Geom: A Hierarchical Spatial Architecture with Hybrid Dataflows for Graph Learning

***Abstract*—Graph convolutional network (GCN) emerges as a promising direction to learn the inductive representation in graph data that are commonly used in widespread applications, such as E-commerce, social networks, and knowledge graphs. However, learning from graphs is non-trivial because of its mixed computation model involving both graph analytics and neural network computing. To this end, we decompose the GCN to graph-level and node-level computing, and propose a GCN accelerator design, Geom, to facilitate efficient computing on both two parts. We incorporate a lightweight graph reordering algorithm and cache hierarchy design to mitigate the irregular memory accesses in graph-level computing. And we implement a mapping methodology aware of data reuse and task-level parallelism to handle various graphs inputs effectively. Results show that Geom accelerator design achieves 5.1x, 3.5x, and 7.8x of speedup compared to NN-like, Graph-like accelerators, and GPU across the diverse datasets on average.**

I. Introduction

With rich and expressive data representation, graphs demonstrate their applicabilities in various domains, such as E-commerce [7], [33], [45], computer vision [32], [44], and molecular structures [16], and *etc*. To fully exploit the their value, approaches based on traditional graph analytics algorithms (*e.g.*, BFS, SSSP) facilitate in-depth understanding of objects-wise relationships in graphs (*e.g.*, molecule similarity in chemistry [16], cells structures in bioinformatics [49], [50], and semantic graphs in computer vision [32], [44]). Recently, as a rising star, extending deep learning techniques to graph analytics has gained lots of attention from both research [10], [21], [27], [46] and industry [8], [51], largely because of their striking success on Euclidean data (*e.g.*, images, videos, text, and speech). Moreover, such geometric deep learning techniques based on graph neural networks (GNNs) not only learn the inductive representations for end-to-end tasks such as classification, clustering, and recommendation, but also show much better accuracy (more than 98%) than traditional methods [45] (*e.g.*, random walks [38], and matrix factorization [19]).

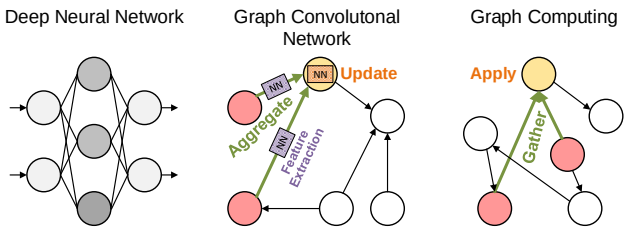

Fig. 1. Comparison between deep neural network (DNN), graph convolutional network (GCN), and graph computing.

Among various kinds of GNNs [21], [24], graph convolutional network (GCN) is the most fundamental model and has been widely studied. Different GCNs can be summarized and abstracted into a uniform computing model with two stages: *aggregate* and *update*. ***Aggregate*** stage collects the localized node representations from the neighboring nodes through two operations: 1) the *feature extraction* for a single node and 2) the *feature reduction* among neighboring nodes. The ***Update*** stage updates the representation vector with the aggregation results. As shown in Figure 1, GCN distinguishes itself by combining graph analytics and neural network computing scheme, where the feature extraction and update involve neural network computation while the feature reduction is essentially a graph traversal. Thus, GCNs computing suffers from two unique challenges:

Hybrid computing paradigm makes the existing NN accelerator and graph accelerator designs pale in effectiveness for handling GCN models. Specifically, NN accelerators aim to build spatial architectures [11], [12] for powerful computation capabilities while maximizing spatial data reuse based on Euclidean data. However, GCN localized computing (aggregation reduction) is based on non-Euclidean graph data, which is non-ordered with a diverse range of node sizes and topology with following issues: 1) Non-Euclidean data introduces irregular memory accesses and synchronization overhead. 2) Non-Euclidean data cannot be easily handled by spatial architectures for spatial data reuse with the statically configured vertical, horizontal, or diagonal dataflow. On the other side, graph accelerators prefer resource-intensive on-chip cache to suppress irregular accesses [20]. However, GCN has much higher dimensionality and complex operations inside node feature computation. For example, GCNs features high-dimension node/edge embedding (10x -1000x [21], [26] than that of traditional graph computing) with complex NN operation (*e.g.*, Multilayer Perceptron), while the traditional graph computing works with simple arithmetic operations (*e.g.*, addition) on nodes with scalar values. Therefore, an off-the-shelf solution to effectively accelerate GCNs is still missing.

