EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks

Shengwen Liang, Ying Wang, *Member, IEEE,* Cheng Liu, Lei He, Huawei Li, *Senior Member, IEEE,* Dawen Xu, and, Xiaowei Li, *Senior Member, IEEE*

Abstract—Graph neural networks (GNNs) emerge as a powerful approach to process non-euclidean data structures and have been proved powerful in various application domains such as social networks and e-commerce. While such graph data maintained in real-world systems can be extremely large and sparse, thus employing GNNs to deal with them requires substantial computational and memory overhead, which induces considerable energy and resource cost on CPUs and GPUs. In this work, we present a specialized accelerator architecture, EnGN, to enable high-throughput and energy-efficient processing of large-scale GNNs. The proposed EnGN is designed to accelerate the three key stages of GNN propagation, which is abstracted as common computing patterns shared by typical GNNs. To support the key stages simultaneously, we propose the ring-edge-reduce(RER) dataflow that tames the poor locality of sparsely-and-randomly connected vertices, and the RER PE-array to practice RER dataflow. In addition, we utilize a graph tiling strategy to fit large graphs into EnGN and make good use of the hierarchical on-chip buffers through adaptive computation reordering and tile scheduling. Overall, EnGN achieves performance speedup by 1802.9X, 19.75X, and 2.97X and energy efficiency by 1326.35X, 304.43X, and 6.2X on average compared to CPU, GPU, and a state-of-the-art GCN accelerator HyGCN, respectively.

Index Terms—Graph neural network, accelerator architecture, hardware acceleration.

1 Introduction

Recently, the success of deep learning methods in many fields has provoked a keen interest in generalizing neural network architectures to non-Euclidean data, such as manifolds and graphs. However, traditional deep neural networks, such as convolutional neural network (CNN) [1], long short term memory (LSTM), are proposed to work for regular grid-like structures in Euclidean space, they are not trivially portable to non-Euclidean data domains like graphs. Therefore, graph neural networks (GNNs) are recently emerging as a powerful approach for graph processing and achieving unparalleled performance on many classic graph processing tasks, such as citation network [2], social networks [3], and knowledge graph [4]. The success of graph neural networks propelled the deployment of GNNs to the real-world production system. For example, Alibaba's AliGraph [5] and Euler [6] platform leverage GNNs to analyze the e-commerce graph data of billion users and items.

The prosperity of GNNs is enabling the development of emerging AI applications and systems that require high-throughput and low-latency processing capability. For instance, a recommendation system in Taobao [6] that leverages GNNs to mine billion-scale e-commerce data needs to perform real-time recommendations to millions of customers shopping at the same time. Therefore, to ease GNN model development and deployment, some high-performance GNN processing frameworks, such as Deep Graph Library (DGL) [7], Pytorch Geometric (PyG) [8], and Neugraph [9] have been developed because the existing deep learning frameworks and graph processing frameworks cannot fulfill large graph-based neural networks [9]. However, the potential performance and energy efficiency of GNNs are still bounded by the hardware architectures assumed by these frameworks. The major drawbacks are attributed to three-fold factors. First, compared to DNNs with regular computing patterns, GNNs inherit both the irregular processing dataflow of graph analytic and the regular computing pattern of DNNs. This hybrid computing pattern that involves large amount of dynamic and irregular data accesses results in the inefficiency of the CPU and GPU. Second, a real-world graph can be extremely huge. For instance, the e-commerce graphs in Alibaba contain billions of nodes and hundreds of billion edges with rich attribute information. Some GNN software frameworks generally adopt a large number of compute nodes equipped with multiple CPUs or GPUs to deal with large-scale graphs, thus it results in high cost and energy overhead. For example, NeuGraph uses eight GPUs to handle a dataset with million vertices [9]. Third, the power-law distribution of the big real-world graph challenges the existing memory hierarchy and caching policy of CPUs and GPUs, for the sparsely distributed low-degree vertices in the graphs make it hard to reuse the graph data in general-purpose processors.

Intuitively, specialized hardware architecture is a promising option to improve the efficiency of GNN. However, previous graph processors and neural network accelerators are optimized to support either graph processing or neural networks, rather than both of them simultaneously. To address this problem, prior work proposed HyGCN to combine the graph processing and neural network processing in a specified hardware architecture. However, HyGCN

- *The authors are with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the University of Chinese Academy of Sciences, Beijing 100190, China. E-mail:{liangshengwen, wangying2009, liucheng, helei19g, lihuawei, lxw}@ict.ac.cn, xudawen@hfut.edu.cn*

Manuscript received April XX, 2020; revised August XX, 2020.

mainly targets at graph convolution network (GCN) and utilizes a systolic array to perform neural network computation operation inside GCN, which is the target workload evaluated in their work, and it is not designed to run general GNN architectures like graph recurrent network, graph attention network, etc. This is because the systolic array adopted by HyGCN is subject to low resource utilization when handling GRN with GRU or LSTM unit. In addition, the inherent nature of the real-world large-scale graph adopted by the GNN model, such as power-law distribution, variable feature length of vertex, significantly impacts the performance of GNN model and it leaves a large potential space to optimize the data locality, on-chip memory hierarchy, task partitioning, and scheduling. Nevertheless, HyGCN does not consider such inherent features of the graph, which dramatically limits the achieved performance and energy efficiency.

Therefore, in order to solve the aforementioned issues and accelerate practical GNN-based applications that process real-world large-scale graphs, we propose EnGN, a high-throughput and energy-efficient edge-centric accelerator for large graph neural network processing. However, designing such an accelerator is a non-trivial task and has to resolve the obstacles that exist in the real-world GNN algorithms: (1) How to tailor a unified architecture that efficiently supports the diverse GNN models and flows not limited to GCNs. It is observed that the dataflow and the dimension of the working-set, e.g., the vertex, dynamically changes in wide ranges during the propagation of different GNN layers, requiring a reconfigurable architecture and interconnects to avoid hardware and memory bandwidth under-utility. (2) large graphs containing millions of vertices pose a significant challenge to the design of energy-efficient and compact GNN accelerators with limited on-chip memory space. Particularly, when massive graphs with million vertices are partitioned into sparsely-connected sub-graphs, there will be intensive random and irregular off-chip memory accesses induced, which leads to poor locality that are hard to harness in the aggregate and update stage. And (3) the power-law distribution [10] creates high-degree but imbalanced connection sparsity in large real-world graphs. Accelerator must be able to deal with the imbalanced sparsity distribution, which leads to processing elements under-utility, poor locality, and redundant memory access issues in hardware.

To cope with issues, first, by observing state-of-the-art GNN processing frameworks such as DGL and PyG, we generalize the architecture of typical GNN algorithms into three key stages: the vertex feature extraction stage, the feature aggregate stage, and the graph update stage. In response to the three key stages abstracted from general GNN frameworks, we support the corresponding computing patterns in EnGN, so that it is a general GNN processor and able to support most of the GNN architectures such as GCN, GRN, etc. In EnGN, a ring-edge-reduce (RER) dataflow and the accompanied hardware architecture of RER processing elements (PEs) arrays are designed to simultaneously conduct the stages of vertex property feature extraction, aggregate, and vertex update on GNNs. It is known that aggregating the property and updating the vertices distributed in the large but sparse graphs will lead to poor hardware resources and memory bandwidth utilization due to poor data locality of vertices and edges. However, the proposed RER PEs connected into a ring topology leverages the RER dataflow to make vertex property flow between rows of PEs and performs efficient update operations without randomly accessing the vertices and edges from the memory.

Second, for the feature extraction stage, EnGN constructs a graph property aware dataflow (GPA) that decouples the vertex property and the hardware structure, which makes the GNN mapping to the RER array independent of the vertex dimension. In addition, we observe that the computational overhead of GNN models is sensitive to the vertex-property dimension and also the order of the GNN processing stages. Based on this observation, EnGN is designed to enable the processing reordering based on the model architecture such that the overhead of the GNN inference can be reduced and higher performance can be achieved.

Third, considering the footprint of large-scale graphs, EnGN adopts a graph tiling strategy to process the partitioned sub-graphs with high degree of data reusability. Graph tiling aims to partition a large-scale graph into sub-graphs that fit the on-chip memory and maximize the locality. The tiles are strategically scheduled in EnGN to select either row-oriented or column-oriented processing dataflow to maximally reuse vertices between tiles and reduce the overhead caused by the off-chip memory access.

Finally, due to the power-law distribution and sparsity characteristics of the real-world graphs, the accessing frequency to different vertices may vary in a large scale. For example, on Cora citation graph [2], the access frequency of a high-degree vertex is 100X than that of a low-degree vertex, which causes access imbalance issue. Thus, EnGN comprises a three-level on-chip memory hierarchy, and the L2 memory is a degree-aware vertex cache (DAVC) to locally cache the high-radix vertices that are densely connected to other vertices in graphs. DAVC reduces considerable memory access cost. In summary, our main contributions are the following:

1) A compact but high-throughput accelerator is designed for large graph neural network, which is implemented based on the edge-centric paradigm and supports various large scale GNNs.
2) We proposed a graph property aware and ring-edge-reduce (RER) dataflow to enable the EnGN to handle a vertex with arbitrary dimension property and high throughput GNN operations. The on-chip memory hierarchy is designed to be aware of the non-uniform distribution of high-radix and low-radix graph vertexes and employ a specialized memory space management to enhance data locality on the chip.
3) We implement the EnGN accelerator in 14nm process and make comprehensive evaluations and compare the performance, power, energy of EnGN to CPU, GPU, and HyGCN baselines. Experimental results show that, compared to CPU and GPU, EnGN achieves on average 1802.9X speedup with 1326.35X energy reduction and 19.75X speedup with 304.43X energy reduction, respectively. The speedup and energy efficiency of EnGN is shown to be 2.97X and 6.2X higher than HyGCN, which is a contemporary work of EnGN on GNN accelerator.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3014632, IEEE Transactions on Computers

TABLE 1: GNN algorithms on EnGN processing model.

