# Cambricon-G: A Polyvalent Energy-efficient Accelerator for Dynamic Graph Neural Networks

Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, *Member, IEEE,* Qi Guo, and Yunji Chen, *Senior Member, IEEE*

***Abstract*—Graph Neural Networks (GNNs), which extend traditional neural networks for processing graph-structured data, have been widely used in many fields. The GNN computation mainly consists of the *edge processing* to generate messages by combining the edge/vertex features and the *vertex processing* to update the vertex features with aggregated messages. In addition to non-trivial vector operations in the edge processing, huge random accesses and neural network operations in the vertex processing, the graph topology of GNNs may also vary during the computation (i.e., dynamic GNNs). The above characteristics pose significant challenges on existing architectures.**

**In this paper, we propose a novel accelerator named CAMBRICON-G for efficient processing of both dynamic and static GNNs. The key of CAMBRICON-G is to abstract the irregular computation of a broad range of GNN variants to the process of regularly tiled *adjacent cuboid* (which extends the traditional adjacent matrix of graph by adding the dimension of vertex features). The intuition is that the adjacent cuboid facilitates exploitation of both data locality and parallelism by offering *multi-dimensional multi-level tiling* (including spatial and temporal tiling) opportunities. To perform the *multi-dimensional spatial tiling*, the CAMBRICON-G architecture mainly consists of the Cuboid Engine (CE) and hybrid on-chip memory. The CE has multiple Vertex Processing Units (VPUs) working in a coordinated manner to efficiently process the sparse data and dynamically update the graph topology with dedicated instructions. The hybrid on-chip memory contains the topology-aware cache and multiple scratchpad memory to reduce off-chip memory access. To perform the *multi-dimensional temporal tiling*, an easy-to-use programming model is provided to flexibly explore different tiling options for large graphs. Experimental results show that compared against Nvidia P100 GPU, the performance and energy efficiency can be improved by 7.14× and 20.18×, respectively, on various GNNs, which validates both the versatility and energy efficiency of CAMBRICON-G.**

***Index Terms*—Accelerator, Architecture, Graph Neural Networks.**

## I. INTRODUCTION

Graph Neural Networks (GNNs), which extend existing Deep Neural Networks (DNNs) for processing the *graph-structured data* that is common in various fields such as electronic commerce [1], molecular biology [2], knowledge graph [3], and social networks [4], [5], have attracted increasing attentions from both industry and academia recently [6], [7]. Technically, GNNs are a category of NN algorithms for representation learning on graphs by extracting localized features of high dimensional vertices in a graph [7]. The key operation in GNNs is *graph convolution* that extends standard 2D convolution of regular data, to non-Euclidean space convolution of irregular high dimensional vertex data. The graph convolution involves three types of data: high-dimensional *vertex* features, *edge* features, and *weight*. The computation on such data can be generally described in a *message passing* paradigm [8], which consists of the edge processing and vertex processing. In the edge processing, a *message* is generated based on the edge features and vertex features. In the vertex processing, each vertex *updates* its features based on the aggregated messages using *aggregation* operations such as sum and mean.

Though GNNs are built upon various DNN computations (e.g., inner product, and activation) and graph analytic operations (e.g., graph traversal), they significantly differ from traditional neural networks/graph analytic in three aspects. The first is that both the edge and vertex processing require a great amount of non-trivial operations (e.g., vector and NN operations). It makes state-of-the-art graph analytic accelerators (e.g., Tesseract [9], Graphicionado [10], and GraphPIM [11]), which mainly focus on scalar operations, inefficient to handle the processing of GNNs. The second is that the vertex processing requires irregular random access to high-dimensional vertex features. It makes the neural network accelerators (e.g., DianNao [12] and Eyeriss [13]), which mainly focus on regular tensor computations, inefficient to address the vertex processing of GNNs. The third is that both the graph topology and the feature dimensions may dynamically vary along with time, and it may dominate the execution of GNNs (which is completely ignored by previous work such as HyGCN [14]). For example, dynamically changing of graph occupies 66.2% and 50.5% of the total execution in GUN [15] and DiffPool [16] on Nvidia P100 GPU, respectively. All these unique features call out for dedicated architecture for efficiently processing dynamic GNNs.

To address the inefficiency of existing accelerators, in this paper, we propose a novel hardware architecture, named as CAMBRICON-G, for processing various GNNs. The key intuition is to abstract the computation of a broad range of GNN variants to the process of so-called *adjacent cuboid*, which adds the vertex feature dimension to the traditional adjacent matrix, in order to improve both data reuse and

All the authors are with State Key Laboratory (SKL) of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China (Email: {songxinkai19b, zhitian, fanzhe, zengxi, liwei, huxing, duzidong, guoqi, cyj}@ict.ac.cn, winkzy@mail.ustc.edu.cn). Xinkai Song and Zhe Fan are also with University of Chinese Academy of Sciences and Cambricon Technologies. Zhenxing Zhang is also with University of Science and Technology of China and Cambricon Technologies. Tian Zhi, Xi Zeng, Wei Li and Zidong Du are also with Cambricon Technologies. Yunji Chen is also with University of Chinese Academy of Sciences and CAS Center for Excellence in Brain Science and Intelligence Technology (CEBSIT).

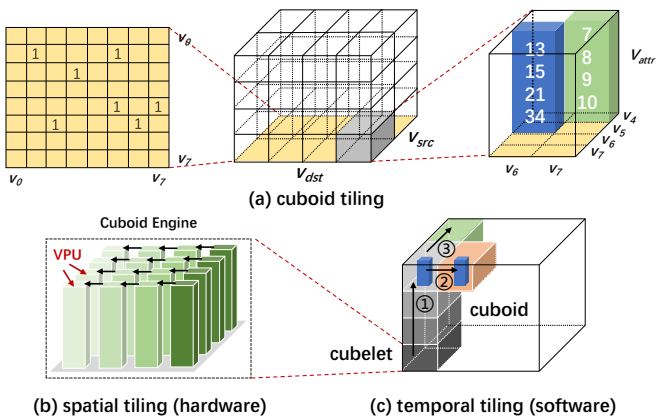


Fig. 1. Multi-dimensional multi-level tiling on the *adjacent cuboid*. (a) The adjacent cuboid consisting of adjacent matrix and vertex features can be tiled in three dimensions, i.e., $dst$, $src$, and $feat$. In the tile on the right side, two edges, i.e., $(V_4 \rightarrow V_7)$ and $(V_5 \rightarrow V_6)$, and the corresponding partial features of vertices $V_4$ and $V_5$ are presented. (b) The multi-dimensional spatial tiling is performed by the hardware cuboid engine with multiple coordinated VPUs, and the size of a cuboid to be processed is limited by the size of on-chip memory. (c) The multi-dimensional temporal tiling is performed by software programming for large graphs, and there are 3 reuse patterns to be exploited by specifying computation orders in different dimensions.

parallelism with *multi-dimensional multi-level tiling*, as shown in Figure 1. The entire cuboid can be tiled in 3 dimensions, i.e., destination vertex ($dst$), source vertex ($src$), and vertex feature ($feat$), and each partitioned tile is called *cubelet*, whose size is determined by the size of on-chip memory for efficiency reason. Based on partitioned cubelets, we propose to perform spatial and temporal tiling by hardware architecture and software programming, respectively.

To perform *multi-dimensional spatial tiling* within a cubelet for increased data reuse and parallelism, the proposed CAMBRICON-G architecture mainly consists of the Cuboid Engine (CE) and hybrid on-chip memory. The CE contains multiple Vertex Processing Units (VPUs), which are organized as a systolic array to exploit data locality and parallelism with minimized hardware costs, for processing the sparse data and update the graph topology. Note that the data that can be processed by the CE is limited by the size of on-chip memory and each VPU only processes a sliced tile of such data, as shown in Figure 1(b). Also, the CE provides dedicated instructions for different computations, i.e., cuboid-wise, plane-wise, vector-wise, and element-wise, according to the characteristics of key operations of dynamic GNNs. For example, the cuboid-wise aggregation instruction processes message aggregation of all vertices, and the vector-wise matrix multiply instruction can be used for updating the graph topology. The hybrid on-chip memory contains the hardware-managed topology-aware cache and software-managed scratchpad memory (SPM) for reducing off-chip memory accesses. Specifically, the topology-aware cache holds the input vertex features for computation, whose replacement policy and cache line size are determined by the graph topology. The SPM stores the edge features, generated vertex features (e.g., messages and partial results), and weight in separate buffers due to their distinct memory behaviors.

To perform *multi-dimensional temporal tiling* across cubelets for large GNNs, we propose a programming model to compute multiple cubelets. In Figure 1(c), after determining the tiling size of each cubelet, all cubelets can be computed

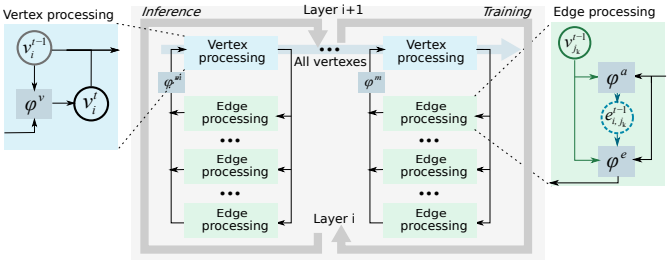


Fig. 2. The general computation paradigm of various representative GNN algorithms, which can be applied to both the forward and backward process.

by following specific orders in different dimensions, which results in different reuse patterns. More specifically, the graph topology (i.e., ①), the vertex features (i.e., ②), and partial aggregated results (i.e., ③) can be reused between different cubelets in the $feat$, $dst$, and $src$ dimension, respectively. Programmers can specify the computation orders by simply adjusting the orders of nested `for` loops with the provided programming model. For example, once it is required to reuse the features of source vertices, the cubelets in the $dst$ dimension should be traversed in the innermost loop during programming. Also, the programming model provides intrinsic to exploit hardware instructions for efficient processing.

We implement a simulator for CAMBRICON-G, and compare it against commodity processors (i.e., a 32-core two-socket Intel Xeon 6130 CPU and Nvidia P100 GPU). Compared with the multi-core CPU, CAMBRICON-G achieves 99.84× better performance. Compared with the P100 GPU, CAMBRICON-G achieves 7.14× and 20.18× better performance and energy efficiency, respectively. Experiments are conducted with 6 GNNs and 9 datasets, and the results well demonstrate the versatility and energy efficiency of the proposed architecture.

**Contributions.** To our best knowledge, this is the first hardware accelerator for efficiently processing and training dynamic GNNs. Specifically, this paper makes the following contributions:

1) *Generalized adjacent cuboid*. We abstract the irregular computation of a broad range of GNN variants to the regularly tiled adjacent cuboid. With the help of adjacent cuboid, multi-dimensional multi-level tiling in hardware and software can be performed to exploit more opportunities of data reuse.

