp. Field- Program. Gate Arrays*, 2020, pp. 173–174.
[40] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. Int. Conf. Learn. Representations*, 2017.
[41] S. Pandey, L. Li, A. Hoisie, X. S. Li, and H. Liu, "C-SAW: A framework for graph sampling and random walk on GPUs," in *Proc. SC20: Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2020, pp. 1–15.

Yi-Chien Lin (Graduate Student Member, IEEE) received the BS degree in electrical engineering from the National Taiwan University, in 2020. He is currently working toward the PhD degree in electrical engineering with the University of Southern California. He is a recipient of the Viterbi School of Engineering Graduate School Fellowship at the Ming Hsieh Department of Electrical and Computer Engineering. His research interests include Machine Learning systems, hardware acceleration, and graph machine learning.

Bingyi Zhang received the BS degree in microelectronics from Fudan University, in 2017, and the MS degree in Integrated Circuit Engineering from Fudan University. He is currently working toward the PhD degree in computer engineering with the University of Southern California (USC). His research interests include parallel computing, digital signal processing, digital circuit design. He focuses on accelerating graph-based machine learning on FPGA platform.

Viktor K. Prasanna received the BS degree in electronics engineering from Bangalore University, the MS degree from the School of Automation, Indian Institute of Science, and the PhD degree in computer science from Pennsylvania State University. He is Charles Lee Powell chair in engineering in the Ming Hsieh Department of Electrical and Computer Engineering and Professor of computer science with the University of Southern California (USC). His research interests include high performance computing, parallel and distributed systems, reconfigurable computing, and embedded systems. He serves as the director of the Center for Energy Informatics at USC. He served as the editor-in-chief of the IEEE Transactions on Computers during 2003–2006. Currently, he is the editor-in-chief of the Journal of Parallel and Distributed Computing. He was the founding chair of the IEEE Computer Society Technical Committee on Parallel Processing. He is the steering chair of the IEEE International Parallel and Distributed Processing Symposium (IPDPS) and is the steering chair of the IEEE International Conference on High Performance Computing (HiPC). He received the 2009 Outstanding Engineering Alumnus Award from the Pennsylvania State University, the 2019 Distinguished Alumnus Award from the Indian Institute of Science (IISc) and the 2016 Distinguished Alumnus Award from the University Visvesvaraya College of Engineering (UVCE), Bangalore University. He received the W. Wallace McDowell Award from the IEEE Computer Society, in 2015 for his contributions to reconfigurable computing. He is a fellow of the IEEE, the ACM, and the American Association for Advancement of Science (AAAS). He is an elected member of Academia Europaea.