Algorithms	Feature extraction	Aggregate	Update
GCN	$h_u^l * V_{degree}^{-1/2}$	$V_{temp}^l = acculumate(Res)$	$ReLu(W^l V_{temp}^l)$
GS-Pool	$ReLu(W_{pool}^l h_u^l + b)$	$V_{temp}^l = max(Res)$	$ReLu(W^l concat(V_{temp}^l, h_v^l))$
Gated-GCN	$Sigmoid\ (W_H^l h_v^l + W_C^l h_u^l)\ \odot h_u^l$	$V_{temp}^l = acculumate(Res)$	$ReLu\ (W^l\ V_{temp}^l)$
GRN	h_u^l	$V_{temp}^l = acculumate(Res)$	$GRU(h_v^{(l)},\ W^l\ V_{temp}^l)$
R-GCN	$h_{r,u}^l * V_{degree}^{-1/2}$	$V_{r,temp}^l = acculumate(Res)$	$ReLu(\sum_{r\in R} W_r^l V_{r,temp}^l)$

TABLE 2: Notations.

Notations	Descriptions	Notations	Descriptions
G	Graph $G = (V, E)$	I_N	Identity matrix
V	Vertices of G	$\tilde{D}_{ii}$	$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$
E	Edges of G	$\tilde{A}$	$\tilde{A} = A + I_N$
A	Adjacency matrix	W^l	Weights at layer l
X	Vertices property of G	b	Bias vectors
h_v^l	Source vertex v property at layer l	$Concate()$	Concatenate function
h_u^l	Destination vertex u property at layer l	N_v	Neighbor set of vertex v

2 General GNN processing model

2.1 Graph neural networks

Unlike CNNs that mainly deal with Euclidean data like images and videos [9], graph neural networks (GNNs) generalize the CNN to operate directly on non-Euclidean data especially graph data such as social networks and chemical molecules. It has been proven to be supremely successful on tasks like node classification, graph classification, and link prediction. Motivated by the success of GNNs, various GNN architectures have been proposed recently [11], [12]. Table 2 lists the notations used in this paper.

Graph convolution network (GCN) generalizes the convolution operation from regular image data to nonstructural graph data. It can be used for node classification [2] and chemistry molecules architecture analysis [13]. A typical GCN [2] is presented and formulated in Eq. 1:

$$h^{l+1} = ReLu(\tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}h^l W^l), h^0 = X \tag{1}$$

GraphSage-Pool (GS-Pool) is proposed in [14] and used for citation network analysis and protein-protein interaction task. Unlike the GCN models, it leverages the averaging function as an aggregation operator and has the source vertex property (h_v^l) involved when updating output in next iteration. The expression of GS-Pool is defined in Eq. 2.

$$h_v^{l+1} = ReLu(W^l concat(ReLu(W_{pool}^l h_u^l + b)), h_v^l) \tag{2}$$

Gated graph convolution network (Gated-GCN) is proposed in [15] and utilized for community detection. It borrows the idea from gate recurrent neural networks and constructs a propagation function that receives and processes the property of source and destination vertex simultaneously. The propagation function is depicted in Eq. 3.

$$\begin{aligned} h_v^{(l+1)} =&\ Relu\ (W^l(\sum_{u\in N(v)} \eta_{uv} \odot h_u^l) \\ \eta_{uv} =&\ Sigmoid\ (W_H^l h_v^l + W_C^l h_u^l) \end{aligned} \tag{3}$$

where $\odot$ refers to element-wise multiplication, $ReLu(\cdot)$ and $sigmoid(\cdot)$ are typical nonlinear activation functions that have been widely adopted in CNNs [1].

Graph Recurrent network (GRN) is similar to the recurrent neural network (RNN), but aims to learn vertex representations [16]. GRN is mostly used in NLP tasks, traffic forecasting, etc. For example, [17] integrates typical RNN units (Gated recurrent unit) into the propagation function as formulated in Eq. 4 to perform graph learning tasks.

$$h_v^{(l+1)} = GRU(h_v^{(l)},\ \sum_{u\in N(v)} W^l h_u^l) \tag{4}$$

Relational graph convolutional network (R-GCN) is an extension of GCN and used to handle graphs with different edge types. For instance, the edges can be used to represent different relations and have distinct weights definition of W_r^l [18]. Similar to GCN, hidden representation of entities in the $(l+1)^{th}$ layer in R-GCN can be formulated in Eq. 5:

$$h_i^{l+1} = \sigma(W_0^l h_i^l + \sum_{r\in R}\sum_{j\in N_i^r} \frac{1}{c_{i,r}} W_r^l h_j^l) \tag{5}$$

where N_i^r denotes the set of neighbor indices of node i under relation $r \in R$ and $c_{i,r}$ is a normalization constant. $c_{i,r} = |N_i^r|$ is used in prior entity classification work [18].

Although GNN algorithms are different in terms of architecture and target applications, we notice that they share common computing patterns. 1) GNNs initially condense vertex property of source vertex with learned parameters to obtain more compact feature representations. 2) Afterwards, GNNs usually gather neighbor properties to embed the information of graph topology to the extracted features. and 3) GNNs usually leverage learned parameters to condense the output features obtained in the aggregate stage making GNN capable to learn and perform more complex tasks. GNN accelerators must be able to support the computation abstractions concluded above, in order to support different GNN architectures efficiently.

Algorithm 1 EnGN processing model

```
Input: Graph $G = (V, E)$, Vertex property $Prop$ and $Tmp_{prop}$, layer $l$, learned
    parameter $W_{feature}$, $W_{update}$
Output: Vertex Property $Result$
for $l \leftarrow 1$ to $l_{max}$ do
    for each edge $e \in Edge$ do
        $tmp \leftarrow$ Feature extraction$(Prop[e.src], Prop[e.dst], W_{feat.})$
        $Tmp_{prop}[e.dst] \leftarrow$ Aggregate$(Tmp_{prop}[e.dst], tmp)$
    end for
    for each edge $e \in Edge$ do
        $Prop[e.dst] \leftarrow$ Update$(Prop[e.dst], Tmp_{prop}[e.dst], W_{update})$
    end for
end for
$Result \leftarrow Prop$
```

2.2 EnGN processing model

According to the goal of the key stages in a typical GNN, the common computing patterns can be abstracted as *feature extraction*, *aggregate*, and *update*. The *feature extraction* stage condenses the property of each vertex in the graph using a neural network. The *aggregate* stage embeds the graph topology into local vertex property by accumulating each vertex's neighbor properties generated in the feature extraction. The choices of aggregate functions include various arithmetic operations such as max, min, and add to produce unified output features. At the end of propagation iteration, the *update* stage leverages learned parameters to further condense the output features obtained in the aggregate stage, then applied a non-linear activation function or GRU/LSTM function to each vertex of the graph before output. Note that when the aggregate stage includes only linear operation, it can be scheduled before or after the feature extraction stage.

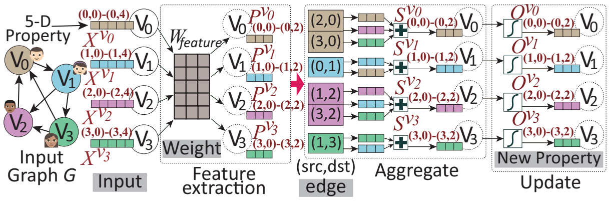

Fig. 1: GCN on EnGN processing model.

It also provides an opportunity for EnGN to dynamically adjust the stages of matrix operations to optimize EnGN performance, which will be introduced in section 5. On top of the abstraction, we propose a unified EnGN processing model that can cover general GNN models using the common computing functions as shown in Algorithm 1. Suppose the graph is represented as $G(V, E)$ where V and E represent the set of vertices and edges in the graph respectively. $Property$ is the set of vertex property of the graph. By default, the input graph is stored as a coordinate list (COO). Each edge in the graph is a tuple (src, dst, val), where val usually stands for the edge property and it depends on graph definition. The EnGN execution flow follows the neighborhood aggregation strategy, which iteratively updates the representation of vertices by aggregating representations of their neighbors. Since all the vertices in the graph will be processed in each iteration for GNN algorithms, EnGN is presented as an edge-centric processing model to ensure more efficient memory accesses [19].

For each edge, both the source vertex property and the destination vertex property are condensed with $W_{feature}$ using $feature\_extraction(\cdot)$ to obtain a temporary property tmp. Then tmp is added to the destination property using $aggregate(\cdot)$ function. Since there may be multiple edges that are incident to the same destination vertices, $aggregate(\cdot)$ is essentially a reduce function. When all the destination vertices are reduced, an activation function or the user-defined operator with learnable weights W_{update} are used to filter the output using $update(\cdot)$ function.

To help understand the EnGN execution model, we present a vivid example of GCN [2] processed by the EnGN architecture as shown in Fig. 1. Suppose an input social network graph G has four vertices (users) and its edges represent the relation between users. Each vertex (user) attaches a 5-dimensions property (embedding vector) which is a learning representation of user information such as age and gender. Dimension usually stands for the length of the embedded user property. The input property of the vertices are denoted as X^{v_0}, X^{v_1}, X^{v_2}, X^{v_3}. In $feature\_extraction(\cdot)$ function, the feature extraction function takes both the vertex property, i.e., X^{v_0}, X^{v_1}, X^{v_2}, X^{v_3} and associated weight matrix $W_{feature}$ as input. Then it has the weight matrix multiplied with the high-dimension input vertex property to generate low-dimension temp features. Note that the size of the weight matrix is associated with both the input property dimension and output temp feature dimension. In this example, the size of the weight matrix is 5×3. With the feature extraction function, the input vertex properties are converted to 3-dimension temp features donated as P^{v_0}, P^{v_1}, P^{v_2}, P^{v_3}. In $aggregate(\cdot)$ function, it receives the results of $feature\_extraction$ function and aggregates the property of each vertex's incoming neighbors. As shown in Fig. 1, the temp properties of vertex 2 and 3, i.e., P^{v_2}, P^{v_3} are added to temp property of vertex 0 as vertex 2 and 3 are incoming neighbors of vertex 0 P^{v_0} according to the graph topology. When the aggregation stage is done, $update(\cdot)$ starts. It has the vertex features, i.e., S^{v_0}, S^{v_1}, S^{v_2}, S^{v_3} filtered using an activation function. The filtered output properties denoted as O^{v_0}, O^{v_1}, O^{v_2}, O^{v_3} become the input to the next iteration.