Workload diversity and graph irregularity raise the difficulty to adaptively utilize the hardware capability. When the input graph has a larger feature dimension, the GCN computing shifts to NN computing and demands more multiply-and-Accumulator (MAC) arrays for powerful computation capability. However, when the input graph has a large number of nodes with high average degrees, the GCN computing demand shifts closer to graph computing that demands large on-chip buffer and date management methodology to eliminate the

irregular memory access. Hence, it is important to design an efficient task mapping methodology to bridge the gap between the diverse application demands and uniform hardware platform.

To this end, we first decompose the GCN computing into two parts and propose a uniform GCN accelerator design to facilitate efficient GCN training on both these two parts respectively: 1) node-level computing: intra-node computing (feature extraction and update) on Euclidean data; and 2) graph-level computing: localized neighboring reduction (aggregation reduction) on non-Euclidean data. Specifically, Geom incorporates a hierarchical spatial architecture with cache design to optimize the computing parallelism and data reuse in graph-level computing and node-level computing. To fully utilize the hardware capability, we propose a hierarchical mapping methodology to improve both the data reuse and task-level parallelism. Specifically, for localized neighboring reduction, Geom maps the node sets with the larger potentiality for non-Euclidean data reuse to the same MAC array based on the locality-sensitive hashing (LSH) clustering techniques. For the intra-node computing, Geom folds the matrix-vector multiplications to the MAC array to handle input graphs with different feature sizes. Cooperated with the intelligent task mapping, Geom significantly remove the off-chip memory accesses and eliminate the synchronization overhead.

Overall, we make the following contributions in this work:

- We present a computing paradigm for GCN workloads, which decouples GCN computing to node-level computing (intra-node feature computing) and graph-level computing (localized neighboring reduction).
- We design a GCN training accelerator, Geom, based on a hierarchical spatial architecture that supports this paradigm and the Euclidean/non-Euclidean data reuse in both node-level computing and graph-level computing.
- We incorporate the hierarchical task mapping strategies for intra-node computing and graph-level computing, which comprehensively optimize both data reuse and task-level parallelism to well adapt diverse datasets with different feature sizes and graph topologies to the hardware platforms.
- Intensive experiments and studies among Geom, graph-like accelerator, NN-like accelerator, GPU over a wide range of graph datasets show that Geom outperforms NN-like accelerator, graph-like accelerator, and GPU with 5.1x, 3.5x, and 7.8x speedup.

II. Background

In this section, we first introduce the GCN basics, the abstract computing model, and the variants derived from GCNs.

A. GCN Basics

The target of graph convolutional neural networks is to learn the state embedding of a graph property from the non-Euclidean input structure. Such state embeddings transform the graph features to low-dimension vectors which are used for graph classification and clustering [50]. In general, we define a graph, $G = (V, E)$, where V and E are vertex and edge sets, respectively; each node has node feature vectors X_v for $v \in V$; and each edge has edge feature vectors X_e for $e \in E$. On such a graph, GCNs learn a representation vector of a node (h_v), an edge (h_e), or the entire graph (h_G) with the information of the graph structure and node/edge features, so that the corresponding classification tasks can be conducted based on the representations.

In terms of the computing paradigm, GCNs has two main categories: spectral GCN [15], [23], [30] and spatial GCN [10], [21], [24], [34], [46]. The former are derived from graph signal processing and its mathematical representation is based on eigen-decomposition of graph Laplacian matrix [27]. However, spectral GCNs fall in short in several aspects: 1) The inability to perform inductive learning due to the fact that Laplacian decomposition is fixed to a specific graph; 2) The inefficiency to handle large graphs since it demands the decomposition for the entire graph adjacency matrix. On the other side, spatial GCNs emerge to learn the inductive representation based on the graph computing paradigm, which identifies the spatial aggregation relationships of nodes/edges. Therefore, spatial GCNs is capable to generate embeddings for unseen nodes, edges, or subgraphs. Moreover, spatial GCNs can process large graphs without compromising performance. In addition, previous works and in-depth studies [21], [24], [51] also demonstrate spatial GCN as a promising direction. Base on its potential of informativeness and powerfulness, we concentrate on spatial GCN for further exploration in this work.

B. GCN Computing Model

The GCN training process consists of the following three stages: forward propagation, loss calculation, and backpropagation. In forward propagation calculates node feature by iteratively incorporating the impact of its neighbors, which finally outputs the status of each node comparing against the ground truth for loss computation. The backpropagation finds the impact of each state on the loss by propagating from the last layer to the input layer based on the chain rule of the gradient. It is similar as the forward propagation but in a reverse direction.