2) *Hardware architecture*. We build the hardware architecture with the *cuboid engine* and *hybrid on-chip memory* to efficiently process sparse data and dynamically update the graph topology. An instruction set based on throughout analysis of dynamic GNN operations is also provided.

3) *Software programming*. We provide a serial programming model to flexibly specify the tiling parameters (e.g., tiling size, computation order, etc.) and implement computation logic.

## II. GRAPH NEURAL NETWORKS

In this section, we first present the general computation paradigm of GNNs. Then, we detail the processes of various representative GNNs. Finally, we investigate the source of the inefficiency of processing GNNs on GPUs and other accelerators with quantitative analysis.

### A. General paradigm of GNNs

Figure 2 presents the general computation paradigm of various GNN algorithms. Generally, a GNN consists of multiple layers to process the graph data layer by layer. Each layer can iteratively process data from two perspectives. From the vertical perspective, each vertex is processed based on aggregating all its neighboring edges with the edge processing tasks. From the horizontal perspective, all vertices are iteratively traversed with *vertex processing* tasks. Also, in a layer, both the edge processing and the vertex processing are independent from each other. Note that the above computation process applies for both forward and backward computations in inference and training, respectively. The main difference is that, for each vertex processing task, in the forward computation all the *input* edges will be processed *before* the vertex processing, while in the backward computation all *output* edges will be processed *after* the vertex processing.

In the edge processing, for vertex $i$, each input edge of vertex $i$ will compute its *message* passing through this edge, based on features of the source vertex, the destination vertex, and the edge itself[1]. Formally, the message of an edge $\vec{m}_{i,j_k}^{t-1}$ in layer $t-1$ is computed as

$$\vec{m}_{i,j_k}^{t-1} = \varphi^e(\vec{v_i}^{t-1}, \vec{v_{j_k}}^{t-1}, \vec{e}_{i,j_k}^{t-1}), \tag{1}$$

where $\vec{v_i}^{t-1}$ and $\vec{v_{j_k}}^{t-1}$ are the features of destination vertex $i$ and source vertex $j_k$, respectively, $\vec{e}_{i,j_k}^{t-1}$ is the feature of edge itself, and $\varphi^e()$ is the *message function* for the edge processing.

In the vertex processing, each vertex updates its feature based on the aggregated messages. More specifically, for the vertex $i$, all messages of its connected edges will be aggregated with the weight matrix $\omega_a$, i.e., $\vec{m}_i^{t-1} = \varphi^m(\vec{m}_{i,j_k}^{t-1}, \omega_a)$, where $\varphi^m()$ is the *aggregation function*. Formally, the feature of the vertex $i$ is computed as

$$\vec{v_i}^{t} = \varphi^v(\vec{m}_i^{t-1}, \vec{v_i}^{t-1}, \omega_u), \tag{2}$$

where $\vec{v_i}^{t-1}$ the old features in previous layer $(t-1)$, and $\omega_u$ is the weight matrix for update. The *update function* $\varphi^v$ can use different neural networks, such as MLP (multi-layer perceptron) and GRU.

### B. GNN Variants

Following the general computation paradigm as stated, we present more details of various GNN variants, including graphic convolutional network (GCN) [5], GraphSage [4], DiffPool [16], DGMG [17], EdgeConv [18], and Graph U-Nets (GUN) [15].

- **GCN.** GCN is a representative feed-forward graph neural network with *multiple graph convolutional layers*. The graph convolution can be performed with two processing tasks—*edge processing* and *vertex processing*. In edge processing, each edge computes a message passing from source vertex to destination vertex, based on the edge features, features of the source vertex and destination vertex. In *vertex processing*, each vertex updates its feature based on the messages passed from all input edges using a 1-layer MLP with a weight matrix. Note that the weight matrix $\omega_u$ is shared among all the vertices, which is similar as shared filters in a traditional convolutional layer in DNNs. In short, the graph convolution can be regarded as performing convolution over all the vertices on an irregular feature map.
- **GraphSage.** GraphSage is a representative graph neural network with *neighbor sampling mechanism*. Specifically, in the *vertex processing*, it randomly selects a certain number of input edges, instead of all input edges in GCN, to update its vertex features. The update of vertex features can use different neural network operations, e.g., fully-connected layer and max-pooling layer.
- **DiffPool.** DiffPool is a representative graph neural network with *graph pooling mechanism*, which achieves state-of-the-art performance on many graph classification tasks. DiffPool consists of several GNN layers and DiffPool layers. In GNN layers, two GCN operations are applied to obtain vertex features and vertex clustering results. In DiffPool layers, based on the vertex clustering results, a new graph is generated (only the connections are changed) and vertices will update their features correspondingly. In other words, DiffPool is a representative dynamic graph.
- **DGMG.** DGMG is a representative graph neural network algorithm to learn generative models over graphs. It contains many iterations and in each of the iteration, the algorithm dynamically adds edges or vertices.
- **EdgeConv.** EdgeConv is a representative graph neural network algorithm for point cloud data classification and segmentation. It first generates the graph topology of point cloud data using the KNN algorithm and then performs GCN on the generated graph.
- **GUN.** GUN is a representative graph neural network algorithm with an encoder-decoder model for graph data. It uses graph pooling and unpooling operations to adaptively select some nodes to form a smaller graph and restore to the original graph, respectively.

Apparently, GCN is the basis of various GNN algorithms, while other GNN algorithms can be treated as variants of GCN with alternative and the stated partial computation (e.g., GraphSage), or dynamic connections (e.g., DiffPool, DGMG, and EdgeConv).

In Table I, we also list major operations in their computation orders of each representative GNN algorithm. Major operations in GNNs are vector-based operations, which can be classified into 3 categories based on the operating objects: *edge*, *vertex*, and *graph*. For example, in GCN, first, each edge generates the message by multiplying the source vertex feature (vector) with the edge feature (scalar), thus the edge with VMS operation; then each destination vertex aggregates all the messages (vectors) on input edges by summation, thus the vertex with VADD; finally, each output destination vertex updates its feature through a 1-layer MLP, thus vertex with VMM. Other GNNs are presented in the same principle. Particularly, dynamic GNNs such as DiffPool, DGMG, and EdgeConv require new operations to process the partial/entire

[1]Here we use the forward computation for simplicity since both the forward and backward computation have the same computation paradigm. In the backward computation, for vertex $i$, each output edge of vertex $i$ will compute and pass the gradients to its destination.

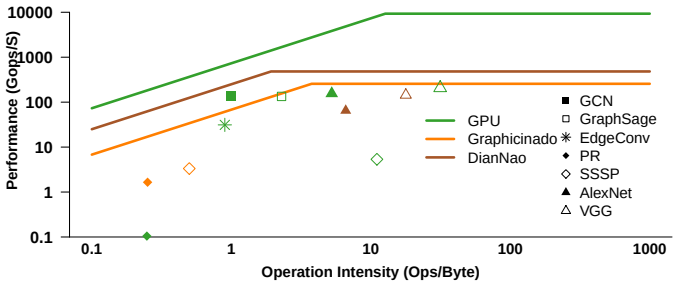


Fig. 3. Roofline model for GPU [19], Graphicinado [10], and DianNao [20].

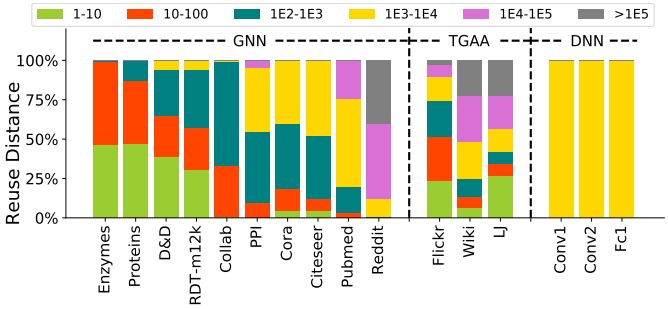


Fig. 4. The reuse distance distribution of vertex access for different datasets in the edge processing, where the reuse distances are divided into 6 categories such as `1--10` and `10--100`. We can see that the diversity of reuse distance distribution lies between that of TGAAs and DNNs.

graph network. These new operations for dynamic GNNs can be classified into two categories: used to generate new graph topology entirely (e.g., KNN, VSORT, VMM), used to update the partial graph topology by adding/removing edges/vertices (e.g., VCAT, VSPT).

### C. Quantitative Analysis

To illustrate the bottleneck of different platforms when processing GNNs as well as traditional graph analytic algorithms (TGAAs) and DNNs, we adopt the roofline model [21] to present the results, see Figure 3. A modern commercial general processing device, Nvidia P100 GPU [19], is included to perform three representative GNNs, i.e., GCN, GraphSage, and EdgeConv. As a comparison, a state-of-the-art TGAA accelerator, Graphicionado [10], and a well-known DNN accelerator, DianNao [20], are modeled to perform TGAAs, e.g., *PageRank* (PR) and *Single Source Shortest-Path* (SSSP), and deep neural networks (DNNs), e.g., AlexNet and VGG16, respectively. It can be observed that on the GPU the efficiency of processing GNNs is quite low compared to that of DNNs. For GraphSage, GPUs only achieves 1.44% computation efficiency with 2.29 operation intensity, while the numbers on VGG are 2.22% and 31.43. The achieved computation efficiency and operation intensity of EdgeConv are even worse, i.e., 0.34% and 0.89.

The inefficiency of GPU is largely from the irregular data access in GNNs. Figure 4 presents the reuse distance distribution of vertex access for different GNN datasets in edge processing, where we compare the GNNs against some TGAAs and typical layers in DNNs. The results demonstrate that the divergence of reuse distance distribution lies between that of TGAAs and DNNs. More specifically, the variance of reuse distance of vertex access is 83479, while that of TGAAs and DNNs are 218428 and 0, respectively. Therefore, as a proof, GPU attains higher efficiency on GNNs than TGAAs but lower efficiency than DNNs.

Such irregularity brings poor data locality, thus increasing the difficulty for data reuse. Therefore, GNNs may require specialized architecture (e.g., dedicated memory hierarchy) to exploit the reuse distance for improving the efficiency of irregular vertex access. Otherwise, it would lead to a low on-chip data reuse ratio. For example, running the *aggregation* operation in GraphSage on Reddit, GPU transfers 264.47 GB data (vertices and edges) between its GDDR and L2 cache in total, 278.68× larger than the size of the dataset.