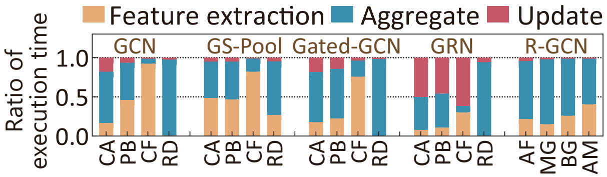

Fig. 2: Execution time breakdown of GNN models.

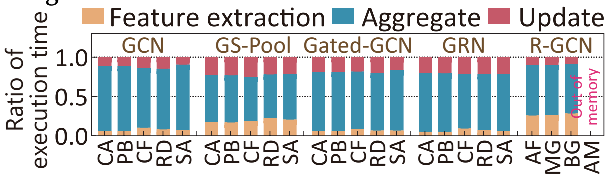

Fig. 3: Execution time breakdown of GNN models on GPU.

Similar to the GCNs, we also have the rest of the typical GNN algorithms mentioned in section 2 mapped to the EnGN processing model. Table 1 summarizes the resulted EnGN processing functions.

3 Motivation

3.1 Workload characterization

To gain insight into the computing characteristics of GNN processing models, we leverage a state-of-the-art GNN software framework, DGL, to analyze the five aforementioned GNN models on Intel Xeon CPU and NVIDIA V100 GPU. Fig. 2 and Fig. 3 shows the execution time breakdown of GCN, GS-Pool, Gated-GCN, GRN, and R-GCN on the datasets that are selected from Table 6. Note that GCN, GS-Pool, Gated-GCN, and GRN executed on datasets of CA, PB, CF, RD, and SA (GPU) while R-GCN is mainly used in the knowledge graph and it works on open datasets including AF, MG, KG, and AM. In general, it can be observed that the three processing stages including feature extraction, aggregate, and update take up a distinct proportion of the execution time on different datasets. Thereby, all the processing stages must be taken into consideration for general GNN acceleration which remains a great design challenge. On the other hand, we observe that the aggregate stage that requires computing and traverse of the graph data involves considerable irregular memory accesses and consumes a large portion of the total execution time on datasets of CA, PB, and RD for algorithms of GCN, GS-Pool, and Gated-GCN. Particularly, the aggregate stage of R-GCN on all the datasets turns out to be the most time-consuming stage. To further investigate the reasons for the processing inefficiencies of the aggregate processing stage, we analyze the statistics of the CPU processing system executing GNNs as listed in Table 3. The results reveal that the aggregate stage has the lowest instructions per cycle (IPC) due to the much higher cache miss rate and memory bandwidth requirements, which are mostly incurred by the intensive irregular memory accesses. According to the I/O to computing ratio metric, i.e., memory accesses per operation in the table, we also confirm that the aggregate stage involves

TABLE 3: Execution pattern of GCN on Cora dataset.

	Feature extraction	Aggregate	Update
IPC	1.73	**0.77**	1.01
L3 cache miss ratio	56.60	**82.62**	46.47
CPU stalls caused by memory loads	15.16	**40.8**	30.15
DRAM Bytes pre Ops	0.24	**11.1**	0.41

intensive memory accesses per operation. In a nutshell, the aggregate stage closely relevant to the irregular graph is the most critical part of GNN processing in most cases. It is an IO-bound task and must be optimized sufficiently for high-performance GNN processing.

As GNNs operate on large attributed graphs and the graph structures including input feature dimension, output feature dimension (corresponding to GNN architecture) affects the execution dramatically, we take GCN as an example and further evaluate how these graph features affect the execution time of GNN. Note that a synthetic graph that can be scaled for the evaluation is utilized in the experiment. The experimental result is presented in Fig. 4. It reveals that the GNN execution time increases with both larger input feature dimension and output feature dimension while the execution time is more sensitive to the input feature dimension. For instance, when input feature dimension changes from 64 to 1024, the execution time increases by 2.21X. However, it increases by only 1.32X when the output feature goes up from 64 to 1024. Meanwhile, we observe that the graph convolution operation can be symmetric in aggregate with sum operator cases and we may exchange the input feature dimension and output feature dimension. The proof will be detailed in section 5. With these observations, we may improve the computing efficiency by exchanging the input and output features without affecting the GNN processing.

3.2 Hardware architecture for GNNs

The state-of-the-art graph learning frameworks such as DGL, PyG essentially rely on general purposed processors (GPPs), i.e., CPU and GPU. Nevertheless, GPPs especially GPUs fail to take advantage of a large amount of parallel processing engines on GNNs that involve a large amount of irregular traverse and computing over large sparse graphs. As a result, GPUs suffer workload imbalance, memory divergence, and branch divergence for GNN processing [20], [21], [22]. Unlike GPPs, specialized hardware accelerators promise to offer energy-efficient processing for a specific domain of applications such as neural networks and graph processing. Nevertheless, neither neural network accelerators nor graph processing accelerators can process GNNs with combined neural network processing (feature extraction) and graph processing (aggregate and update). In addition, even for the graph processing part, existing graph processing accelerators assume simple graph structure with fixed scalar feature while the graphs utilized in GNNs usually have much more complex attributes and the attributes change across the different GNN layers, which poses more pressure to the on-chip data buffering and memory access optimization.

Instead of reusing existing hardware architectures for graph convolution network, the authors proposed HyGCN, a specialized accelerator for GCN processing. They take the hybrid computing pattern of GCN [21] into consideration and have separate processing modules for the neural network processing and graph-like computing respectively. Nevertheless, HyGCN still fails to unleash the potential of the GNN acceleration on a few aspects.

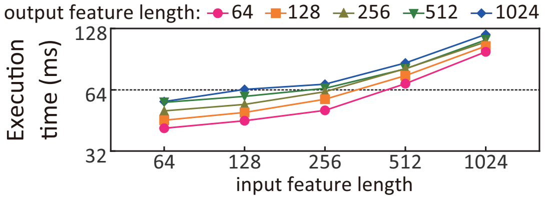

Fig. 4: Execution time of GCN model on graph with 0.25M vertices and 0.96M edges w.r.t input/output feature length.

First of all, HyGCN lacks optimization for the irregular memory accesses caused by the large sparse attributed graph, which plays a key role in general GNN processing. For instance, many large graphs extracted from social networks are highly skewed [20]. The analysis of datasets used by this work indicates the top 20% vertices with higher degree are connected to the 50-85% edges of the whole graph. The vertex property of these high-degree vertices are more likely to be reused during the aggregation. In contrast, the low-degree vertices are less probably to be reused. These skewed vertices are equally buffered in HyGCN, which can lead to frequent data movement between the on-chip buffer and the external DRAM. While the feature dimension is usually large in GNNs, this further deteriorates the memory access efficiency of HyGCN.

Secondly, HyGCN has separate the modules for the regular neural network processing part and irregular graph processing part. Accordingly, they need independent on-chip buffers which consume considerable chip area. Although they can be pipelined, the imbalanced computing of the different processing stages as shown in the prior subsection makes it difficult to make use of both modules for general GNN processing efficiently. We argue that a unified hardware design that can reuse the limited on-chip buffer among the different processing stages can provide more energy-efficient GNN processing.

In summary, GNNs that combine both neural network processing and graph-like processing can be computing bound and memory bound. Particularly, the processing bottleneck changes with the GNN algorithms and targeted graphs, which makes general GNN accelerator design rather challenge. The unique combined computing features also make GNN processing inefficient on GPPs and hinders us to reuse the existing DNN accelerators and graph processing accelerators. The state-of-the-art accelerator HyGCN which focuses on GCN acceleration still fails to consider the influence of the large sparse graphs on GNN processing sufficiently and to unleash the potential of GNN acceleration. This motivates us to concentrate on the memory access optimization for energy-efficient general GNN processing in this work.

4 EnGN Architecture

4.1 EnGN hardware architecture

On top of the unified EnGN processing model, we develop a customized EnGN accelerator as shown in Fig. 5. It only focuses on the GNN inference and adopts 32-bit fixed point to maintain the accuracy of GNN inference. A neural graph

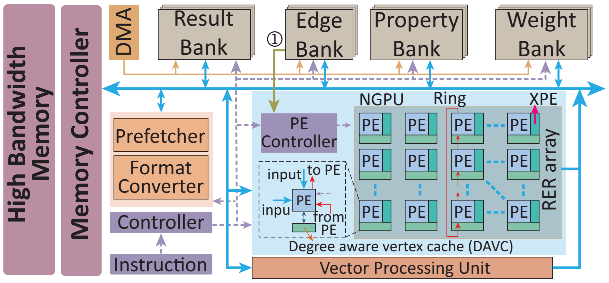

Fig. 5: EnGN hardware architecture.

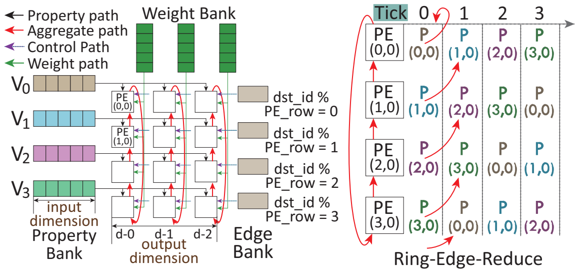

Fig. 6: Architecture details.

processing unit (NGPU) is integrated to perform Feature extraction, Aggregate, and Update operation in a unified architecture. It has an array of homogeneous processing elements (PE) and the array size is 128 × 16. Each PE unit contains a local register file to store the temporary results and acts as intermediate for inter-PE communication. Each PE in the same column of the Ring-Edge-Reduce (RER) array is connected to its neighbors in a ring network to perform aggregate operation and each PE in the same row of the RER array can process a vertex property, which means the NGPU can process 128 vertices simultaneously. However, such processing parallelism requires substantial memory bandwidth. Thereby, to avoid performance degradation, EnGN optimizes the memory access patterns for vertex data and edge data moving. For source vertex data access in the large graph, we adopt the graph tiling technique and ensure that the source vertex fetching only induces accesses to the continuous memory addresses. For random destination vertex accesses in the aggregate and update stage, EnGN leverages the hashed edge data layout and multi-level cache method to avoid write conflicts and improve data hit rate in the compact on-chip buffer. During processing, the edge parser of NGPU reads the edge list of the graph from the edge banks and parses it into bit-stream that controls the PE-array to perform inter-row aggregate operation (① in Fig. 5). The hardware modules are controlled by the signals decoded from the EnGn instructions. Each coarse-grained instruction is responsible for a specific processing function such as feature extraction and data movement operations. Meanwhile, the instruction also contains hardware-relevant parameters for the processing functions such as the tiling sizes, feature dimension, and data starting addresses in on-chip buffers or the external memory. Since the parameters of different instructions vary, the instructions are variable-length but aligned to 64bit. The instructions are generated with an offline GNN compiler specifically for the EnGN accelerator and the sequence of the instructions determines the processing order of the GNNs on EnGN. In addition, each PE in the NGPU is attached by an XPE to perform activation functions, bias operation, and rounding operation in the GNN processing stage. A vector processing unit (VPU) is used to deal with different feature extraction, aggregate, and update functions of GNNs illustrated in Table 1. Two auxiliary modules: Prefetcher and Format converter, are used to assist the memory accesses and improve the input graph format compatibility respectively.