Algorithm 1: GCN algorithms

1 **Inputs:** Graph (V,E); input features $\{X_v, \forall v \in V\}$, depth K; weight matrices $\{W^k, \forall k \in K\}$; aggregator functions $\{AGGREGATE_k, \forall k \in K\}$; neighborhood function $\{N : v \rightarrow 2^V\}$ **Output:** Vector representation z_v for all $v \in V$
$h_v^{(0)} = X_v$ **for** $k = 1...K$ **do**
2 **for** $v \in V$ **do**
3 $a_v^{(k)} = AGGREGATE^k(\{h_u^{(k-1)} | u \in N(v)\})$
4 $h_v^{(k)} = UPDATE^{(k)}(h_v^{(k-1)}, a_v^{(k)})$
5 **end**
6 **end**
7 $z_v = h_v^{(k)}$

We detail the process of the forward propagation by taking node classification as an example. The forward propagation stage of modern GCNs works in an iterative manner, as shown at the for-loop iteration at *line 4* in Algorithm 1. Assume a node v in Graph (V,E) with embedding $h_v^{(0)}$ that initialized as X_v

(*line 3*), and $N(v)$ refers to the set of v's neighbors. $a_v^{(k)}$ and $h_v^{(k)}$ are the aggregation results and the node embedding of v after the completion of the *k-th* layer of a GCN. The computation process of GCN repeats the following two steps: 1) Aggregate the node representation from the neighboring nodes (*line 7*); 2) Update the representation vector based on the aggregation results and its previous state (*line 9*). (some work also adopt the term of "Combine" instead of Update [43]). GCNs adopt both the concepts of graph computing and deep neural network techniques. And they exhibit the graph computing mode that aggregates the node representations from the neighborhood, and the neural network computing mode that extracts and embeds the features or hidden states of nodes/edges/subgraphs with deep neural network techniques. The forward propagation process is illustrated in Figure 2 which shows the cases with two iterations. The backward propagation process is similar to the process shown in Figure 2 by aggregating the gradient of $a_v^{(k+1)}$ when computing the gradient of $h_v^{(k)}$.

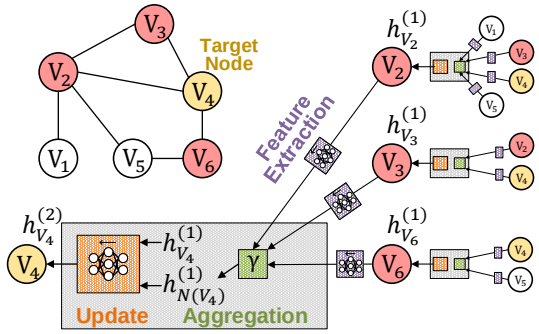

Fig. 2. GCN forward propagation flow with two iterations.

C. Other Variants

Many variants of the functions $AGGREGATE^{(k)}(.)$ and $UPDATE^{(k)}(.)$ have been proposed to improve the prediction accuracy or to reduce the computation complexity in GCNs. For example, convolutional aggregators are used in graph convolutional neural networks, attention aggregators are used in graph attention neural networks [48]. Gate updaters are adopted in gated graph neural networks or graph LSTM [47]. Although there are many variants of GCN models [49], they can be abstracted into the uniform computing model discussed in Section II-B. Hence, Without loss of generality, we focus on graph convolutional neural networks in this work.

III. Characterization and Motivation

In this section, we quantitatively characterize GCN workloads and provide insights on how they differ from graph computing and neural networks and their unique challenges. And we also highlight the hybrid computing paradigm and dual dataflow in GCNs, which is a key motivator of our accelerator design.

1) GCN vs. NN: The node-level computing in GCN is similar like NN which operates on the regular Euclidean data like images or text to obtain the highly expressive representations. The graph-level computing in GCNs, however, operate on non-Euclidean data of the input graphs, which introduces the graph traversal overhead. We conduct the execution time breakdown of GraphSage [21] forward propagation on GPU platform over six datasets, as shown in Figure 3. The details of these datasets are presented in Section V-A. The results show that 1) Graph-level computing (*Aggregation*) exhibiting graph traversal occupies a large proportion of the overall execution time. 2) Diverse workloads (*e.g.*, Citeseer-S vs. Reddit) with different feature size and graph scale have different bottleneck of graph-level computing (*Aggregation*) or node-level computing (*Update*) .

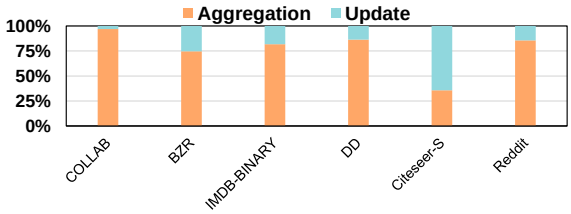

Fig. 3. Execution time breakdown of GraphSage: Aggregate vs. Update.