The large size of graphs and dynamic GNNs further exacerbate the challenge of data reuse. The former is from both the large amount of vertices/edges in real-life scenarios (e.g., 232,965 vertices and 114,615,892 edges in *Reddit*, a representative social network graph) and the high-dimensional vertex feature (e.g., 602 float numbers in each vertex feature in *Reddit*). In our practice, the 16 GB GDDR P100 GPU is unable to run GraphSage with LJ dataset due to the out-of-memory error. The latter poses new challenges on the architecture design to handle the complicated control logics of dynamic topology update, which currently requires costly invocation of the host CPU (including both the CPU-GPU interaction and the CPU execution). We evaluate the performance overhead of topology update on GUN and experimental results show that the execution of CPU even takes 96.6% of the execution of topology update, i.e., 63.9% of the execution of the entire algorithm.

[2]VMS: vector multiplying scalar; VADD: vector addition; VMM: vector multiplying matrix; VELW: vector element-wise operation; VCAT: vector concat; VSPT: vector split; MMM: matrix multiplying matrix; VSORT: vector sort; KNN: K-nearest Neighbor.

TABLE I
MAJOR OPERATIONS[2] WITH COMPUTING ORDER (❶-❺).

| GNN | Edge | Vertex | Graph |
|---|---|---|---|
| GCN | ❶VMS | ❷VADD; ❸VMM (MLP) | - |
| GraphSage | ❷VMS | ❶VMM (MLP); ❸VELW; ❹VMM (MLP) | - |
| DiffPool | ❶VMS | ❷VADD; ❸VMM (MLP) | ❹MMM |
| DGMG | ❶VMS | ❷VADD; ❸VMM (MLP) | ❹VCAT/VSPT |
| EdgeConv | ❸VMS | ❹VADD; ❺VMM (MLP) | ❶KNN; ❷VSORT |
| GUN | ❸VMS | ❹VADD; ❺VMM (MLP) | ❶VMM; ❷VSORT |

## III. Adjacent Cuboid

As stated, it is challenging to improve data reuse on existing architecture. We first reveal that data locality can be exploited with naive tiling on traditional GPU, and then elaborate that the adjacent cuboid is a systematic approach for better exploiting data locality with improved reuse.

### A. Data locality

We investigate various data locality that might be exploited for improved efficiency. Specifically, we consider exploiting the vertex locality, edge locality, and message locality on a commodity GPU. Figure 5 reports the performance and data access (i.e., off-chip memory access) results of different tiling strategies, when using GCN with three datasets (i.e., FL, PK, and RDT) as driving examples. We simply tile the graph topology (including the source vertex and destination vertex) and vertex features by equally dividing into 1, 4, and 16 segments, and the tiling is conducted either separately or jointly, which results in 26 different tiling solutions. Generally,

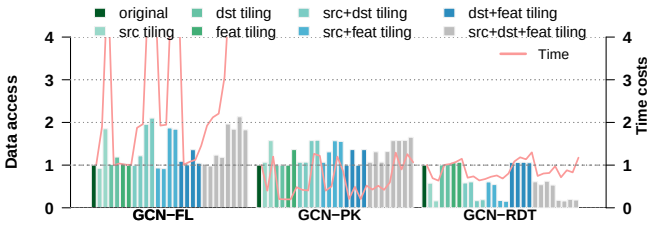


Fig. 5. Exploiting data locality with tiling on different kinds of data of GNNs.

the *vertex locality* can be exploited when tiling the destination vertex, where the source vertices are shared among different destination vertices; the *edge locality* can be exploited when tiling the vertex features, where the edge weights are shared among the same destination vertices; the *message locality* can be exploited when tiling the source vertex, where thusly produced messages are shared among intermediate results.

*Observation 1:* Data locality could lead to performance improvement, reduced data access, or both. As shown in Figure 5, GCN with *RDT* can always benefit from tiling the source vertices (w/ and wo/ tiling other data) with a maximum performance improvement of 1.57× and a maximum off-chip data traffic reduction of 84.7%. GCN with *PK* can achieve performance improvement in most of the tiling strategies but with almost none reduction in off-chip data access.

*Observation 2:* Simple tiling does not always lead to performance benefits. Figure 5 shows that GCN with *FL* cannot achieve performance improvement and data access reduction in most of the tiling strategies. Particularly, many tiling strategies could even lead to significant performance slowdown, e.g., 6.05×. The major reason is that with our simple and naive tiling strategies, the costs of GPU kernel launch, intermediate results load/store could not be surpassed by the exploited data locality. Therefore, GPU may need fine-grain exploration of optimal tiling strategies.

### B. Improving data reuse

As there exist different data locality opportunities during GNN processing, a systematic and general approach might be able to maximally exploit the locality for improved data reuse. We observe that in addition to tiling the traditional graph topology (e.g., adjacent matrix), tiling the vertex features can also improve data reuse. Thus, it is expected to further improve the data reuse via exploring the co-space of multiple tiling options. Following this intuition, we propose to use adjacent cuboid to abstract the computation of a broad range of GNNs. In the adjacent cuboid, it is easy to decompose the cuboid into small tiles in different dimensions for exploring different reuse opportunities. Figure 6 shows how to reuse the graph topology, vertex features, and partially aggregated results across tiles by decomposing the adjacent cuboid in 3 dimensions, i.e., vertex feature ($feat$), destination vertex ($dst$), and source feature ($src$), respectively. Also, topology changes such as adding/removing edges in the dynamic graphs can benefit from tiling the adjacent cuboid to reduce unnecessary data movements.

**Reuse graph topology.** Figure 6(b) shows that the adjacent cuboid is partitioned in the $feat$ dimension. If these two partitioned tiles are computed continuously, it only needs to access the graph topology once as the topology can be reused between different tiles in the $feat$ dimension.

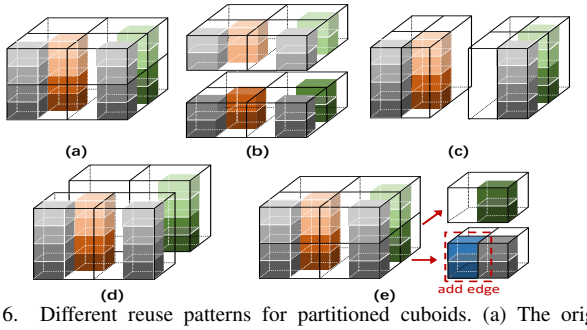


Fig. 6. Different reuse patterns for partitioned cuboids. (a) The original adjacent cuboid. (b) The adjacent cuboid partitioned in the $feat$ dimension, where the graph topology can be reused. (c) The adjacent cuboid partitioned in the $dst$ dimension, where the vertex features can be reused. (d) The adjacent cuboid partitioned in the $src$ dimension, where the partially aggregated results can be reused. (e) Adding a new edge into the cuboid, and the topology of most tiles remain due to tiled storage of sparse graphs.

**Reuse vertex features.** Figure 6(c) shows that the adjacent cuboid is partitioned in the $dst$ dimension. If these two partitioned tiles are computed continuously, the vertex feature for the same source vertex can be reused between different tiles in the $dst$ dimension.

**Reuse partial aggregation.** Figure 6(d) shows that the adjacent cuboid is partitioned in the $src$ dimension. When aggregating all messages for a vertex, the partially aggregated results of partitioned tiles can be reused for further accumulation.

**Reuse tiled representation.** In the tiled adjacent cuboid, the graph topology is also stored in a tiled format (i.e., the topology data within a tile is stored continuously). Thus, when adding/removing edges in dynamic graphs, it is only necessary to conduct data movement in associated tiles, while the topology of other tiles remains due to the sparsity of graphs, as shown in Figure 6(e).

Note that in the above examples, all decomposed tiles can be processed in parallel with abundant computation units. Once the size of the adjacent cuboid (i.e., the graph size) exceeds the processing ability of the hardware, such a large graph should be further tiled temporally, which is addressed by software programming for guaranteeing flexibility. In short, such *multi-dimensional spatial and temporal tiling* offered by the adjacent cuboid can be exploited by hardware architecture and software programming, respectively, so as to balance the efficiency and flexibility.

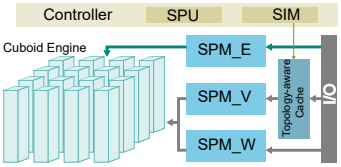


Fig. 7. Architecture overview of CAMBRICON-G.

## IV. HARDWARE ARCHITECTURE

To fully utilize the data locality and computing parallelism with the adjacent cuboid, we propose CAMBRICON-G, a polyvalent energy-efficient accelerator for dynamic GNNs. Figure 7 shows the overview of CAMBRICON-G, which consists of three major design components: the cuboid engine (CE), the hybrid memory hierarchy, and the controller. The *CE* is the most important processing component for processing GNNs, and it consists of multiple vertex processing units (VPUs) which are organized as a 2D-mesh. The *hybrid memory hierarchy* contains both the software-controlled scratchpad

memory for regular data access and the hardware-managed cache for irregular data access. The hardware-managed cache is designed to handle the reuse of source vertices in cuboid-wise operations, and it is topology-aware to improve the data locality of vertex access. The *controller* is designed to coordinate all the modules in CAMBRICON-G to efficiently process GNNs as well as manage the spatial tiling and the sparsity in GNNs.

### A. Instruction Set Architecture

Before detailing the architecture of CAMBRICON-G, we first elaborate on the proposed instruction set architecture (ISA) to well support the vector-major operations for edge, vertex, and entire graph in GNNs. The ISA contains four-level instructions, including element-wise, vector-wise, plane-wise, and cuboid-wise, see Table II. The cuboid-wise instructions are specially designed to perform common operations on the entire cuboid, e.g., *message aggregation*. The plane-wise instructions are designed to perform operations on a plane of the cuboid, e.g., *update* with matrix-vector multiply, and other 2D matrix related operations. The vector-wise and element-wise instructions are designed to perform operations on a vector and an element, respectively, in the cuboid. Specially, for efficiently supporting dynamic GNNs, CAMBRICON-G contains 3 vector instructions, including vector sorting (VSORT), vector split (VSPT), and vector concat (VCAT). The VSPT and VCAT instructions are used for adding/removing an edge/vertex into/from the GNNs. The VSORT instruction is used for efficiently sorting a vector in GNNs such as KNN in EdgeConv.