4.1.1 The RER PE array

The feature extraction stage maps the high dimensions property of vertices to the low dimensions by using the learned weight matrix, and this stage is simply matrix multiplication operation. As shown in Fig. 6, in order to handle the arbitrary-dimension property of GNN algorithms, we propose the graph property aware (GPA) dataflow to decouple the input property of the vertex and the hardware computing structure. In this manner, each PE in the same column of PE-array is responsible for a single dimension of vertex property and each PE in the same row handles a single vertex. The properties of a vertex are arranged in columns and aligned in the property bank. The dimensions of input vertex property become independent to the hardware architecture and can be continuously injected into the PE-array regardless of the array size and the property dimension. When the weight matrix has a column number larger than the size of the PE-array, we choose to split the weight matrix into partitions such that each partition match the size of PE-array. Note that the split weight matrices are placed in the weight banks in row-major order. After partitioning, the processing unit can handle vertex properties of arbitrary dimensions.

4.1.2 The RER topology for PE communication

The aggregate procedure needs to collect the information according to the edge information. Thereby, as shown in Fig. 6, each row of the PE-array in NGPU possesses a dedicated edge bank and each PE in the same row receives the same control signal parsed from edge list in the graph to gather the corresponding vertex property. Meanwhile, because each PE needs to broadcast its own vertex features generated by the feature extraction stage to all other PEs in the same column, aggregating the received information simultaneously can result in a large amount of hardware resource and power consumption. Thereby, inspired by the ring-all-reduce concept [23], we propose the ring-edge-reduce (RER) aggregate dataflow to conduct aggregate stage inside the PE array instead of moving the data out to the buffer. As shown in Fig. 6, because each column of PE performs the same operations without any communication in between, each PE in the same column of the array is connected to its neighbors through an on-chip network of ring topology. Each PE in the same column only communicates with its two nearest neighbors (north, south). In our design, the PE sends its data to the northern neighbors and receives the data sent from the southern neighbors for property aggregating. In this manner, a PE can select the relevant vertices to aggregate based on the control signal parsed from the edges during the data flow across the ring.

The RER dataflow makes the hardware design simple yet efficient when the graph is dense and the vertex properties that flow through the ring are mostly used for aggregation. However, many of the large graphs in practice are sparse

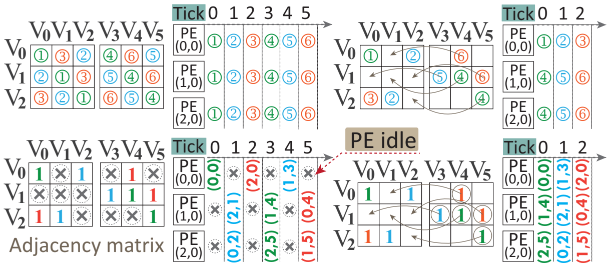

Fig. 7: Edge reorganization.

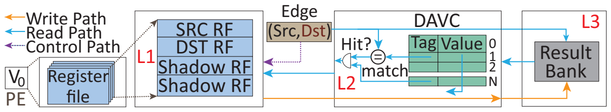

Fig. 8: Memory hierarchy.

and aggregation in PEs is inactive in many cases. A RER dataflow example on a sparse graph and the adjacency matrix is shown in Fig. 7. The computing array is assumed to be 3×3. In cycle 0, three edges from different edge banks will be fetched and the properties of V_0, V_1, and V_2 will flow across the ring at the same time. It takes the RER three cycles to complete the movement of the three vertex properties and the corresponding aggregation on V_0, V_1, and V_2. Similarly, it takes the RER another three cycles to repeatedly transfer the three vertex properties through the ring to aggregate on V_3, V_4, and V_5. Thereby, it takes the RER at least 6 cycles to perform the aggregate of the graph and many of the time slots are idle as marked with crosses in the figure.

To improve the efficiency of the aggregation, we further analyze the reason for the idle time slots. For example, PE(1, 0) is idle in Cycle 0 because the edge to be processed is $2 \rightarrow 1$ and it does not have the properties of vertex 2 yet. However, if it fetches the edge $1 \rightarrow 4$ first, it can perform the aggregate of vertex 4 using the property of vertex 1 at Cycle 0. With this observation, we propose to reorganize the edges in each edge bank to ensure the vertex properties flowing through the ring is used as much as possible. Fig. 7 exhibits the reorganized edges and the corresponding aggregation. With the edge reorganization, the aggregate completes in 3 cycles and the computing array is fully utilized. Basically, the order of the vertex properties flowing through the ring is known given the computing array. The required vertex property of each edge is also determined. Thereby, reorganizing the edges in each edge bank based on the order of the vertex properties flowing through the ring can maximize the aggregation efficiency of the computing array. The proposed edge reorganization result is depending on the structure of the input graph and PE array. It can be reused by different GNN algorithms targeting at the same graph structure and EnGN architecture. It is typically performed offline on CPUs and considered as a general approach of preprocessing widely applied in many graph computing applications. The preprocessing time lasts from several seconds to minutes once and for all, and it makes no impact on the on-line GNN processing performance.

4.2 The On-chip Memory Hierarchy

PE register file The register files (RF) equipped in the PEs are divided into four groups including source vertex groups(SRC RF), destination vertex groups (DST RF), and two shadow groups (Shadow RF), which is depicted in Fig. 8. The SRC RF stores the source vertex values generated in the feature extraction stage. The DST RF stores the destination vertex feature updated during the aggregate and update stages. In addition, there are two programmer-invisible Shadow RFs holding the SRC and DST vertex values previously generated by the PEs of the same column.

Multiple-level caches The real-world graph has up to millions of vertices. Although the graph tiling technique adopted by EnGN helps fit the sub-graphs into the on-chip buffer, the set of vertices in the sub-graphs will still outsize the register files of the PE array. Meanwhile, the result banks are used to store the temporary aggregate results. PE frequently accesses the long-latency result bank will result in performance degradation. Consequently, as shown in Fig. 8, we insert a degree aware vertex cache (DAVC) between the result banks and the register file of each PE to improve the performance of the EnGN. The register file, DAVC, and the result banks are regarded as the first, second, and last level memories on-chip, respectively. All capacity of DAVC is used to cache high-degree vertices. The reason will be illustrated in section 6. DAVC uses the destination vertex id of edges as the line tag to determine whether the access to the vertex data hit or not in the DAVC. If hit, the vertex data will be directly read to DST RF in the PE unit. Otherwise, EnGN will access the last-level result banks. In this manner, the DAVC can alleviate the overhead incurred by the result bank accesses.

5 ENGN OPTIMIZATION

5.1 Observations of GNN computing

To further optimize the EnGN design, we try to explore the characteristics of GNN algorithms and seek key observations that may guide the EnGN architecture optimization. Suppose the input graph $G = (V, E)$ with N vertices and E edges is depicted with an adjacency matrix $A \in \mathbb{R}^{N \times N}$. The vertex property of the graph is $X \in \mathbb{R}^{N \times F}$ with F channels and the learned filters, i.e., weight is $W \in \mathbb{R}^{F \times H}$ where H is output property dimension. Then, the output of the GNN, i.e., O can be represented as Eq. 6:

$$O = \sigma(A(XW)) \tag{6}$$

According to the formulation of GNNs, we obtain two major exploitable observations:

1. *The order of feature extraction processing and aggregate processing in GNNs are exchangeable when the operator in aggregate processing is sum.*

When the operator used in *aggregate* is *sum* which is widely adopted in GNN algorithms, the computing in Eq. 6 can be changed to Eq. 7 without affecting the result because of matrix multiplication associative law. While the amount of operations using the distinct computing order is also different, we may choose the order that incurs less computation in each iteration.

$$O = \sigma((AX)W) \tag{7}$$

2. *The weight size of GNNs is independent to the size of the input graph and it is usually small. While the input graphs can be large and typically dominate the memory accesses.*

According to Eq. 6, the weight size of GNNs is irrelevant to the number of vertices in the graph. In this case, the

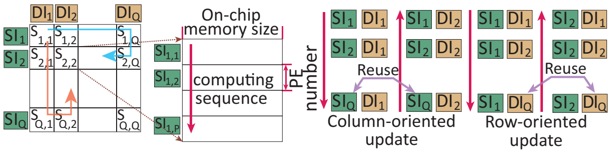

Fig. 9: Graph tiling and tile scheduling.

weight size can be much smaller compared to the graphs that may include millions of vertices, which is also a key distinction from CNNs. Input graphs will dominate the memory accesses and dealing with the large graphs in GNNs will be critical to the accelerator performance.

5.2 Dimension-aware stage re-ordering

According to Observation 1, the processing order of GNN stages, the feature extraction, aggregate, and update stages, will not affect the computing results, but it can change the total number of operations in a GNN. We analyze the quantity of operations when using different computing order, and aim to find the best way to schedule the stages. For *feature_extraction*, the number of operations, i.e., multiply-accumulate in Eq. 6 and Eq. 7 are the same and it is equal to $N \times F \times H$. Similarly, *update* does not change with the computing order. Nevertheless, for *aggregate*, the order of GNN computing leads to different number of operations, i.e., accumulation in *aggregate*. When Eq. 6 is used, the number of operations is $E \times F$. When Eq. 7 is chosen, the amount of operations becomes $E \times H$.