Graph-level computing could bring two challenges: **Irregular memory accesses.** Graph traversal during the aggregation of the feature data of neighboring nodes introduces a large volume of irregular memory accesses. This is in contrast to NNs on Euclidean data where the memory access patterns are hardware-friendly because it can be regularly transformed to matrix/tensor computations. Hence, GCNs are more memory-bounded compared to NNs. This is further confirmed by our VTune analysis in Figure 4. NNs execution are more dominated by their computations. The smaller networks, such as Lenet, are mainly computing bound. The larger networks, such as ResNet18 and VGG19, have more memory stall time because of the larger weight and feature map matrices. Thus, lots of data flow optimizations are proposed to enlarge data reuse and eliminate the off-chip memory accesses [12]. However, such data reuse methodologies are not applicable to GCN graph-level computing which conducts computations on non-Euclidean data with irregular memory accesses. **Synchronization stalls.** Synchronization stalls are introduced by irregular memory accesses. Specifically, in *Aggregate* function, all the feature data of in-neighbors during graph traversal should be in ready status before conducting the aggregation reduction operations. If the feature data of one neighbor misses, the aggregation function needs to wait even when all the other neighbor feature data is ready in the cache. Therefore, the synchronization among neighbors is another key source of the computation overhead. Such synchronization contributes to the computation stalls in Vtune analysis in Figure 4.

2) GCN vs. Graph: Graph analytics possesses a similar programming model as the graph-level computing in GCNs. As shown in Figure 1, in a pull-implementation programming model of graph computing, processing a vertex involves three steps: 1) *Gather*: reading properties from the in-neighbors of vertex; 2) *Apply*: processing the update operation on the gathering properties. The functionalities of *Gather* and *Apply* are similar to *Aggregate* and *Update* operations in GCNs. However, at the node-level computing, GCN and graph analytics have different data dimensions and computation

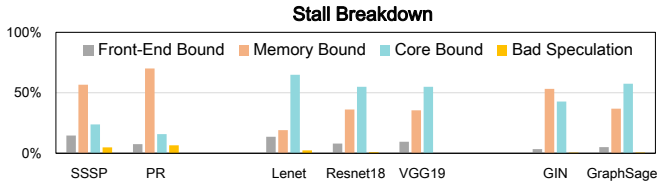

Fig. 4. Stall breakdown of Graph, NN, and GCN applications.[1]

TABLE I
GRAPH COMPUTING VS. GRAPH NEURAL NETWORK.

	Graph Computing		Graph Neural Network	
	SSSP	PR	Citeseer	Reddit
Property/Feature Size	4Bytes	4Bytes	3703*4Bytes	602*4Bytes
Feature Extraction	/	/	DNN	DNN
Aggregate	Scalar	Scalar	Vector Reduction (1024/128)	Vector Reduction (1024/128)
Update	Scalar	Scalar	DNN	DNN

complexities of nodes/edge features, as compared in Table I. GCNs, in general, have much larger feature data size (10x – 1000x) than the property data in graph analytics applications. In addition, *Aggregate* and *Update* operations in GCNs have significantly larger computation complexity due to neural network computing, while that in graph analytics usually consist of simple scalar computations.

Because of the different data dimensions and processing complexity, graph analytics and GCNs exhibit different bottlenecks at the architecture level. First, graph applications are mainly memory-bound because of the irregular accesses and simple update computation [5], whereas GCNs demand more computation capabilities. Second, typical prefetch-oriented optimizations in graph analytics (*e.g.*, prefetching property data on to on-chip buffers for better memory access efficiency) are less effective in GCN applications. This is because less node feature data can be prefetched than that in graph analytics due to the larger feature data size (hundreds to thousands of times larger than property data in graphs). In summary, GCN is confronted with the following challenges: irregular memory accesses and synchronization overhead introduced in graph-level computing; larger computational complexity in node-level computing. This motivates our accelerator design.

IV. GEOM DESIGN

To address the challenges in GCNs, we decompose the GCN computing into graph-level and node-level computing and propose, Geom, a hierarchical spatial architecture to efficiently support both of them (Figure 5(d) and (e)). Furthermore, we propose a hierarchical task mapping methodology to improve the non-Euclidean data reuse and task-level parallelism in graph-level computing and Euclidean data reuse in node-level computing (Figure 5(b) and (c)). This mapping methodology well adapts diverse applications with variable node feature dimension and graph topology into the hardware platform.