### B. Cuboid Engine

In CAMBRICON-G, we construct the core computation component, i.e., the Cuboid Engine (CE), in a 3D form to match the 3D cuboid tiling of GNNs as mentioned in Section III. In Figure 8, we show the detailed architecture of the proposed CE. The CE consists of multiple Vertex Processing Units (VPUs) (Figure 8 (a)), which are organized as a 2D array. Each VPU contains several scalar operators followed by a reduction operator (Figure 8 (b)), which is able to gather the results of scalar operators for further processing (e.g., summation, comparison, nonlinear functions). Thus, the CE is a 3D cuboid with $N_R \times N_C \times N_{SOP}$ scalar operators, where $N_R$ is the number of VPUs in a row, $N_C$ is the number of VPUs in a column, and the $N_{SOP}$ is the number of scalar operators in a VPU. The VPU is designed to perform vector operations in GNNs, including inner production (IP), vector element-wise operations (e.g., addition, subtraction, multiplication, division), vector comparison (e.g., greater, less, equal), etc.

TABLE II
EXAMPLE INSTRUCTIONS IN CAMBRICON-G'S ISA.

| Type | Instructions | Operand |
|---|---|---|
| Cuboid-wise | cuboid aggregation (CAG), cuboid aggregation-update fusion (CAUF) | A cubelet |
| Plane-wise | matrix multiplying matrix/vector/scalar (MMM, MMV, MMS), matrix adding/subtracting matrix (MAM, MSM) | A plane (2D matrix) |
| Vector-vise | vector element-wise operations (VADD, VSUB, VMV, VDV), inner production (IP), vector sort (VSORT), vector concat (VCAT), vector split (VSPT) | A 1D vector |
| Element-wise | scalar arithmetic | Two scalars |

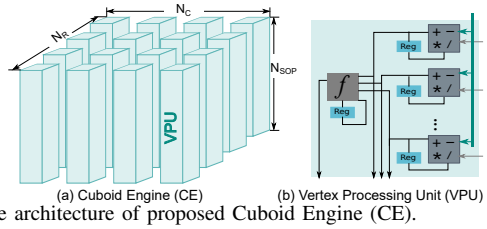


Fig. 8. The architecture of proposed Cuboid Engine (CE).

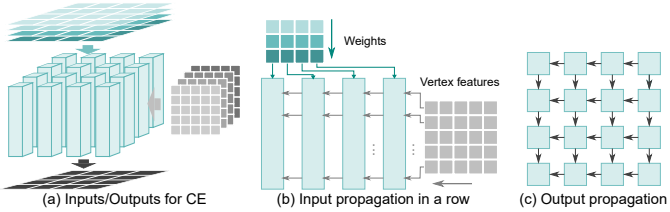


Fig. 9. Data propagation in the CE.

**Data propagation.** Figure 9(a) shows the data propagation mechanism among VPUs adopted in the CE. For each operation, the CE receives two 2D matrices as inputs from the on-chip buffer: one feeding from the top direction and the other feeding from the right direction, and the CE generates the output at the bottom of the VPUs. This mechanism is achieved with the input propagation and output propagation mechanism, as shown in Figure 9(b) and Figure 9(c), respectively. Regarding the input propagation, each VPU in the CE receives one scalar input (e.g., edge weight) from the top direction but the other vector input from its right neighboring VPU (or the on-chip buffer). Regarding the output propagation, each VPU in the CE is able to send its output to its left or bottom VPU. Thus, the reduction operations could be directly performed with output propagation among VPUs, rather than ineffectively accessing the on-chip buffer for synchronization of VPUs. We adopt such a design rather than a naive design that connects all the VPUs to the on-chip buffer, which suffers from heavy wire congestion. For example, for a $16 \times 16 \times 16$ CE with 32-bit floating-point units, the naive solution would require $16 \times 16 \times 16 \times 32 \times 2$ =262144-bit inputs (i.e., $N_C \times N_R \times N_{SOP} \times data\_width \times 2$) from on-chip buffer. With a 45 nm TSMC technology which contain 4 metal layers for routing, wires in such a CE would take most chip area (98.18%). On the contrary, in the proposed data propagation mechanism, it only requires $(16 \times 16 + 16 \times 16) \times 32$ =16384-bit inputs (i.e., $(N_R \times N_{SOP} + N_R \times N_C) \times data\_width$). Apparently, the wire congestion between the CE and the on-chip buffer is drastically reduced.

**Data reuse.** Massive data reuse can be achieved by the stated data propagation mechanism. Particularly, source vertices are shared among destination vertices when generating *messages* (Section II-A Formula 1) for *aggregation* in a cubelet, and weight matrix is shared among all the vertices when *updating* the vertices features (Section II-A Formula 2).

The former *message aggregation* can be implemented by the CAG instruction. Each VPU would work on one destination vertex, where independent weights are sent to VPUs and the features of source vertices are passed from the rightmost VPU to the leftmost VPU gradually, i.e., *input propagation*. Each VPU receives, checks, and processes the source vertex that has edge connections with its destination vertex. In this way,

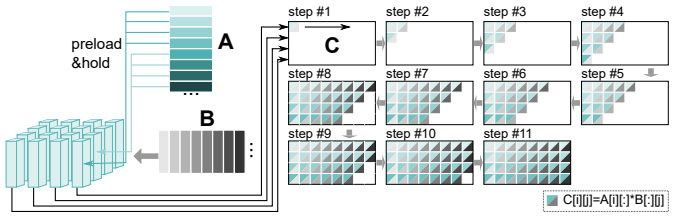


Fig. 10. The MMM computation example in the CE ($\mathbf{C} = \mathbf{A} \times \mathbf{B}$, each VPU is processing one row in $\mathbf{C}$ where it holds a row vector in $\mathbf{A}$ with column vectors from B passing through.).

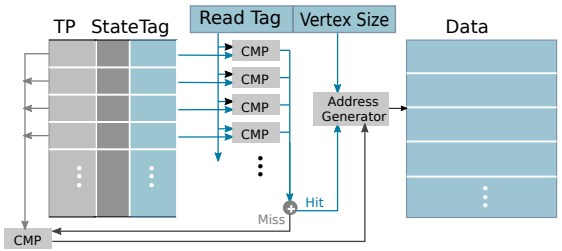


Fig. 11. Topology-aware cache.

the input data is reused in the CE by passing the input data across VPUs in the same row.

The latter *vertex update* can be achieved by the MMM instruction for performing *update* with the MLP (Multi-layer Perceptron) or GRU (Gated Recurrent Unit) operation. Figure 10 shows how the CE performs the MMM instruction. Each VPU in a row keeps one row vector of the input matrix $\mathbf{A}$ for performing inner production with the column vectors of the input matrix $\mathbf{B}$, which is propagated from the rightmost VPU to the leftmost VPU. Therefore, each VPU processes one row of the output matrix $\mathbf{C}$ with one element at each time step. Similarly, other rows of VPUs will keep other row vectors of the input matrix $\mathbf{A}$ to compute others rows of the output matrix $\mathbf{C}$, while the input matrix $\mathbf{B}$ is shared among all the VPUs.

### C. Hybrid memory hierarchy

To leverage the shared data among cubelets, we architect a software-hardware hybrid memory hierarchy, which contains both the software-controlled scratchpad memory (SPM) for regular data accesses and the hardware-managed topology-aware cache for irregular data accesses.

*1) Scratchpad memory:* There are three separate SPMs, i.e., the SPM_E, the SPM_V and the SPM_W, for holding the edge features, (intermediate or output) vertex features, and weights, respectively. The reason to organize the SPM with 3 separate buffers rather than a unified one is that the reuse characteristics of these 3 types of data are quite different. In detail, the edge features can be shared among neighboring tiles (i.e., VPUs in spatial tiling or cubelets in temporal tiling) in the $feat$ dimension. The vertex features can be shared among neighboring tiles in the $src$ dimension. The weights can be shared among neighboring tiles in the $dst$ dimension.

*2) Topology-aware cache:* In addition to the data stored in the SPM (i.e., edge features, intermediate/output vertex features, and weights), the original input vertex features are accessed via the proposed topology-aware cache, which is specially designed for improving the reuse ratio of randomly accessed vertices between tiles. The reason to employ the hardware-managed cache rather than a software-controlled SPM is two-fold. First, as the memory accesses of vertices are quite random, when using the SPM, it typically requires dedicated preprocessing algorithms (e.g., clustering vertices based on their linked edges) for the graph data to significantly improve the reuse ratio, which no doubt results in considerable programming and computation overheads. Second, for dynamically changing graphs where some of the memory accesses can only be determined at runtime, it is tedious and error-prone to manage the SPM during programming. The topology-aware cache is implemented by adding an additional topology flag field in the cache architecture and deploying the topology-aware replacement policy to exploit the data locality in accessing vertex features.

**Basic cache architecture.** The basic architecture of the topology-aware cache is shown in Figure 11. It is a simplified example (N-way 1-set-associate) of the topology-aware cache with the following parts: a data field (i.e., Data) for buffering the vertex features, a status field (i.e., State) field for labeling the status of cache lines, a tag (i.e., Tag) field for recording the vertex ID, and a topology flag (i.e., TP) field for recording the topology-related metrics. The topology-aware replacement policy uses the TP field to determine the replacement priority. The cache line size is configurable to efficiently support GNNs, as the lengths of the vertex feature could vary among GNNs and even different layers in a GNN. We use a dedicated register to record the parameter of cache line size.

**Topology metrics.** There are two types of GNN topology metrics, i.e., vertex reuse distance (VRD) and degree, that can be applied in the TP field for achieving different replacement policies. The former metric refers to the number of visited vertices between the two consecutive visits to the same source vertex. The corresponding vertex reuse distance can be generated during compilation. With the VRD metric in the TP field, the topology-aware replacement policy replaces the vertex that has the largest VRD for better data reuse. The latter refers to the degree value (in or out degree) of vertices, and the vertex with the smallest degree will be replaced, as it has the least possibility to be revisited than other vertices in the cache.

We compared the data access efficiency of *topology-aware replacement policy* using the above two different metrics. Specifically, we implement a customized 16-way set-associative cache with the capacity of 10 MB and adopt the off-chip memory access volume to indicate the memory access efficiency. According to the evaluation results in three commonly-used input graphs (*Livejournal, Youtube*, and *Reddit*), the *VRD-aware* cache and *degree-aware* cache exhibit $1.275\times$ and $1.390\times$, respectively, more off-chip memory traffic than the optimal case where data only need to be loaded once. Hence, we adopt the VRD as the topology metrics when establishing the topology-aware cache architecture.