While the property dimension varies as observed in last subsection, F is not equal to H. To reduce the total computing, when the input vertex property dimension F is larger than output feature dimension H, we should choose Eq. 6 for GNN computing. Otherwise, we should use Eq. 7. Following this idea, we propose a dimension-aware stage reordering (DASR) strategy based on the input and output property dimension comparison. The DASR can be implemented by altering the instruction sequence that defines the computing order of GNNs, so it will not incur additional hardware overhead.

5.3 Graph tiling and scheduling

According to Observation 2, a real-world graph that can be very large dominates the memory accesses in GNNs and it cannot be fitted to the limited on-chip memory of EnGN. To address this issue, EnGN tiles the large graph into intervals and shards using a grid partition approach proposed in [19]. The basic idea of the grid partition is to divide all the vertices into Q disjointed intervals. Then the edges of the graph with both source and destination vertices limited to one interval can be partitioned into Q^2 disjointed shards. Each shard must be fitted to the on-chip memory of EnGN to ensure efficient computing without external memory accesses.

With the tiling, EnGN processes with the granularity of a tile. For each tile, the number of vertices remains larger than the row size of the PE array while each row of PE can only handle a single vertex at one time according to the dataflow proposed in prior section. In this case, the vertices are processed in batch and the batch size is equal to the row size of the PE array. The batch processing of a tile is described in Fig. 9. Instead of conducting *feature_extraction* and *aggregate* sequentially, we have them overlapped. Basically, *aggregate* starts when a batch of vertices complete *feature_extraction*.

TABLE 4: I/O cost.

	Read Size	Write Size
Column-oriented	$(Q^2 - Q + 1)F + QH$	QH
Row-oriented	$QF + (Q^2 - Q + 1)H$	Q^2H

Although tiling ensures EnGN to process using just the data that are accommodated in the on-chip buffers, there are still data dependency between the different tiles. The order of the tile execution essentially affects the data reuse and the amount of external memory accesses accordingly. Thereby, tile scheduling is also an important design option that needs to be intensively optimized.

The graph is split into a 2D array of tiles. The tiles in each row have the same source vertices while the tiles in the same column have the same destination vertices. Intuitively, we may schedule in either a row manner or a column manner. In the column-major order, new source vertices must be reloaded tile by tile while the destination vertices in the same interval reside in on-chip buffer until the column of tiles complete execution. In the row-major order, source vertex properties can be buffered until the whole row of tiles is processed. We also notice that there are also shared data between neighboring columns or rows and propose to schedule with an S-shape as shown in Fig. 9. For example, the bottom tile of a column shares the same source vertices with the bottom tile in the next column. Similar data sharing can be observed in row manner.

The different tile scheduling strategies mainly differ on the external memory accesses and we quantitatively analyze the I/O cost. For column-major order, each column of tiles requires to load Q tiles of source vertices and the total amount of load is Q^2. When neighboring column data reuse is considered, the amount of data to be loaded becomes $Q^2 - Q + 1$. While the destination vertices in each column can be reused, the total amount of write is Q. For row-major order, the amount of read is the same, but the amount of write is much larger, because tiles in a row generate many intermediate outputs and must be frequently swapped to external memory among different tile execution. The total amount of write is Q^2. While the dimension of the vertex property also affects the amount of I/O cost and the dimension of input vertex property and output vertex property is usually different, we further take the vertex property dimension into consideration and the I/O cost is summarized in Table 4.

Suppose that the latency of read and write external memory is equal. Comparing the overhead of the two different tile scheduling strategies, we obtain the following formulation:

$$IO_{column-major} - IO_{row-major} \approx (Q-1)(2H-F) > 0 \quad (8)$$

Based on Eq. 8, it can be concluded that the column-major order scheduling outperforms the row-major order scheduling when F is smaller than 2H. Otherwise, row-major order scheduling is preferred. While F and $2H$ are mostly determined by the GNNs and the comparison varies, we employ an adaptive scheduling to minimize the external memory accesses. The adaptive scheduling option is explicitly encoded in the instructions which are generated at compilation time based on the GNN models.

TABLE 5: System configurations and performance comparison with a state-of-the-art GCN accelerator.

	CPU-DGL/PyG	GPU-DGL/PyG	HyGCN	EnGN_22MB	EnGN
Compute Unit	3.0GHz @ 65 cores	1.25GHz @ 5120 cores	1GHz @ 32 SIMD 16 cores and 32X128 arrays	1GHz @ 128X16 arrays 32 PE units in VPU	1GHz @ 128X16 arrays 32 PE units in VPU
On-chip Memory	42.75MB	34MB	22MB+128KB	22MB+128KB	1600KB
Off-chip Memory	255.9GB/s DDR4	∼900GB/s HBM 2.0	256GB/s HBM 1.0	256GB/s HBM 2.0	256GB/s HBM 2.0
Peak Performance*(GOP/s)	-	-	8704	6144	6144
Area (mm^2)	-	-	7.8 (12nm)	31.2 (14nm)	**4.54 (14nm)**
Power (W)	150	300	6.7	10.2	**2.56**
Energy Efficiency (GOPS/W)	-	-	1299.1	602.35	**2400**
Area Efficiency (GOPS/mm^2)	-	-	1115.9	196.9	**1353.3**
GNN speedup on average	-	-	1	5.44X	**2.97X**

* Peak performance only takes into account the computing units designed for the feature extraction and aggregate stage.

TABLE 6: GNN models and datasets.

Model	Graph	#Vertices	#Edges	#Feature/ #Relation	Label
GCN	Cora (CA) [2]	2708	10556	1433	7
	PubMed (PB) [2]	19717	88651	500	3
	Nell (NE) [11]	65755	251550	5415	210
GS-Pool	CoraFull (CF) [11]	19793	126842	8710	67
	Reddit (RD) [14]	232965	114.6M	602	41
	Enwiki (EN) [9]	3.6M	276.0M	300	12
Gated-GCN	Amazon (AN) [9]	8.6M	231.6M	96	22
	Synthetic A (SA) [24]	4.19M	67.1M	100	16
	Synthetic B (SB) [24]	8.38M	134.2M	100	16
GRN	Synthetic C (SC) [24]	12.41M	205.3M	64	16
	Synthetic D (SD) [24]	16.76M	268.4M	50	16
R-GCN	AIFB (AF) [18]	8285	29043	91	4
	MUTAG (MG) [18]	23644	192098	47	2
	BGS (BG) [18]	333845	2166243	207	2
	AM (AM) [18]	1666764	13643406	267	11

Note that the strategy of DASR, graph tiling, and tile scheduling depends on both the input graph structure and the GNN model, and such processing optimization measures can be taken during the GNN model compilation stage and it influecnes the generation of EnGN instructions.

6 EVALUATION

6.1 Experimental setup

Accelerator simulator We built a cycle-accurate simulator to measure the performance of EnGN accelerator. This simulator models each module of EnGN accelerator faithfully and the timing behaviors of the modules are co-verified with the synthesized RTL design. The simulator is also integrated with Ramulator [25] that supports High Bandwidth Memory (HBM 2.0) to characterize the memory accesses to HBM 2.0 with 256GB/s bandwidth.

EnGN configuration&implementation The configuration of EnGN is depicted in Table 5. EnGN includes a 512KB multi-bank property buffer, a 512KB multi-bank weight buffer, a 256KB multi-bank edge buffer, a 256KB multi-bank result buffer, and a 64KB distributed vertex cache. We synthesized the EnGN using Design Compiler (DC) with the TSMC 14nm process technology, conducted the placing-and-routing using ICC compiler (ICC), and estimated the power consumption using PrimeTime (PT). The energy of HBM 2.0 is estimated with 3.9 pJ/bit as in [26].

Baselines We compared the performance and energy efficiency of EnGN with that of three different baseline architectures. The first two are general-purpose processors, i.e., CPU and GPU, and the third one is a state-of-the-art GCN accelerator called HyGCN. **CPU** platform is equipped with Intel Xeon(Skylake) 6151@3.0GHz processor and 696GB DRAM and **GPU** platform is equipped with NVIDIA Tesla V100 SXM2 and 32GB HBM2. To make good use of the general-purposed processors, we adopted the state-of-the-art frameworks, i.e., DGL and Pytorch geometric (PyG) to execute the GNN algorithms. The implementations are denoted as CPU-DGL, CPU-PyG, GPU-DGL, and GPU-PyG respectively. **HyGCN** that leverages 22MB eDRAM and specialized computing arrays for GNN processing achieve remarkable performance speedup over the GPU implementations. To make a fair comparison with HyGCN, we have EnGN configured with the same amount of on-chip buffer. Due to the lack of 14nm eDRAM library, we replace the eDRAM with SRAM in the experiments. More detailed configurations can be found in Table 5.

GNN models and datasets To benchmark the performance of EnGN accelerator, we implemented a set of typical GNN models on two distinct groups of datasets as shown in Table 6. The top part includes four algorithms, i.e., GCN [2], GraphSage-Pool (GS-Pool) [14], Gated-GCN [15], and GRN [27], which are mainly used for semi-supervised classification. The four algorithms perform on seven real-world graph datasets and four synthetic graph datasets. The bottom part mainly targets at knowledge graph application and R-GCN [18] is a widely adopted entity classification algorithm. The corresponding datasets are from four typical knowledge graphs. Particularly, note that the feature and label columns represent the dimension of a vertex and the number of labeled classes respectively.

6.2 Experimental results

Power&Area Table 5 shows the power and area of HyGCN, EnGN_22MB, and EnGN. As the area of eDRAM is much smaller than SRAMs, the power and area of EnGN_22MB are larger than HyGCN, but the performance speedup is more than 5X higher. Accordingly, the energy efficiency of EnGN_22MB is relatively lower in general. Nevertheless, when we compare HyGCN and EnGN, we notice that EnGN still achieves around 3X performance speedup despite the much smaller on-chip buffer. It indicates that the architecture of EnGN greatly lowers the on-chip memory requirements and power consumption. In this case, the overall energy efficiency of EnGN is 1.85X higher.