A. Computing Paradigm with Hybrid Dataflows

There are two dataflows in GCNs. **Euclidean dataflow in node-level computing:** Taking the illustrative case in Figure 6(a) as an example, during the embedding calculation of *node* 4, feature extraction and update operations exhibit Euclidean dataflow. Thus the spatial architectures that exploit high compute parallelism using direct communication between processing elements (PEs) can be used to optimize the data reuse in either vertical, horizontal, or diagonal directions [12]. **Non-Euclidean dataflow in graph-level computing:** The aggregation reduction operation, however, involves the graph traversal and computations for non-Euclidean data. What also worth notice is that aggregation function offers two types of data reuse opportunities to be explored for potential performance improvement: *input feature data reuse* and *partial aggregation results reuse*. 1) Input feature data reuse: As shown in Figure 6(b), the feature data of *node* 2 is repetitively used when conducting neighbor aggregation for *node* 4, *node* 1, *node* 3, and *node* 5; 2) Intermediate aggregation results reuse: Since the *node* 4 and *node* 5 have the shared neighbor sets: *node* 2 and *node* 6, the intermediate aggregation results of *node* 2 and *node* 6 can be reused when computing the aggregation for *node* 4 and *node* 5 because the aggregation reduction operations are based on *sum*, *average*, or *min/max*, which are order invariant. Although there is a large volume of potential data reuse, it can hardly be handled by the spatial architecture with the Euclidean data reuse path, because there is no specific order and geometric data reuse direction for such computations. Thus, we optimize the temporal reuse of non-Euclidean data in the cache hierarchy.

B. Accelerator

The architectural design of Geom is demonstrated in Figure 5. Geom is mainly comprised of following basic components: processing element (PE) array, memory controller, global on-chip buffer, and control logic.

The overall PE array is hierarchically organized based on MAC arrays. The MAC array is the spatial architecture that commonly used in NN accelerators [12], [25] capable of direction communication for spatial data reuse. The PEs are connected in a 2D-mesh in the multi-core way. The graph-level computing tasks are dispatched to PE array and node-level computing tasks are scheduling inside the MAC array. The global on-chip buffer stores the data shared by the whole PE array, such as the weight matrices of GCNs. We next describe the design details of these components and the data reuse strategy in Geom.

1) PE Array: The PE array consists of multiple PEs connected with the 2D-mesh network on chip (NoC) interconnections. The leftmost and rightmost PEs in the NoC mesh communicate with the memory controller directly. The other central PEs get the read and send write requests through the 2D-mesh NoC. All the traffic between PEs and memory goes through the NoC network in a first-come-first-serve manner

[1]For neural network workload, three neural network models in different scales are selected and evaluated based on Cifar10 dataset [28], including Lenet [29], VGG19 [41], and ResNet18 [22]. For graph neural network workload, we test GIN [43] and GraphSage [21], on COLLAB graph dataset [26]. For graph workload, two graph analytic applications in the GAP Benchmark Suite [6] including Single Source Shortest Path (SSSP) and Pagerank (PR) are profiled using the LiveJournal (LJ) [3] dataset.