### D. Controller

The controller in CAMBRICON-G is designed to manage all the components for efficiently processing instructions. In addition to the common functionalities (e.g., instruction fetching and decoding), the controller contains two extra functional modules: the Scalar Processing Unit (SPU) for scalar compu-

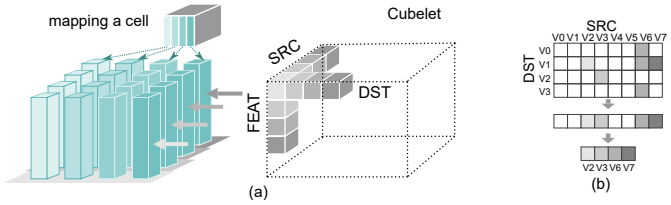


Fig. 12. (a) Hardware scheduling in processing a cubelet with partition in 3 dimensions, i.e., $src$, $dst$, and $feat$. (b) Processing the sparse source vertices.

tations and the Sparsity Indexing Module (SIM) for leveraging the sparsity in GNNs. The SPU directly processes the element-wise instructions, while the other three types of instructions (i.e., vector-wise, plane-wise, and cuboid-wise) are processed by the VPUs. Nevertheless, for efficiently processing cuboid-wise instructions such as CAG, the controller employs a dedicated hardware scheduling mechanism to improve the data reuse and the SIM to improve the VPU utilization for processing sparse data.

**Hardware scheduling.** The controller supports different scheduling strategies by partitioning the cubelet in $src$, $dst$, and $feat$ dimensions, see Figure 12 (a). Precisely, the controller divides the cubelet into small cells, where each cell contains $N_C$ vertices and $N_{SOP}$ feature value. Each cell is directly processed by a VPU row in the CE and each VPU is working on a destination vertex. In the $src$ dimension, the cubelet is partitioned equally to $N_R$ parts, which avoids the overlap of source vertices among VPU rows for maximizing the throughput between the on-chip buffer and the CE. Therefore, the $dst$ partition leads to the reuse of source vertices among VPU rows, the $feat$ partition the edge weights reuse among VPU rows, and the $src$ partition the intermediates results reuse in the VPU columns (i.e., output propagation).

**SIM for sparsity.** To well leverage the sparsity in source vertices for high efficiency, the controller deploys a dedicated hardware module, i.e., Sparsity Indexing Module (SIM). In detail, during the decoding stage of a CAG instruction, the SIM first generates indexes for the only necessary source vertices required by each VPU row (Figure 12 (b)). Those indexes are sent to the VRD-aware cache to check whether the corresponding source vertices requests are *hit* or *miss*. The missed requests will be processed by the VRD-aware cache for fetching data from the memory. Note that since we use the CSR compressed format for graphs in GNNs, the filtering process is achieved by simply combining the *column indexes*.

## V. Programming Model

Figure 13 shows an example of programming DiffPool on CAMBRICON-G, where the key is to partition the execution of entire cuboid into cubelets by specifying the *tiling size*, *computation order* and *tiled storage* of graph topology. The *tiling size* is specified with variables in 3 dimensions, e.g., `dstDim` and `dstSize` in the $dst$ dimension. In this example, the entire cuboid is partitioned into $4 \times 2 \times 4$ cubelets, and each cubelet can be identified with a tuple of $(feat, src, dst)$, e.g., the cubelets on the left bottom and the right top can be identified as $(0, 0, 0)$ and $(3, 1, 3)$, respectively. The *computation order* can be specified with nested `for` loops from the software perspective. In this example, the computation order is specified as $(dst \rightarrow feat \rightarrow src)$ by traversing the cubelet

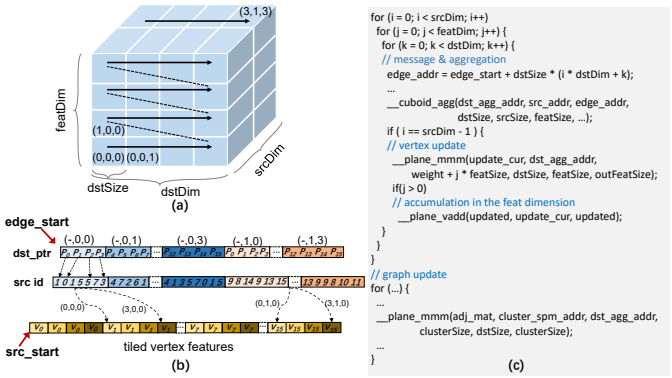


Fig. 13. Programming example of DiffPool. (a) Temporally tiling the cuboid into $4 \times 2 \times 4$ cubelets in all 3 dimensions, and the computation order is $dst \rightarrow feat \rightarrow src$. (b) The graph topology of each cubelet is reorganized in the tiled CSR format. (c) The main code for processing one layer of DiffPool.

computation along with the $dst$ dimension in the inner-most loop. The intuition of *tiled storage* is to organize the data required by cubelet processing in a compact form so that they can be held in the on-chip memory. In this example, the graph topology (e.g., which is stored in the CSR format[2]) is reorganized in a tiled manner during compilation. Therefore, at runtime, the cubelet data can be held in the on-chip memory.

There are two nested `for` loops as the core implementation of DiffPool. The first nested loop processes the traditional GCN operations and the second updates the graph topology. In the first nested loop, the starting address of all processed edges of each cubelet (i.e., `edge_addr`) is first obtained by using the tiling parameters (i.e., `dstSize` and `dstDim`), which will be used as the inputs of the cuboid-wise instruction, i.e., `__cuboid_agg` (i.e., the intrinsic wrapper of the CAG instruction), for obtaining the aggregated results of messages. Then, for the last cubelet in the $src$ dimension with the id of `(srcDim - 1)`, the neural network operation as `__plane_mmv` (i.e., the intrinsic wrapper of the MMV instruction) should be enforced to update features of all vertices. Also, for the cubelet with the id in the $feat$ dimension larger than 0, the partial results of updated vertex features should be accumulated. In the second nested loop, in addition to the similar process of GCN operations, matrix multiply (i.e., `__plane_mmm`) should be performed to generate a new graph topology.

**Address space.** Since CAMBRICON-G has both hardware-managed cache and software-managed SPM, and the address space should be specified for each variable. There are two different types of address space, i.e., DRAM (with cache) and SPM. The variables are by default stored in DRAM, which will be accessed through the cache. The data stored in the SPM should be specified by using type qualifier, e.g., `__spmw__` for weight data. During computation, only the input vertex features are stored in DRAM, and other data including the graph topology, aggregated messages, weights, and updated vertex features should be stored in different types of SPM. To support dynamic memory management of DRAM, we implemented memory management functions including memory allocation (`__malloc`) and memory free (`__free`).

[2]Other typical sparse formats of graph topology such as COO (Coordinate) can be addressed in a similar way.

Also, data transferring between DRAM and SPM is supported by using `__memcpy` with specified parameters as `DRAM2SPM`.

**Tiling parameters.** The tiling parameters include the tiling size and computation order, which are crucial to the performance and energy efficiency. The tiling size is intuitively determined by the size of the on-chip SPM. Once the size of the used SPM exceeds the total capacity, the compilation will terminate with a reported error message. The computation order is also very critical, and there are 6 possible computation orders (e.g., $dst \rightarrow src \rightarrow feat$ and $src \rightarrow feat \rightarrow dst$). Different computation orders imply different reuse patterns. For example, given the computation order as $dst \rightarrow src \rightarrow feat$, the graph topology can be reused for different cubelets in the $feat$ dimension.

**Intrinsic.** The intrinsic functions are wrappers of native hardware instructions as stated in Section IV-A. There are four types of intrinsic functions, i.e., cuboid-wise (e.g., `__cuboid_agg`), plane-wide (e.g., `__plane_mmv`), vector-wise, and element-wise, which can be directly compiled to related hardware instructions. Other computational (e.g., `+`, `-`, and `*`) and control statements (e.g., `if`) are compiled to scalar instructions.

## VI. METHODOLOGY

### A. Simulators and benchmarks

**Simulators.** We use both hardware layout and simulator to evaluate the performance and cost of the CAMBRICON-G. Due to the extreme long hardware emulation time, we carefully build an in-house cycle-accurate simulator in C++ to simulate the exact behavior of CAMBRICON-G. For the hardware characteristics, we implement the CAMBRICON-G with Verilog for synthesizing, placing, and routing using TSMC 45 nm technology to evaluate timing, area, and power costs. We use the Ramulator [22] to estimate the latency of main memory, for the benefit that it can reflect the effect of the crucial random accesses to the main memory. We integrate the Ramulator into our simulator to simulate the behaviors of an HBM bank with 128GB/s memory bandwidth by sending the memory traces directly to Ramulator.

**Benchmarks.** The benchmarks used for evaluation are listed in Table III. We train six GNN algorithms, two static GNNs (GCN and GraphSage) and four dynamic GNNs (DiffPool, DGMG, EdgeConv, and GUN). For each GNN, various realistic datasets are considered for evaluation.

TABLE III
BENCHMARKS.

| GNN | dataset | # vertex | # feature | # edges |
|---|---|---|---|---|
| GCN, GraphSage | youtube (YT) | 1,134,890 | 256 | 5,975,248 |
| | flickr (FL) | 105,938 | 512 | 4,633,896 |
| | pokec (PK) | 1,632,803 | 128 | 30,622,564 |
| | reddit (RDT) | 232,965 | 602 | 114,615,892 |
| DiffPool | D&D (DD) | 334,925 | 89 | 1,686,092 |
| EdgeConv | MN40 | 1024 | 3 | 20480 |
| DGMG | Cycle (CY) | 10∼20 | 16 | - |
| GUN | Colors (CL) | 10∼20 | 16 | - |
| | enzymes (EN) | 19,580 | 18 | 74,564 |

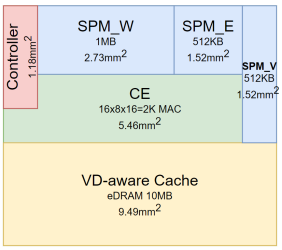


Fig. 14. Floor Plan of CAMBRICON-G.

### B. Baselines

We use CPU and GPU as our baselines for comparison.

**CPU.** We select a server processor as our CPU baseline, i.e., Intel Xeon 6130, which has 32 cores, 512GB memory, and 171GB/s bandwidth. We implemented all these benchmarks with the DGLv0.4 [8] and compiled with SIMD support (`--native` option).

**GPU.** We use the modern Nvidia P100 GPU card as our GPU baseline, which has a 9.3 TeraOpes/sec peak performance (single precision), 21.5 MB on-chip memory, with a main memory bandwidth of 732GB/s. We use the official NVprof tool [23] to measure the performance and power. We enable the persistent mode to avoid startup cost. The benchmarks are also implemented in DGLv0.4 and compiled onto the GPU. Especially, for a fair comparison, we measure the start time after the data are copied to its HBM2 memory (ignore the long CUDA malloc time) and the end time after the final results are written back to HBM2.

## VII. EXPERIMENTAL RESULTS

In this section, we first present the hardware characteristics of CAMBRICON-G, and then report the performance and energy by comparing against the baselines.