Performance We compare the performance of EnGN to that obtained from the baseline computing platforms including CPU-DGL, GPU-DGL, CPU-PyG, GPU-PyG, and HyGCN. The comparison result is shown in Fig. 10. The average performance speedup of all the models on all the datasets over CPU-DGL and CPU-PyG are 1802.9X and 5108.4X respectively as shown in the last bar of Fig. 10 (a) denoted as AVG. Also it can be observed that EnGN outperforms CPU in all cases despite the software frameworks, datasets and GNN models. We also compare EnGN with GPU using DGL and PyG respectively. However, PyG runs out of memory on larger datasets due to the lack of sufficient memory optimizations. Thus, we only compare GPU-DGL

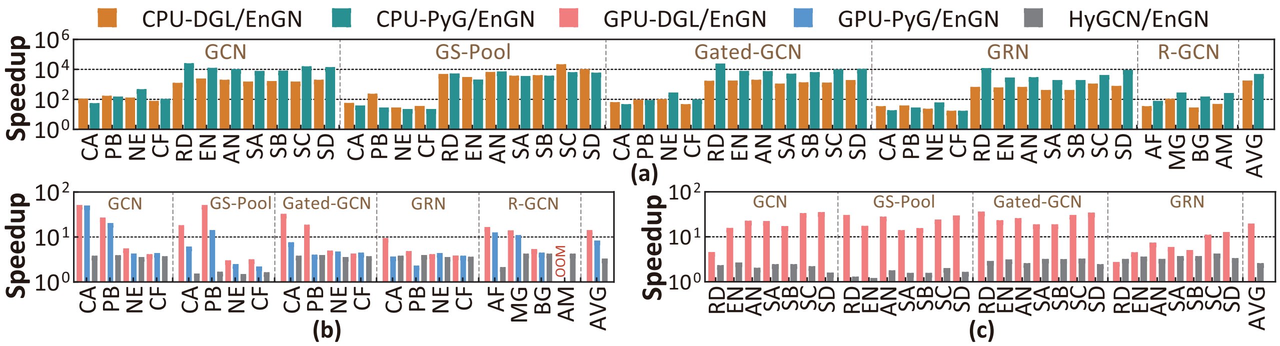

Fig. 10: Performance comparison of EnGN over CPU, GPU, and HyGCN. (a) Performance speedup of EnGN over CPU-DGL and CPU-PyG. (b) Performance speedup of EnGN over GPU-DGL, GPU-PyG, and HyGCN on small datasets. (c) Performance speedup of EnGN over GPU-DGL and HyGCN on large datasets. Since GPU-PyG runs out of memory (OOM), it is omitted.

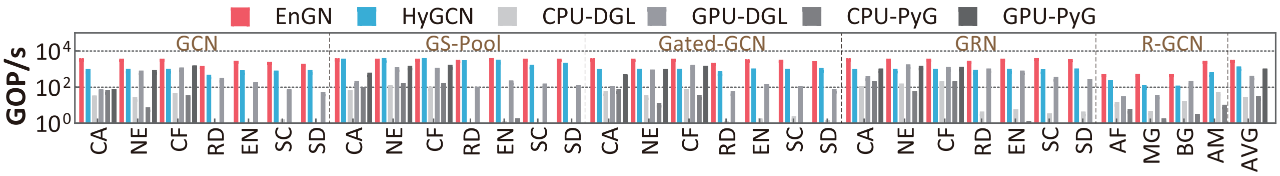

Fig. 11: Throughput of EnGN, CPU, GPU, and HyGCN. Some datasets are ignored due to literature space constraints.

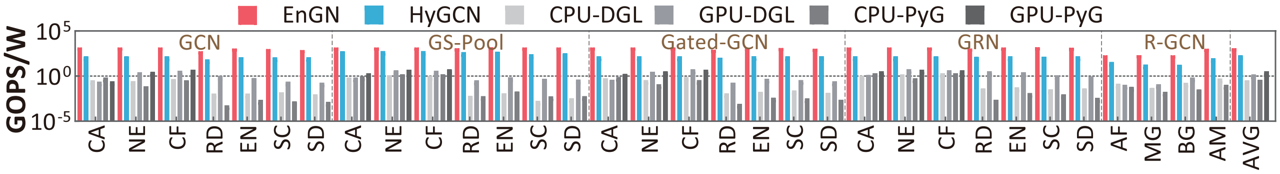

Fig. 12: Energy efficiency of EnGN, CPU, GPU, and HyGCN. Some datasets are ignored due to literature space constraints.

on large graph datasets as shown in Fig. 10 (c). On small graph datasets, we have both GPU-DGL and GPU-PyG compared and the comparison is presented in Fig. 10 (b). EnGN gains 14.41X, 8.35X, and 3.33X performance speedup over the GPU-DGL, GPU-PyG, and HyGCN respectively on the small datasets. On large datasets, EnGN achieves 19.75X and 2.61X speedup on average compared to GPU-DGL and HyGCN, respectively. In general, although GPU performs much better than CPU, EnGN still outperforms in all cases.

On top of the computing platforms, we further compare the performance speedup of EnGN on different datasets, it can be noticed that EnGN typically shows significantly higher performance speedup when the dimension of the graph feature is small. For instance, the performance speedup of GS-Pool on SD with smaller feature dimensions is around 10613.17X on CPU-DGL and 35.34X on GPU-DGL while the performance speedup of GS-Pool on CF with the larger feature dimension is less than 36.47X on CPU-DGL and less 2.22X on GPU-DGL. While EnGN with fine-grained dataflow can make good use of the computing resources, the computing efficiency does not vary much with the datasets, which will be illustrated in the following experiments. In contrast, CPUs and GPUs prefer datasets with high-dimension features that can be accessed sequentially and efficiently. Thereby, the different graph features of the datasets lead to distinct performance speedup. Meanwhile, we also find that the performance speedup of EnGN on RD with the relatively high-dimension feature is actually clearly higher than the average performance speedup. The reason for this exception is that RD has rather high average degree than the other graphs. The high-degree graph requires a large memory footprint during the aggregate stage and can no longer be fitted to the on-chip memory or cache. Thereby, the computing efficiency degrades.

Throughput Fig. 11 shows the measured throughput of EnGN, CPU, GPU, and HyGCN on the GNN benchmark in Table 6. The average throughput of EnGN is 3265.87 GOP/s, which achieves 53.15% of the peak throughput. The reason why the throughput does not reach the peak is that the execution time of the feature extraction stage is higher than that of the aggregate stage, which results in the computing units designed for the aggregate stage usually in idle status. In contrast, the measured average throughput of CPU-DGL and CPU-PyG is only 29.29 GOP/s and 31.95 GOP/s respectively, which is 111.50X and 102.21X lower. GPU with massive parallel processing units performs much better. The average throughput using GPU-DGL and GPU-PyG is 426.30 GOP/s and 1056.91 GOP/s respectively. Still, the throughput of EnGN is 7.66X and 3.09X higher. This is because GPUs are inherently optimized for compute-intensive workloads with regular execution patterns such as neural networks, but handling the aggregate stage of the EnGN processing model with irregular memory accesses suffers from low efficiency. While specialized GNN accelerators achieve much higher throughput than the general-purpose processors, architecture optimization of the accelerators especially the on-chip memory hierarchy optimizations proposed in EnGN can further improve the throughput by 2.34X over HyGCN on average. To gain insight into the computing efficiency on different GNN models and datasets, we measure the computing efficiency of the different computing architectures including EnGN, CPU, GPU, and HyGCN. As shown in Fig. 11, the computing efficiency of EnGN typically keeps steady and does not vary much with the models and datasets while CPU, GPU, and HyGCN are more sensitive and the computing efficiency

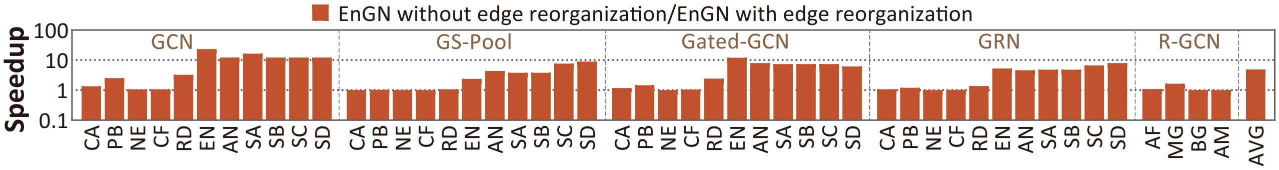

Fig. 13: Performance comparison of GNNs on EnGN with edge reorganization layout and EnGN without edge reorganization.

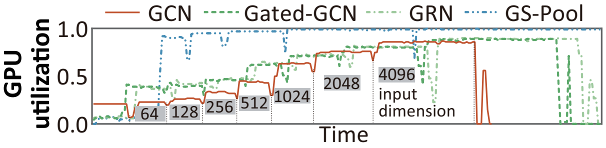

Fig. 14: GPU utilization w.r.t feature dimensions.

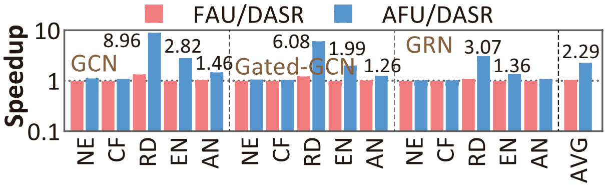

Fig. 15: Speedup of DASR over FAU and AFU.

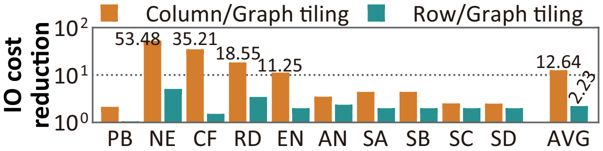

Fig. 16: I/O cost reduction.

usually fluctuates with the feature dimension of the graphs as pointed out in prior section.