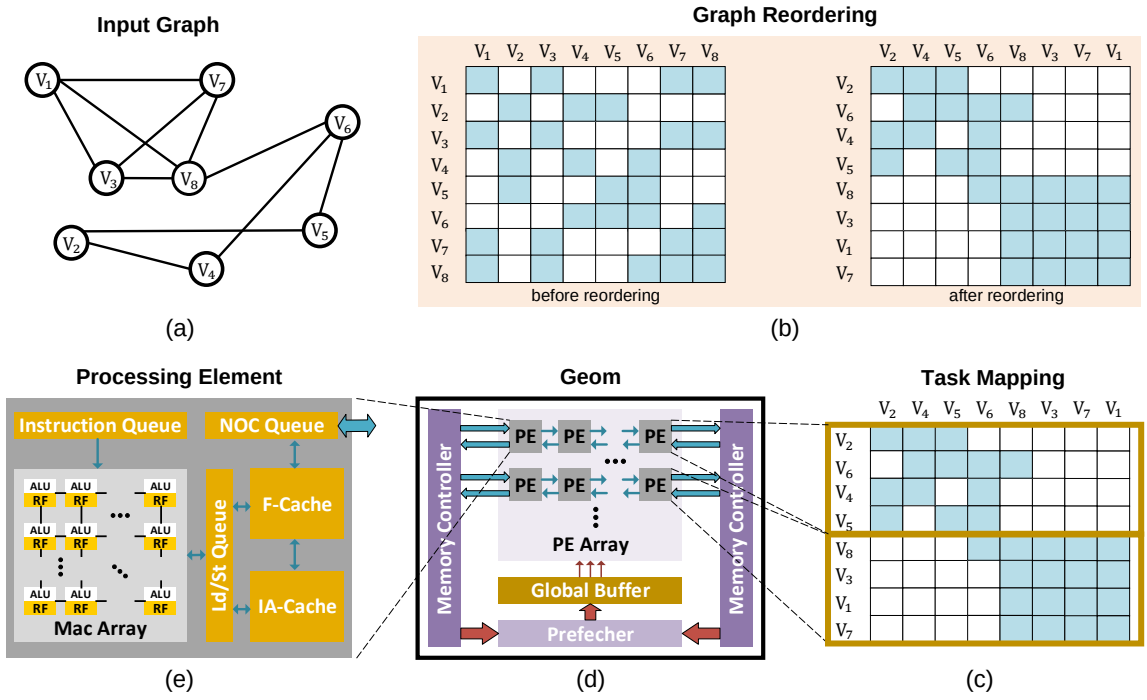

Fig. 5. Design overview of Geom: (a) An example of input graph; (b) Graph reordering to group the nodes with more shared neighbors together; (c) Mapping reordered graph to PE array; (d) The top-level architecture of Geom for graph-level computing; (e) The composition of one PE in Geom for node-level computing.

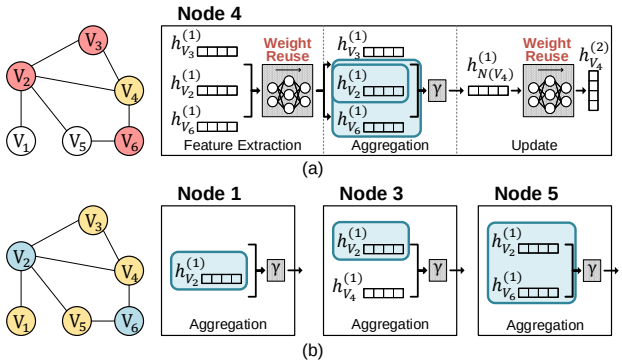

Fig. 6. Two types of data reuse: (a) Euclidean data reuse in node-level computing; (b) Non-Euclidean data reuse in graph-level computing (including feature data reuse and intermediate aggregation result reuse).

with the one-way routing strategy. Specifically, there are two memory controllers in Geom at the left and right side of PE arrays. The access location of a memory request is determined by the memory address. Once the access location is determined, the memory request is transferred through the NoC at either left-horizontal or right-horizontal directions.

2) PE Design: The detailed design of PE is shown in Figure 5(e), which consists of the instruction queue, the load-store queue (LSQ), the NoC queue, the multiply and add accumulator (MAC) array, and two private caches for data reuse of non-Euclidean data. Instruction Queue buffers the micro-instructions including three major categories: load, store, and computation. The entire GCN training process can be translating to hardware primitives according to the input graphs. The micro-instructions can be generated by the driver and prefetched to the instruction queue with the streaming strategy for good access efficiency. LSQ buffers the load and store requests for accessing the feature extraction data, aggregation data, and update data.

3) On-chip Memory Hierarchy: Hierarchically, the on-chip memory consists of several layers that handle different data reuse granularities: global buffer, PE private cache, and MAC register files (RFs). Global buffer: Global buffer exploits data reuse between PEs such as the weight matrices. PE private cache buffers the feature map data and intermediates aggregation data for input reuse and partial sum reuse of non-Euclidean data in Figure 6. Such data reuse is in a temporal manner based on the cache implementation. Tasks in different PEs do not have non-Euclidean data reuse in these private caches. Thus, during task mapping, we should carefully do the tradeoff between task parallelism and data reuse efficiency. MAC register files are similar to that in NN accelerators which exploit all types of data movement within the computations of one node, including the convolutional reuse and filter reuse. Except the weight metrics, all other store requests are write-through and sent back to the memory controller directly without on-chip buffering.

C. Programming Model and Hardware Primitives

To generally support diverse GCN algorithms, we adopt a vertex-centric programming model, since most graph neural network are based on this model [10], [13], [21], [24]. Based on the vertex-programming model, we provide the following hardware primitives to support the execution of GCNs in Algorithm 1: *load-f*, *load-i*, *comp*, and *store*. The first two

primitives, *load-f* and *load-i*, are used to load and aggregate the feature vector of single node and the intermediate aggregation result of two nodes. The third primitive, *comp*, is used to invoke the computation of feature extraction and update function, which is usually composed to matrix-vector multiplication and some element-wise computation instructions. After computing the feature vector of a node v for the k-th layer ($h_v^{(k)}$), the *store* primitive is used to flush the result of computation into memory so that it is visible to other PEs in the iteration of $(k+1)$-th layer.