### A. Hardware characteristics

In this paper, we implement CAMBRICON-G with $N_C \times N_R = 16 \times 8 = 128$ VPUs ($N_{SOP} = 16$), thus 2048 scalar operators in total. CAMBRICON-G has 12.08 MB on-chip buffer in total, i.e., 10 MB for the topology-aware cache, 1 MB for the SPM_W, 512 KB for the SPM_E, and 512 KB for the SPM_V. In Table IV, we present the layout results of CAMBRICON-G. Figure 14 is the floor plan of our hardware. It achieves a peak performance of 4 Top/s and works at 1 GHz, with a total area of 21.87 $mm^2$ and a power consumption of 3.62 W. Comparing against the GPU baseline, CAMBRICON-G has only 44.09% peak performance, 17.49% on-chip main memory bandwidth, 66.7% on-chip buffer, but achieving 4.35× better performance and 37.25× more energy savings, averagely.

TABLE IV
HARDWARE CHARACTERISTICS OF CAMBRICON-G.

| | Area($mm^2$) | % | Power(mW) | % |
|---|---|---|---|---|
| Total | 21.87 | 100.00% | 3628.98 | 100.00% |
| CE | 5.46 | 24.97 | 2408.50 | 66.37 |
| Controller | 1.18 | 5.40 | 68.73 | 1.90 |
| SPM_V | 1.52 | 6.94 | 155.43 | 4.28 |
| SPM_E | 1.52 | 6.94 | 155.43 | 4.28 |
| SPM_W | 2.73 | 12.47 | 208.83 | 5.75 |
| VD-aware Cache (edram) | 9.49 | 43.36 | 632.56 | 17.43 |

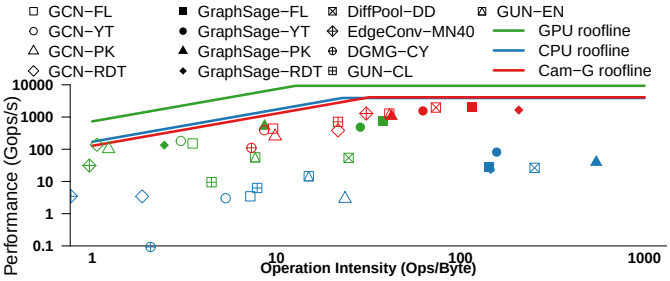


Fig. 15. Roofline model of CPU/GPU baselines and CAMBRICON-G.

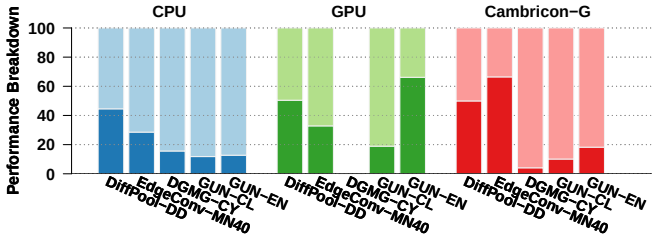


Fig. 16. Performance breakdown (dark colors: dynamic topology change).

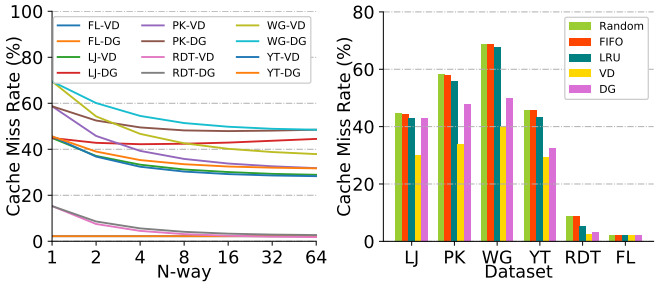


Fig. 17. *Left:* The miss rate when varying the $N$ in N-way set-associate topology-aware Cache (VRD: vertex-distance aware cache; DG: degree-aware cache). *Right:* The miss rate when using different replacement policies.

## B. Performance

To illustrate the efficiency and bottleneck, we use the roofline model [21] to show the results of CAMBRICON-G when comparing against CPU and GPU baseline. As shown in Figure 15, we label all the benchmarks [3] on the three platforms to the same roofline figure, where each benchmark uses the same point character but different colors for different platforms, i.e., green, blue, and red for GPU, CPU, and CAMBRICON-G. On average, CAMBRICON-G achieves 99.84× and 7.14× better performance, 69.5% and 14.9% lower traffic, when comparing to CPU and GPU, respectively.

Benefit from the VRD-cache which improves the data reuse ratio of irregular vertices accesses, the CAMBRICON-G attained higher operation intensity (larger in x-axis direction in Figure 15), 28% on average, which is 4.98× higher than the GPU baseline (4.68%). For example, running aggregation operation in GraphSage with *Reddit*, GPU transfers 264.47 GB between its memory and on-chip buffer; such number on CAMBRICON-G is reduced to 7 GB, a 37.78× reduction to GPU. Also, CAMBRICON-G attained higher hardware utilization (higher in y-axis direction), 17.9% on average, which is 14.18× larger than GPU baseline (1.26%).

To well demonstrate the efficiency of CAMBRICON-G on dynamic GNNs, we breakdown the performance into static graph processing part (i.e., *feature update*) and dynamic graph processing part (i.e., *dynamic topology change*), as shown in Figure 16. It can be observed that on the dynamic GNNs that require generating new topology , i.e., DiffPool and EdgeConv, the dynamic part takes 49.25% of the total execution time on CAMBRICON-G, comparing to 35.68% on CPU and 40.71% on GPU. It shows that the static part in GNNs is well accelerated on CAMBRICON-G, achieving more speedup than the dynamic part. On the dynamic GNNs that require update partial topology, i.e., DGMG and GUN, the dynamic part takes 9.11% of the total execution time on CAMBRICON-G, comparing to 13.29% and 35.42% on CPU and GPU, respectively. It shows that the complex, heavy interacted dynamic parts are efficiently processed on CAMBRICON-G.

*1) Cache specifications.:* We also investigate different organizations of the topology-aware cache in CAMBRICON-G, including direct-map and N-way set-association. Figure 17 (*Left*) shows the miss rates for six different GNN graphs, with varying the number of ways $N$ in our in the N-way set-associate topology-aware cache. Note when $N = 1$, the cache changes into a direct-map cache; when N goes as large as the number of the entries, the cache changes into a fully associate cache. Since the fully associate cache is only practical to a smaller number of entries, we stop our exploration at $N = 64$. The two types of topology-aware cache behaviors similar but VRD-aware cache achieves a better hit rate. Obviously, the miss rates reach the plateau when $N$ is large than 16 in the current implementation with 10 MB cache size, and thus we select N as 16 in our topology-aware cache in current CAMBRICON-G implementation.

Figure 17 (*Right*) compares the miss rates for the same GNN graphs with different replacement policies, including random, first-in-first-out (FIFO), least-recently-used (LRU), and our *topology-aware replacement policy*. It can be observed that when using a 10 MB cache, the *topology-aware replacement policy* on VRD-aware cache achieves much better hit rates, i.e., 1.32×, 1.32×, 1.28, and 1.12× better than random, FIFO, LRU, and DG, respectively.

## C. Energy consumption

In Figure 18, we report the energy results when comparing the CAMBRICON-G against the GPU baseline. The case of DGMG-CY running on GPU is missing because it is not supported by DGL due to the complex control flow of dynamic graph changing in DGMG. We observed that the CAMBRICON-G could achieve 20.18× energy savings. Averagely, CAMBRICON-G achieves 9.98× energy savings on the eight static GNNs, while 82.41× on the four dynamic GNNs. The CAMBRICON-G achieves the highest benefit on GUN with dataset CL, i.e., 205.0× energy savings. CAMBRICON-G achieves the lowest benefit on GraphSage with *flickr* dataset with only 4.30× energy savings. The reason for such low benefit is because of the high ratio of VMM operation (MLP) in computation of GraphSage (see the computing order of GraphSage (with maxpool) in Table I), 44.93%, 4.99× higher than that in GCN (9.00%).

[3]DGL does not provide a GPU version of DGMG.

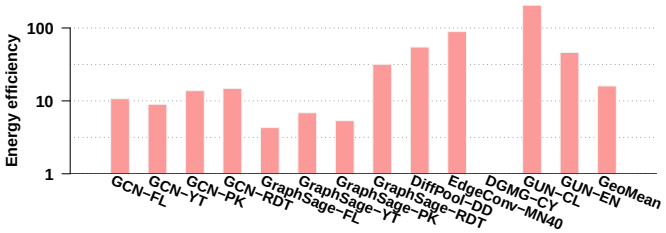


Fig. 18. Energy savings over GPU.

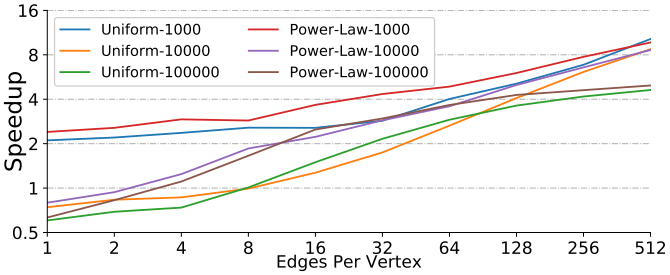


Fig. 19. Exploration of CAMBRICON-G speedups over GPU.

### *D. Topology exploration*

We also explore the efficiency scope of CAMBRICON-G. Figure 19 reports the speedup of CAMBRICON-G over the GPU baseline of GCN on several generated graphs. Those generated graphs have 1000, 10000, and 100000 256-length vertices and the average vertex degree vary from 1 to 512: thus the sparsity vary from 48.8%∼99.9%, 94.88%∼99.99%, and 99.488%∼99.999% on the 1000-vertex, 10000-vertex, and 100000-vertex graphs, respectively. In order to mimic the graph data in real-life [24], we generate the graphs with their degree following power-law distribution, besides uniform distribution. It can be observed that CAMBRICON-G always performs better on graphs in power-law distribution than in uniform distribution. The main reason is that the VRD-aware cache can better model the distribution feature by leveraging the vertices reuse distances. Additionally, when numbers of average vertex degree are less than 3 for 10000-vertex graph and 100000-vertex graph in power-law distribution, 8 for 10000-vertex graph and 100000-vertex graph in uniform distribution, CAMBRICON-G is even slower than GPU. For such sparse graphs (sparsity $> 99.92\%$ for uniform distribution or sparsity $> 99.97\%$ for power-law distribution), the possibility of data reuse among destination vertices is quite small. Therefore, the memory system begins to dominate the performance, where the 21.5 MB on-chip buffer in GPU beats the 12.8 MB in CAMBRICON-G and the 732GB/s bandwidth of GPU beats the 128GB/s of CAMBRICON-G. Except for enlarging the on-chip buffer and the DRAM bandwidth, a possible approach to improve the performance of CAMBRICON-G in these extremely sparse cases is to enhance the off-chip bandwidth utilization. For instance, CAMBRICON-G can prefetch and preprocess a batch of edges and then reorder the vertices to be accessed to take full advantage of pipeline parallelism of DRAM banks by avoiding workloads unbalance among them. And CAMBRICON-G can also reorder the accesses to give priorities to those in opened DRAM rows. As real-life graph datasets such as measured in this paper have sparsities less than $99.95\%$ with the power-law distribution of vertex degrees, CAMBRICON-G is able to attain speedup on all the graph datasets. Such results well demonstrate the efficiency of CAMBRICON-G.