Energy Efficiency To obtain the energy efficiency of the different computing architectures, we need to measure its power first. The power consumption of CPU and GPU is obtained from the power meter and NVPROF respectively. The power consumption of EnGN is estimated using PrimeTime. The power consumption of CPU, GPU, HyGCN, and EnGN is 150W, 300W, 6.7W, and 2.56W respectively. On top of the power consumption, we further calculated the energy efficiency using the total amount of operations and the execution time. The energy efficiency is shown in Fig. 12. The average energy efficiency of EnGN is 1326.35X and 1196.04X higher than CPU-DGL and CPU-PyG respectively. When compared to GPU, the energy efficiency of EnGN over GPU-DGL and GPU-PyG is 213.61X and 133.17X higher on small datasets. The speedup goes up to 529.13X for large datasets on which only DGL can be applied. Meanwhile, the energy efficiency of EnGN is 6.2X higher than HyGCN on average. The great energy efficiency speedup is mainly attributed to the much lower power consumption of the customized EnGN accelerator over the power-hungry general purposed processors and the much higher performance reported in the performance paragraph. The reasons for the higher performance and lower power consumption are already discussed, and we will not dwell on it.

6.3 EnGN optimization evaluation

Edge reorganization and RER In order to avoid the PE idling in RER, we propose to reorganize the edge list to improve the utilization of the computing array in EnGN. Fig. 13 exhibits the performance comparison of GNNs on EnGN with edge reorganization and EnGN without edge reorganization. It can be noted that the edge reorganization approach improves the performance significantly and the average performance speedup is 5.4X. Meanwhile, we find that the proposed edge reorganization approach typically works much better for large datasets. The variation of the benefits is mainly caused by the different proportions of aggregation in the total amount of GNN computing. While the aggregation in GNNs dominates the computing when the graph is large, thus the performance improvement is higher.

Sensitivity to the variation of vertex dimension The vertex property dimension varies dramatically in GNNs, so to be insensitive to the vertex property dimension variation is of vital importance to a general GNN accelerator design. In this experiment, we generated a synthetic graph with 65000 vertices, 2.5M edges, and 16 classes. Then we change the input vertex dimension from 64 to 4096 gradually to evaluate the computing efficiency variation under the different vertex property dimension setups. We compare the computing variation of EnGN and GPU-DGL. Fig. 14 depicts GPU utilization is lower than 50% when the vertex property dimension is smaller than 512. In contrast, the PE utilization of EnGN is irrelevant to the input vertex property dimension because the dataflow in EnGN decouples the input vertex property dimension and the computing array.

Dimension aware stage re-ordering As mentioned in section 5, the proposed dimension aware stage reordering technique can reduce the total computing cost. In this evaluation, we get rid of the GS-Pool model because its aggregate stage adopts the average operator which hinders the stage reordering. We compared the performance speedup of EnGN that adopts **d**imension-**a**ware **s**tage **r**e-ordering (DASR) strategy to two fixed processing strategy: (1) feature_extraction, aggregate, and update (FAU), and (2) aggregate, feature_extraction, and update (AFU). Fig. 15 illustrates that the DASR strategy can improve the performance of EnGN by 1.047x and 2.297x on averages compared to FAU and AFU, respectively. The reason for the poor performance improvement compared to FAU is the output dimensions of GNN models on most datasets are decreasing, which makes no scheduling necessary. However, in Reddit datasets, our DASR strategy can improve the performance of EnGN by 1.34x and 8.96x compared to FAU and AFU strategy. This is because the output dimensions of vertex property on the last layer are 210 (Table 6), which is higher than that of on the first layer. When the feature extraction stage performs after the aggregate stage, higher dimensions incurs massive accumulate operators in the aggregate stage. In contrast, when we perform the feature extraction stage before the aggregate stage, the dimension will be compressed to 16 and accumulates operators is only 16 for a vertex in the aggregate stage.

Graph tiling scheduling In this evaluation, we leveraged the column-major (Column) and row-major (Row) update strategy as baselines to evaluate our scheduling strategy on GCN model. Fig. 16 illustrates the total I/O

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2020.3014632, IEEE Transactions on Computers

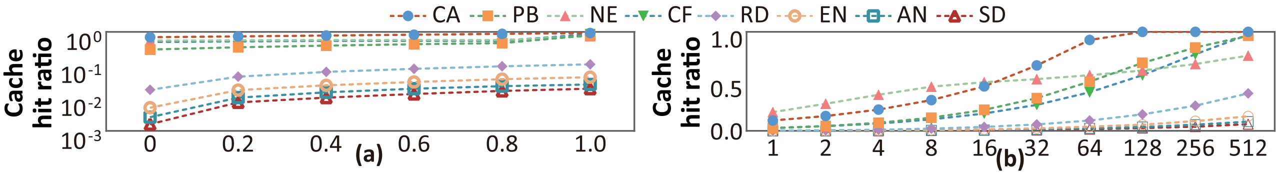

Fig. 17: Cache hit ratio over different proportions (a) and cache size (KB)(b).

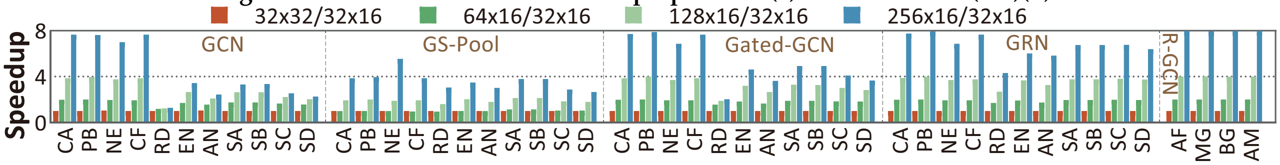

Fig. 18: Performance over number of PEs.

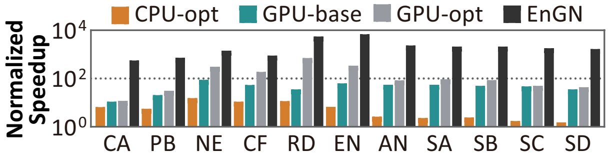

Fig. 19: Normalized speedup to CPU (DGL) without optimization (CPU-base) on GCN model.

cost reduction induced by the EnGN scheduling strategy compared to the Column and Row strategies, respectively. In PubMed and large datasets, our graph tiling scheduling strategy only reduces total I/O cost by 3.26x and 1.90x compared to the Column strategy. This is because PubMed and the large dataset only contain 3 $\sim$ 16 class labels, which is less than the output dimension of the first layer. In contrast, Nell, Cora-full, and Reddit contain 210, 67, and 41 classes respectively. Thereby, in this case, graph tiling scheduling can reduce the total memory access cost by 29.62x and 3.02x on average when compared to Column and Row, respectively. This is because the Column and Row strategy stick to the fixed policy to update the graph while our graph tiling scheduling can adjust the update dataflow from the Row to Column based on the dimension changes in GNNs. To further investigate the efficacy of DASR and graph tiling, we applied these operations to the CPU and GPU implementations using DGL framework, and compare the results of performance. The optimized CPU and GPU baselines are abbreviated as CPU-opt and GPU-opt respectively, while the original implementations on CPU and GPU are denoted as CPU-base and GPU-base accordingly. The performance speedup over the CPU-base on the selected GNN benchmark is presented in Fig. 19. It can be observed that DASR and the proposed graph tiling are beneficial to the GNN processing performance on both CPU and GPU solutions, though the speedup varies across different input graphs. Meanwhile, it confirms that EnGN outperforms all the implementations on CPU and GPU with and without such improvement.

Degree Aware Vertex Cache (DAVC) DAVC is a standard cache supporting replacement policy like LRU in general. To improve the cache hit rate, we take the vertex degree information into consideration and reserve part of the cache entries for high-degree vertices which are determined with offline static analysis and will not be replaced during the execution. To determine the proportion of the reserved cache entries, we analyze the cache hit rate under various proportion setups ranging from 0 to 1. The experiment in Fig. 17 (a) reveals that the cache hit rate increases monotonically with the proportion especially for the larger graphs. The main reason is that on-chip cache is too small relative to the large graphs and thus suffers frequent replacement when LRU policy is applied. Thereby, we have all the cache used for high-degree vertices. Meanwhile, we also analyze the influence of cache size on the cache hit rate. Similar conclusion can be drawn as shown in Fig. 17 (b). Basically, the cache hit rate for large graphs remains rather low and larger cache size is preferred. Thus, in order to reduce hardware complexity, the size of DAVC is configured to 64KB.

6.4 Scalability Analysis

Performance over number of PEs Since each row of PE array handles one vertex and each column is in charge of one dimension of output property, as the input graph and output property dimensions get larger, the system can be scaled up by adjusting the size of PE-array. We varied the size of PE-array in EnGN, where the EnGN with 32 × 16 PE-array is set as baseline. Fig. 18 show EnGN achieves good scalability on all GNN models and datasets. With the increase of the row number in PE-array, the throughput of EnGN is increasing. However, 32 × 32 array exhibits no improvement over the baseline. This is because the output property dimensions of the first layer (16) on all models are below the column number of PE array (32), which causes underutilization of PE array. Thereby, we can adjust the size of PE array according to the datasets and the complexity of GNN models to maximize the throughput of EnGN. Fig. 18 also witnessed the speedup on large datasets is lower than on small datasets. This is due to the large data has higher edge-to-vertex ratio compared to small datasets, which makes the aggregated stage new bottleneck.

7 RELATED WORK

7.1 GNNs software framework

There is a large amount of work that aims at building an efficient system for graph applications on single node-machines (CPUs) and GPUs [28]. However, these graph processing frameworks aim at traditional algorithms, and there is a lack of support for graph neural network computation. Even though TuX2 [29] aims to bridge the gap between graph and traditional machine learning algorithms, it is still unable to support the inference and training stage of emerging GNN algorithms. Thereby, NeuGraph [9] is proposed to recast the graph specific optimization as dataflow optimization based on Tensorflow. Meanwhile, [8] published a geometric learning library for deep learning on irregularly structured input data based on Pytorch. The deep graph library [7] provides a fast implementation of GNN models based on PyTorch and MxNet. NeuGraph, Pytoch geometric, and DGL are generally running on the power-hungry CPU and GPUs, which incurs energy-efficient issues and high cost. More importantly, GPUs suffer from the under-utility of stream processors during parallel GNN computation because of the impact of the irregular graph data structure, which makes energy-efficient issues more serious. Thereby, to address these issues, we build an EnGN accelerator designed for large GNNs to support energy-efficient GNN processing.