Vertex-centric programming models have no need to worry about the data conflict issue in edge-centric programming, but confronted with synchronization issues during execution. Such overhead is introduced when the node update operation is blocked due to waiting for neighbors to be aggregated. Thus, we propose a graph reordering method and intelligent mapping to alleviate irregular memory access effect and reduce synchronization overhead, while retaining the task-level parallelism. The graph reordering and mapping stage generate two inputs to the hardware accelerator. The first input is the task assignment with the ordered vertex ID. Each PE is assigned with a set of vertices to compute (from *line 5* to *line 10* in Algorithm 1). The second input is the indicator for the reuse of the intermediate aggregation results, which generates the *load-i* instructions. The hardware accelerator executes these hardware primitives generated by the reordering and mapping stages, and the reordering and mapping stage are responsible to exploit the locality of feature vectors and the computation reuse of partial intermediate results.

D. Task Mapping Generation

In this section, we introduce a lightweight graph reordering methodology which improves the temporal reuse distance of the non-Euclidean data. With the reordered graph, we propose the mapping strategy combining the data reuse optimization and task-level parallelism.

LSH-based Graph Reordering. Graph reordering techniques exploit the structural properties to reassign index ID to vertices, so that the layout changes to improve the locality of graph data [4]. In our work, we propose a lightweight off-line graph reordering technique based on locality-sensitive hashing (LSH) to determine order and mapping location. The goal of reordering in our work is to group the nodes with more shared neighbors together to improve the temporal reuse when conducting the aggregation reduction. The intrinsic reason that the reordering method can provide better temporal reuse is based on the fact that real-world graphs exhibit a "community structure" [18], which means that some nodes share neighbors or have a closer relationship to each other. Therefore, by grouping them together, the data locality during execution will be significantly improved. Note that graph reordering does not change the graph structure but only affects the execution order of the graph. We develop the graph reordering algorithm by synergistic *Locality-Sensitive Hashing* and *Row-Column Ordering*.

1) Locality-sensitive Hashing: Locality-sensitive hashing (LSH) is an algorithmic technique, being widely used to solve the approximate or exact nearest neighbor problem in high dimension space [1], [2], [14]. It groups similar items into the same "buckets" with high probability. The basic concept based on random projection: for every input vector x, the hash function is calculated by projecting this vector x to several random vectors. With a series of random vectors, LSH maps an input vector to a bit vector (buckets). Input vectors with smaller distances have a higher probability to result in the same cluster with the same bit vector.

2) Reordering Flow: We leverage LSH to cluster the nodes with more shared neighbors. Every row in the adjacency matrix of the graph is a vector that represents the neighbor connections for this vertex. Taking these vectors of rows in the adjacency matrix, LSH hashing groups the rows into several clusters. Taking the detailed example in Figure 7 for illustration, Row-02 and Row-04 are grouped in the cluster because they share most of the neighbors and have similar row vectors. Similarly, Row-05 and Row-08 are grouped in a cluster. Row-03, Row-01, Row-07, and Row-08 are grouped in a cluster. Thus, after the row transformation in step 1, we have the transformed graph with the nodes assigned to the same buckets being placed continuously. Then we conduct the LSH clustering to every column in this transformed adjacency matrix (step 2 in Figure 7). Similarly, Col-02, Col-04, and Col-06 are grouped together. Col-05 and Col-08 are in one cluster. Col-03, Col-07, Col-01 are grouped in one cluster. Thus the columns are transformed by placing the columns in one cluster continuously. This process can be conducted iteratively to achieve better reordering effectiveness. The iteration number is a tradeoff between the processing overhead and the reordering effectiveness. Based on our empirical study, one iteration is sufficient for most of the graphs. We further discuss the computation complexity in Section VI.

Shared Node Set Exploration. Based on the reordered graph, we explore the reuse potentiality of the intermediate aggregation results. The basic idea is to find the shared node sets in the window of neighboring rows. In this work, the node-set granularity is set to 2 for the simplicity of implementation. A simple example is illustrated in Figure 7(b), where *node* 2, *node* 6 share the neighbor of *node* 4 and *node* 5. Therefore, the intermediate results of *node* 4 and *node* 5 can be reused for *node* 2 and *node* 6 aggregation computation. Similarly, the intermediate results of *node* 1 and *node* 7 are being reused for *node* 3 and *node* 8.

Mapping Methodology. Based on the reordered graph, we map the tasks onto the Geom accelerator in a hierarchical manner. Specifically, Task mapping first allocates the node sets to every processing element. Then the intra-node computations are organized into MAC arrays.