## VIII. DISCUSSION

Following the message-passing paradigm, GNNs have certain relations with traditional graph analytic algorithms and neural networks. Therefore, it is valuable but also interesting to see if the straightforward idea of extending those accelerators with their missing part could work. Here we only focus on static GNNs as dynamic GNNs require non-trivial redesign.

**TGAA-based accelerator.** The TGAA-based acceleratoris extended from a state-of-the-art TGAA accelerator, Graphicionado [10], with a NN functional unit (i.e., the NPU). While this accelerator features an application-specific pipeline, we implement the *apply* phase (used for performing vertex computation in Graphicionado) with an NPU, for the sake of efficiently *vertex processing*. However, it still fails to efficiently reuse data and computation, especially the high-dimensional vertices. When processing *Reddit* with GCN, the TGAA based accelerator (64 MB on-chip SRAM) has to read the vertex features roughly $\sim 10\times$ more times ($\sim$6 GB off-chip traffic), leading to a low data reuse ratio and high bandwidth requirement. Additionally, due to the computation imbalance between NN operations and traditional scalar operations (e.g., pointer chasing) in graph analytics, the execution pipeline is frequently stalled by NN operations, which causes 23.26x performance degradation compared to the GPU baseline.

**DNN-based accelerator.** The DNN based accelerator is extended from a well-known DNN accelerator, DianNao [12], with a large cache. The cache is commonly employed as an effective solution for irregular random memory access. Therefore, to guarantee a high hit rate, the naive accelerator is equipped with a large cache, i.e., 64 MB as used in the TGAA based accelerator. However, this accelerator still suffers from huge off-chip traffic and low data reuse ratio. For the same *Reddit* with GCN, this accelerator need to transfer 193 GB data, $340\times$ more than the original size of *Reddit*, leading to an $8.33\times$ worse performance than the GPU baseline.

Overall, these two naive solutions are still far away from well addressing the challenges for efficiently processing GNNs. The main reason is that the data are still not well reused (the hit rates are only 30% on the DNN-based designs, respectively).

## IX. RELATED WORK

We review the related work on traditional graph analytic algorithm (TGAA) accelerators, DNN accelerators, and GCN accelerators.

**TGAA accelerators.** In addition to several software optimization techniques, GAA accelerators have been proposed to improve the performance and energy efficiency from a hardware perspective [25]–[27]. Most of them tried to leverage near data processing techniques, including near memory processing [9]–[11], [28], [29] and near storage processing [30], [31], to alleviate the most-critical memory-bounded problem of traditional GAAs. Moreover, as traditional GAAs typically

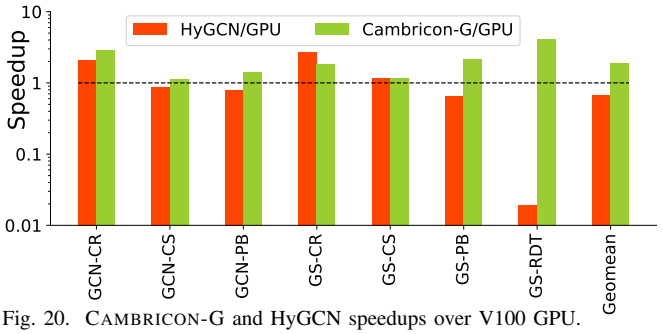


Fig. 20. CAMBRICON-G and HyGCN speedups over V100 GPU.

have lightweight computations (e.g., pointer chasing and index computation), these accelerators typically adopt general-purpose multi-core architecture for computation, e.g., 16 out-of-order cores [11], [25], dual-core ARM processor [30], or even 512 in-order cores [9]. Thus, they cannot efficiently handle relatively heavy computations such as neural network operations in each vertex of GNNs. Compared against these GAA accelerators, the proposed CAMBRICON-G architecture employs both specialized computation units (i.e., the cuboid engine) and memory hierarchy (i.e., hybrid on-chip memory) to alleviate the memory-bound and computation-bound problem of GNNs.

**DNN accelerators.** With the popularity of deep neural networks, both academia and industry have proposed many dedicated accelerators implemented with FPGAs or ASICs to accelerate the inference/training of the memory-intensive and computation-intensive DNNs instead of using CPUs/GPUs [32]–[53]. Many of these accelerators achieve high efficiency with better on-chip data reuse and special computation flows such as vector/matrix-based architecture including DianNao Family [54], and systolic-like architectures, including Eyeriss [13], ShiDianNao [55], and Google TPU [56]. Later accelerators tried to leverage shorter bit-width data through bit-serial based computation [57]–[60] and sparse data through dedicated hardware modules [61]–[65]. However, although the sparsity introduces irregularity in computation, these DNN accelerators still perform computations with regular memory access, e.g, unidirectional order with contiguous or discontiguous access. As the graphs processed by GNNs are high-dimensional data with irregular memory access, i.e., random access without a specific order, such DNN accelerators cannot be directly applied to GNNs for high efficiency.

**GCN accelerators.** In addition to TGAA and DNN accelerators, GCN accelerators also emerged recently. HyGCN [14] is proposed for accelerating GCN with two processing engines for the aggregation and update phase, respectively. However, HyGCN targets at accelerating the inference of the basic GNN algorithm (i.e., GCN) without paying special attention to accelerating topology update. Moreover, due to the lack of dedicated ISA and system-level programming model, HyGCN is much less flexible and versatile compared to CAMBRICON-G.

TABLE V
BENCHMARKS OF HYGCN.

| GNN | dataset | # vertex | # feature | # edges | storage |
|---|---|---|---|---|---|
| GCN, GS | CiteSeer (CS) | 3,327 | 3,703 | 9,104 | 47MB |
| | Cora (CR) | 2,708 | 1,433 | 10,556 | 15MB |
| | Pubmed (PB) | 19,717 | 500 | 88,648 | 38MB |
| | Reddit (RDT) | 232,965 | 602 | 114,615,892 | 972MB |

Figure 20 shows the comparison of CAMBRICON-G and HyGCN speedups over V100 GPU. For a fair comparison, we use the same benchmarks as HyGCN and use the numbers (speedups over V100 GPU) reported in the original paper. Results show that CAMBRICON-G achieves 1.89× speedup over V100 GPU while HyGCN achieves 0.659× speedup over V100 GPU, averagely. On small graphs such as Cora, Citeseer, and Pubmed, CAMBRICON-G achieves similar performance with HyGCN. On the largest graph Reddit (with 232,965 vertices and 114,615,892 edges), CAMBRICON-G achieves much better performance than HyGCN due to that CAMBRICON-G takes full advantage of data reuse and greatly reduces the off-chip memory accesses through the cuboid-level abstraction of GNNs and the topology-aware cache.

## X. CONCLUSION

In this paper, we propose CAMBRICON-G, the first polyvalent accelerator for efficiently processing both dynamic and static graph neural networks. The CAMBRICON-G architecture consists of the cuboid engine and hybrid on-chip memory to process the so-called adjacent cuboid to maximally exploit the data locality and parallelism. An instruction set architecture is proposed to conduct both traditional GNN operations and dynamic change of graph topology. Moreover, an easy-to-use programming model is proposed to further explore different tiling options for large graphs. Experimental results demonstrate that compared against Nvidia P100 GPU, CAMBRICON-G improves the performance and energy efficiency by 7.14× and 20.18×, respectively, on 6 GNN algorithms with 9 datasets.

## ACKNOWLEDGMENT

This work is partially supported by the National Key Research and Development Program of China (under Grant 2017YFA0700900, 2017YFA0700902, 2017YFA0700901), the NSF of China (under Grants 61925208, 61732007, 61732002, 61702478, 61906179, 62002338, 61702459, U19B2019, U20A20227), Beijing Natural Science Foundation (JQ18013), Key Research Projects in Frontier Science of Chinese Academy of Sciences (QYZDB-SSW-JSC001), Strategic Priority Research Program of Chinese Academy of Science (XDB32050200, XDC05010300, XDC08040102), Beijing Academy of Artificial Intelligence (BAAI) and Beijing Nova Program of Science and Technology (Z191100001119093), Guangdong Science and Technology Program (2019B090909005), Youth Innovation Promotion Association CAS and Xplore Prize.

## 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 *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983, ACM, 2018.

[2] A. Fout, J. Byrd, B. Shariat, and A. Ben-Hur, "Protein interface prediction using graph convolutional networks," in *Advances in Neural Information Processing Systems*, pp. 6530–6539, 2017.

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

[4] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[6] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, and C. Li, "Graph Neural Networks : A Review of Methods and Applications," pp. 1–22.

[7] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, S. Member, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," vol. XX, no. Xx, pp. 1–22, 2019.

[8] y. Wang, Minjie and Yu, Lingfan and Zheng, Da and Gan, Quan and Gai, Yu and Ye, Zihao and Li, Mufei and Zhou, Jinjing and Huang, Qi and Ma, Chao and Huang, Ziyue and Guo, Qipeng and Zhang, Hao and Lin, Haibin and Zhao, Junbo and Li, Jinyang and Smola, Alexander J and Zhang, Zheng, journal=ICLR Workshop on Representation Learning on Graphs and Manifolds, "Deep graph library: Towards efficient and scalable deep learning on graphs,"

[9] J. Ahn, "A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture - ISCA '15*, pp. 105–117, 2015.

[10] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi, "Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[11] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks," *Proceedings - International Symposium on High-Performance Computer Architecture*, pp. 457–468, 2017.

[12] T. Chen, Z. Du, N. Sun, J. Wang, and C. Wu, "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 (ASPLOS'14)*, pp. 269–283, 2014.

[13] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.

[14] 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," *CoRR*, vol. abs/2001.02514, 2020.

[15] H. Gao and S. Ji, "Graph U-nets," in *Proceedings of The 36th International Conference on Machine Learning*, 2019.

[16] Z. Ying, J. You, C. Morris, X. Ren, W. Hamilton, and J. Leskovec, "Hierarchical graph representation learning with differentiable pooling," in *Advances in Neural Information Processing Systems*, pp. 4800–4810, 2018.