7.2 Deep learning & Graph accelerator

The resurgence of deep neural network (DNN) and its substantial progress in various applications including image, video, and speech spur the flourishing of the DNN hardware accelerator [30]. For example, Diannao [31] maps DNN onto an array of multiply-add units and employs a data tiling policy to exploiting the locality in the parameters. EIE [32] performs inference using compressed technique and accelerates the inherent modified sparse matrix-vector multiplication. However, these DNN accelerators are designed for traditional DNN such as CNN and RNN, which cannot handle GNNs because they lack the graph propagation model on the accelerator.

The wide gap between the general-purpose architectures and the unique features of graph processing promotes the rapid development of graph processing-specific accelerators based on FPGA and ASIC. For example, Graphicionado [33] and [34] presented a domain-specific accelerator for graph analytics based on a well-defined, popular vertex programming model. However, traditional graph accelerators are designed for traditional graph algorithms, it lacks the computation abstraction required by the neural network, such as tensor and activation operations. Thereby, HyGCN [21] abstracted the execution flow of GCN into aggregation and combination stage and leveraged the SIMD and systolic arrays to support neural network computation and graph propagation model simultaneously.

8 Conclusions

In this paper, we present a high-throughput and energy-efficient accelerator EnGN specialized for large graph neural network processing. In order to provide high throughput processing ability and solve the arbitrary dimension change issues in the GNN algorithms, we proposed ring-edge-reduce update dataflow and the accompanied hardware architecture of RER PE-arrays is designed to simultaneously conduct high-throughput processing in the feature-extraction, aggregate and update stages on GNNs. Meanwhile, the proposed graph tiling and scheduling technique cooperating with a well-designed three-level memory hierarchy enable EnGN to process large graphs efficiently. Experimental results show that EnGN achieves 2.97X speedup and improves energy efficiency by 6.2X on average compared to the state-of-the-art GCN accelerator HyGCN. EnGN achieves performance gains of 1802.9X and 19.75X and energy efficiency of 1326.35X and 304.43X compared to CPUs and GPUs on average, respectively.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant 61874124, Grant 61876173, Grant 61432017, Grant 61532017, Grant 61772300, and YESS hip program No.YESS2016qnrc001. Corresponding authors are Ying Wang and Huawei Li.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017.

[2] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *CoRR*, vol. abs/1609.02907, 2016.

[3] J. Chen, T. Ma, and C. Xiao, "Fastgcn: Fast learning with graph convolutional networks via importance sampling," *CoRR*, vol. abs/1801.10247, 2018.

[4] T. Hamaguchi, H. Oiwa, M. Shimbo, and Y. Matsumoto, "Knowledge transfer for out-of-knowledge-base entities: A graph neural network approach," *CoRR*, vol. abs/1706.05674, 2017.

[5] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "Aligraph: A comprehensive graph neural network platform," *CoRR*, vol. abs/1902.08730, 2019.

[6] Alibaba, "Euler: A distributed graph deep learning framework." [Online]. Available: https://github.com/alibaba/euler

[7] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *ArXiv*, vol. abs/1909.01315, 2019.

[8] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *ArXiv*, vol. abs/1903.02428, 2019.

[9] L. Ma, Z. Yang, Y. Miao, J. Xue, M. Wu, L. Zhou, and Y. Dai, "Neugraph: Parallel deep neural network computation on large graphs," in *2019 USENIX Annual Technical Conference*, July 2019.

[10] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Comput. Netw.*, vol. 33, no. 1-6, pp. 309–320, Jun. 2000.

[11] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, and M. Sun, "Graph neural networks: A review of methods and applications," *ArXiv*, vol. abs/1812.08434, 2018.

[12] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *CoRR*, vol. abs/1901.00596, 2019.

[13] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," *CoRR*, vol. abs/1704.01212, 2017.

[14] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *CoRR*, vol. abs/1706.02216, 2017.

[15] Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, "Language modeling with gated convolutional networks," *CoRR*, vol. abs/1612.08083, 2016.

[16] H. Salehinejad, J. Baarbe, S. Sankar, J. Barfett, E. Colak, and S. Valaee, "Recent advances in recurrent neural networks," *CoRR*, vol. abs/1801.01078, 2018.

[17] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," *CoRR*, vol. abs/1511.05493, 2016.

[18] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," *arXiv preprint arXiv:1703.06103*, 2017.

[19] X. Zhu, W. Han, and W. Chen, "Gridgraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 375–386.

[20] P. Faldu, J. Diamond, and B. Grot, "Poster: Domain-specialized cache management for graph analytics," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2019, pp. 473–474.

[21] 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," *ArXiv*, vol. abs/2001.02514, 2020.

[22] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and understanding gcns on gpu," *IEEE Computer Architecture Letters*, pp. 1–1, 2020.

[23] Z. Cheng and Z. Xu, "Bandwidth reduction using importance weighted pruning on ring allreduce," *CoRR*, vol. abs/1901.01544, 2019.

[24] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *SIAM International Conference on Data Mining*, 2004.

[25] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Comput. Archit. Lett.*, vol. 15, no. 1, p. 45–49, Jan. 2016.

[26] M. O'Connor, N. Chatterjee, D. Lee, J. Wilson, A. Agrawal, S. W. Keckler, and W. J. Dally, "Fine-grained dram: Energy-efficient dram for extreme bandwidth systems," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 41–54.

[27] J. You, R. Ying, X. Ren, W. L. Hamilton, and J. Leskovec, "Graphrnn: A deep generative model for graphs," *CoRR*, vol. abs/1802.08773, 2018.

[28] S. Heidari, Y. Simmhan, R. N. Calheiros, and R. Buyya, "Scalable graph processing frameworks: A taxonomy and open challenges," *ACM Comput. Surv.*, vol. 51, no. 3, Jun. 2018. [Online]. Available: https://doi.org/10.1145/3199523

[29] W. Xiao, J. Xue, Y. Miao, Z. Li, C. Chen, M. Wu, W. Li, and L. Zhou, "Tux2: Distributed graph computation for machine learning," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'17. Berkeley, CA, USA: USENIX Association, 2017, pp. 669–682.

[30] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, 2017.

[31] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 269–284.

[32] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: efficient inference engine on compressed deep neural network," *CoRR*, vol. abs/1602.01528, 2016.

[33] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A high-performance and energy-efficient accelerator for graph analytics," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-49. Piscataway, NJ, USA: IEEE Press, 2016, pp. 56:1–56:13.

[34] M. Yan, X. Hu, S. Li, A. Basak, H. Li, X. Ma, I. Akgun, Y. Feng, P. Gu, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Alleviating irregularity in graph analytics acceleration: A hardware/software co-design approach," in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: ACM, 2019, pp. 615–628.

Shengwen Liang received the B.S. degree from the HeFei University of Technology, HeFei, China, in 2016. He is currently working toward the PhD degree with the Institute of Computing Technology, Chinese Academy of Sciences, China. His current research interests include graph accelerator, near-data processing, and computer architecture.

Ying Wang (M'14) received the B.S. and M.S. degrees in Electrical Engineering from Harbin Institute of Technology, in 2007 and 2009 respectively, and the Ph.D degree of computer science from Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, in 2014. He is currently an associate professor at ICT, CAS. His research interests includes computer architecture, VLSI design, specifically memory system, on-chip interconnects, resilient and energy-efficient architecture, and machine learning accelerators.

Cheng Liu is an associate professor of Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing. He received his B.E. and M.E. degree in Microelectronic engineering from Harbin Institute of Technology in 2009 and his Ph.D. degree in computer engineering from The University of Hong Kong in 2016. His research focuses on FPGA based reconfigurable computing and domain-specific computing.

Lei He received the B.S. degree from the Wuhan University, Wuhan, China, in 2019. He is currently working toward the M.S. degree with the Institute of Computing Technology, Chinese Academy of Sciences, China. His current research interests include graph accelerator and computer architecture.

Huawei Li (SM'09) received the B.S. degree in computer science from Xiangtan University, Xiangtan, China, in 1996, and the M.S. and Ph.D. degrees from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China, in 1999 and 2001, respectively. She has been a Professor at ICT, CAS since 2008. She visited the University of California, Santa Barbara from 2009 to 2010. Her research interests include testing of VLSI/SOC circuits, design verification, design for reliability, and error tolerant computing. She has published over 120 technical papers, and holds 20 Chinese Patents in these areas. She has served as the Secretary General of the China Computer Federation Technical Committee on Fault-Tolerant Computing since 2008, and served on the technical program committees for several IEEE conferences.

Dawen Xu received the B.S. degree in computer science from Xi'dian University, Xi'an, China, in 2007, and the M.S. and Ph.D. degrees from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2009 and 2013, respectively. He is currently an associate professor in Hefei University of Technology, Hefei, China. His current research interests include heterogeneous computing, VLSI design and testing, and reliable system.

Xiaowei Li (SM'04) received his B.Eng. and M.Eng. degrees in Computer Science from Hefei University of Technology, China, in 1985 and 1988, respectively, and his Ph.D. degree in Computer Science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), in 1991. From 1991 to 2000, he was an assistant professor and an associate professor (since 1993) in the Department of Computer Science, Peking University, China. He joined the ICT, CAS as a Professor in 2000. He is now the deputy (executive) director of the State Key Lab. of Computer Architecture (ICT, CAS). He is a senior member of IEEE. Dr. Li's research interests include VLSI Testing, design verification, and dependable computing. He has co-published over 200 papers in academic journals and international conference, hold 50 patents and 45 software copyrights. Dr. Li served as Chair of CCF (China Computer Federation) Technical Committee on Fault Tolerant Computing since 2008. He served as IEEE Asian Pacific Regional TTTC (Test Technology Technical Council) Vice Chair since 2004. He served as the Steering Committee Chair of IEEE Asian Test Symposium (ATS) from 2011 to 2013.