[17] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia, "Learning deep generative models of graphs," *arXiv preprint arXiv:1803.03324*, 2018.

[18] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *ACM Transactions on Graphics (TOG)*, vol. 38, no. 5, pp. 1–12, 2019.

[19] NVIDIA Corporation, "NVIDIA Tesla P100," 2017. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf.

[20] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam, "DianNao family: Energy-efficient hardware accelerators for machine learning," *Commun. ACM*, vol. 59, pp. 105–112, Oct. 2016.

[21] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[22] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," *IEEE Computer architecture letters*, vol. 15, no. 1, pp. 45–49, 2015.

[23] NVIDIA Corporation, "CUDA toolkit documentation: Profiler user's guide." https://docs.nvidia.com/cuda/profiler-users-guide/index.html.

[24] N. Eikmeier and D. F. Gleich, "Revisiting power-law distributions in spectra of real world networks," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 817–826, 2017.

[25] A. Mukkara, N. Beckmann, M. Abeydeera, and D. Sanchez, "Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–14, 2018.

[26] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 166–177, 2016.

[27] X. Ma, D. Zhang, and D. Chiou, "Fpga-accelerated transactional execution of graph workloads," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 227–236, ACM, 2017.

[28] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, "GraphR : Accelerating Graph Processing Using ReRAM," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[29] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "GraphP : Reducing Communication for PIM-based Graph Processing with Efficient Data Partition," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 544–557, 2018.

[30] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "GraphSSD : Graph Semantics Aware SSD," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 116–128, 2019.

[31] M. Zhang, Y. Wu, Y. Zhuo, X. Qian, C. Huan, and K. Chen, "Wonderland : A Novel Abstraction-Based Out-Of-Core Graph Processing System," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*, pp. 14–17, 2018.

[32] S. Venkataramani, A. Ranjan, S. Banerjee, D. Das, S. Avancha, A. Jagannathan, A. Durg, D. Nagaraj, B. Kaul, P. Dubey, and A. Raghunathan, "SCALEDEEP : A Scalable Compute Architecture for Learning and Evaluating Deep Networks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 2017.

[33] M. Imani, S. Gupta, Y. Kim, and T. Rosing, "FloatPIM : In-Memory Acceleration of Deep Neural Network Training with High Precision," in *Proceedings of the 46th International Symposium on Computer Architecture*, 2019.

[34] H. Jang, J. Kim, J.-e. Jo, J. Lee, and J. Kim, "MnnFast : A Fast and Scalable System Architecture for Memory-Augmented Neural Networks," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 250–263, 2019.

[35] J. Liu, H. Zhao, A. Ogleari, D. Li, and J. Zhao, "Processing-in-Memory for Energy-efficient Neural Network Training: A Heterogeneous Approach," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, no. Section II, 2018.

[36] M. Mahmoud, K. Siu, and A. Moshovos, "Diffy: a Dejavu-Free Differential Deep Neural Network Accelerator," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[37] C. Eckert, X. Wang, J. Wang, A. Subramaniyan, R. Iyer, D. Sylvester, D. Blaauw, and R. Das, "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[38] E. Park, D. Kim, and S. Yoo, "Energy-efficient Neural Network Accelerator Based on Outlier-aware Low-precision Computation," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[39] V. Akhlaghi, A. Yazdanbakhsh, K. Samadi, R. K. Gupta, and H. Esmaeilzadeh, "SnaPEA : Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[40] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, and S. W. Keckler, "vDNN: Virtualizing Deep Neural Networks for Memory-Efficient Neural Network Design," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[41] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko, "Gist : Efficient Data Encoding for Deep Neural Network Training," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pp. 776–789, 2018.

[42] M. Gao and M. Horowitz, "TETRIS : Scalable and Efficient Neural Network Acceleration with 3D Memory," in *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, pp. 751–764, 2017.

[43] Y. Li, J. Park, M. Alian, Y. Yuan, Z. Qu, P. Pan, R. Wang, A. G. Schwing, H. Esmaeilzadeh, and N. S. Kim, "A Network-Centric Hardware / Algorithm Co-Design to Accelerate Distributed Training of Deep Neural Networks," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, no. 2, 2018.

[44] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators," in

*2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 267–278, 2016.

[45] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 14–26, 2016.

[46] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. W. Fletcher, "UCNN: Exploiting Computational Reuse in Deep Neural Networks via Weight Repetition," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[47] Z. Li, C. Ding, S. Wang, W. Wen, Y. Zhuo, C. Liu, Q. Qiu, W. Xu, X. Lin, X. Qian, and Y. Wang, "E-RNN: Design Optimization for Efficient Recurrent Neural Networks in FPGAs," *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 69–80, 2018.

[48] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das, "Bit Prudent In-Cache Acceleration of Deep Convolutional Neural Networks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 81–93, IEEE, 2019.

[49] A. Samajdar, P. Mannan, K. Garg, and T. Krishna, "GeneSys: Enabling Continuous Learning through Neural Network Evolution in Hardware," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[50] K. Hegde, R. Agrawal, Y. Yao, and C. W. Fletcher, "Morph: Flexible Acceleration for 3D CNN-based Video Understanding," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, 2018.

[51] A. Yazdanbakhsh, K. Samadi, N. S. Kim, and H. Esmaeilzadeh, "GANAX : A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.

[52] G. Gobieski, N. Beckmann, and B. Lucia, "Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[53] A. Segura, J.-m. Arnau, and A. González, "SCU : A GPU Stream Compaction Unit for Graph Processing," in *Proceedings of the 46th International Symposium on Computer Architecture*, pp. 423–434, 2019.

[54] Y. Chen, T. Chen, X. Zhiwei, and O. Temam, "DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning," *Communications of the ACM*, vol. 57, no. 5, p. 109, 2014.

[55] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pp. 92–104, 2015.

[56] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. Mackean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17)*, pp. 1–17, 2017.

[57] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-Serial Deep Neural Network Computing," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, vol. 6056, pp. 1–1, 2016.

[58] J. Albericio, P. Judd, A. D. Lascorz, S. Sharify, and A. Moshovos, "Bit-Pragmatic Deep Neural Network Computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–16, 2017.

[59] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, J. K. Kim, V. Chandra, and H. Esmaeilzadeh, "Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Networks," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2017.

[60] J. Albericio, P. Judd, A. D. Lascorz, S. Sharify, and A. Moshovos, "Bit-Pragmatic Deep Neural Network Computing," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 1–16, 2017.

[61] 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," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, vol. 16, pp. 243–254, 2016.

[62] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "SCNN : An Accelerator for Compressed-sparse Convolutional Neural Networks," in *The 44th International Symposium on Computer Architecture (ISCA)*, 2017.

[63] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos, "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–13, 2016.

[64] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "SparTen: A sparse tensor accelerator for convolutional neural networks," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 151–165, 2019.

[65] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 359–371, 2019.

**Xinkai Song** received the bachelor's degree in engineering from the Department of Automation, Tsinghua University, Beijing, China, in 2016. Currently he is working toward the PhD degree in the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, and the University of Chinese Academy of Science, Beijing, China, under the guidance of prof. Yunji Chen.

**Tian Zhi** received the bachelor's degree in engineering from the Department of Biomedical Engineering & Instrument Science, Zhejiang University, Hangzhou, China, in 2009, and the PhD degree from Institute of Electronics, Chinese Academy of Sciences, Beijing, China, in 2014, with the guidance from prof. Haigang Yang. She is currently an associate professor at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

**Zhe Fan** received the bachelor's degree in engineering from the Department of Computer Science, Huazhong University of Science and Technology, Wuhan, China, in 2017. Currently he is working toward the PhD degree in the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, and the University of Chinese Academy of Science, Beijing, China.

**Zhenxing Zhang** received the bachelor's degree in mathematics from University of Science and Technology of China, Hefei, China, in 2017. Currently he is working toward the PhD degree at the University of Science and Technology of China, Hefei, China, under the guidance of prof. Xuehai Zhou. He is also a visiting student with the Intelligent Processor Research Center, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, under the guidance of prof. Qi Guo. His current research interests include computer architecture and parallel computing.

**Xi Zeng** Zengxi received the bachelors degree in computer science and technology from Central South University, Changsha, China, in 2015. and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2020, with the guidance from prof. Ninghui Sun, and prof. Yunji Chen. Currently she is an engineer of Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

**Wei Li** recieved the bachelor's degree in engineering from the Department of Electronic Engineering and Information Science, University of Science and Technology of China, Hefei, China, in 2005, and the PhD degree from the Institute of Electronics, Chinese Academy of Sciences, Beijing, China, in 2010, with the guidance from prof. Haigang Yang. She is currently an associate professor at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

**Xing Hu** received the B.S. degree from Huazhong University of Science and Technology, Wuhan, China, and Ph.D. degree from University of Chinese Academy of Sciences, Beijing, China, in 2009 and 2014, respectively. She is currently an associate professor of State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), Beijing, China. Her current research interests include domain-specific hardware architectures. She publishes over 30 papers in major conferences and journals including MICRO, HPCA, ASPLOS, DAC, DATE, TCAD, etc. She serves as reviewer for a number of journals and conference.

**Zidong Du** (Member, IEEE) recieved the bachelor's degree in engineering from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2011, and the PhD degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2016, with the guidance from prof. Yunji Chen, prof. Olivier Temam, and prof. Chengyong Wu. He is currently an associate professor at the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China.

**Qi Guo** (Member, IEEE) received the B.E. degree in computer science from Tongji University, Shanghai, China, in 2007, and the Ph.D. degree from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China, in 2012.,From 2012 to 2014, he was a Staff Researcher at IBM Research, Beijing. From 2014 to 2015, he was a Postdoctoral Researcher with Carnegie Mellon University, Pittsburgh, PA, USA. He is currently a Professor with the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include computer architecture, programming models, and machine learning.

**Yunji Chen** (Senior Member, IEEE) graduated from the Special Class for the Gifted Young, University of Science and Technology of China, Hefei, China, in 2002, and recieved the PhD degree in computer science from the Institute of Computing Technology (ICT), Chinese Academy of Sciences, Beijing, China, in 2007. Currently, he is a full professor at Institute of Computing Technology, Chinese Academy of Sciences. He leads his lab to develop neural network processors. Before that, he participated in the Godson/Loongson project for more than ten years, and was a chief architect of Godson-3 microprocessor. Yunji Chen has authored or coauthored 2 books and over 90 papers. He was a recipient of ASPLOS'14 and MICRO'14 best paper awards for advances in neural network processors.