1) Graph-level mapping: The mapping strategy of allocating vertices to PEs considers both data reuse and task-level parallelism. In the adjacency matrix of the transformed input graph, the destination nodes of neighboring rows have similar source nodes for aggregation, which introduces both the input data reuse and intermediate aggregation result reuse. To improve the data reuse along with their execution, we evenly partition the

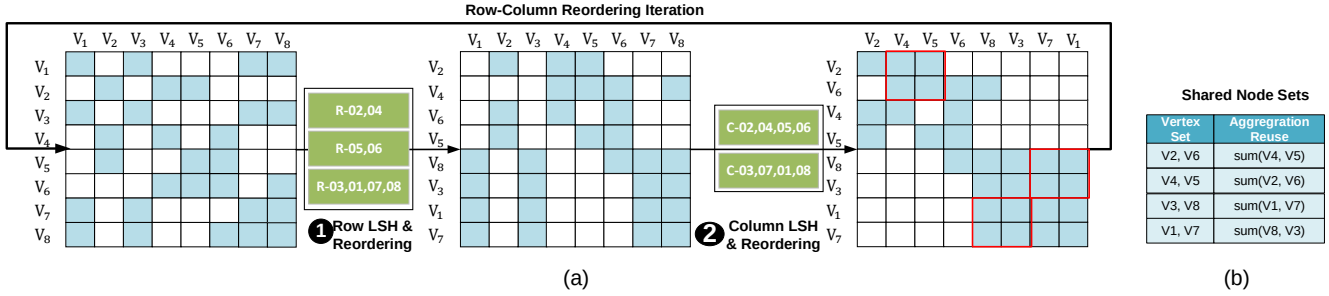

Fig. 7. LSH-based row-column reordering: (a) The reordering process; (b) Shared node set exploration.

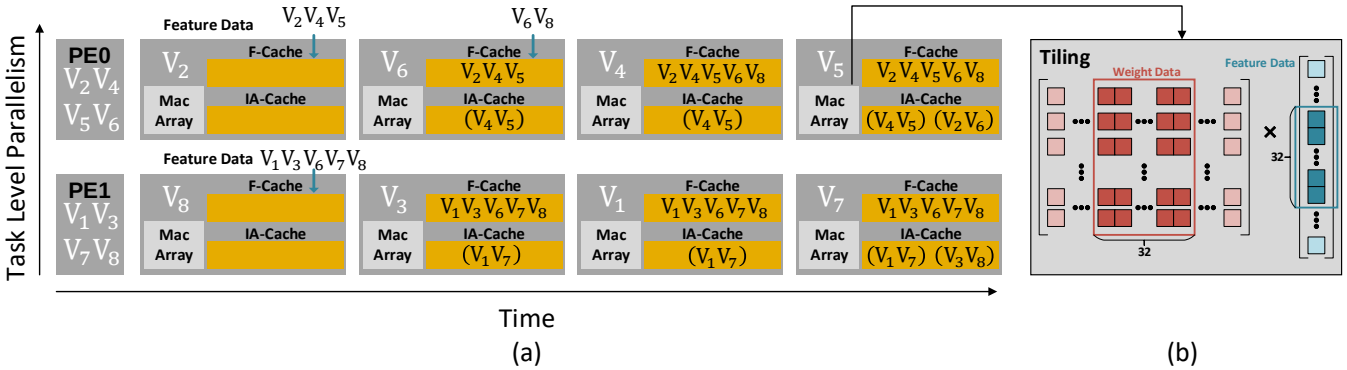

Fig. 8. Hierarchical task mapping: (a) Node sets allocation; (b) Intra-node task tiling.

rows to 64 sets (the number is equal to that of PE number). Then allocate every set to an individual PE for parallel computation. Such a process is illustrated in Figure 8 (a) with a simplified example. After graph reordering, V_2, V_4, V_5, and V_6 are allocated in PE0, while the V_1, V_3, V_7, and V_8 are allocated in PE1. Tasks in PE0 and PE1 can be computed in parallel. In PE0, V_2, V_4, V_5, and V_6 will be executed sequentially. When conducting the aggregation operations for V_2, feature data of V_4 and V_5 are obtained from off-chip memory and buffered in F-cache. Since there is an indicator of shared node sets of (V_4, V_5), the intermediate aggregation results of them will be stored in IA-cache for further reuse. When conducting the aggregation operations for V_6, feature data of V_2, V_8, and intermediate results of (V_4, V_5) are needed. V_2 and (V_4, V_5) are hit in F-Cache and IA-Cache respectively, therefore we only need to get the feature data of V_6 and V_8. When computing the aggregation and update operations of V_4 and V_6, all the feature data of neighbors are in cache and no off-chip memory traffic is introduced during computation. Such an example shows that the tasking mapping based on the reordered graph improves the temporal reuse distance for the vertices in the same PE.

2) Node-level mapping: For the feature computation inside nodes (feature extraction and update), we tile the vector-matrix multiplication onto the MAC arrays for a better data reuse. Such mapping and tiling techniques have been well studied in previous work [11], [12]. We adopt a similar methodology, as shown in Figure 11(b). The matrix-vector multiplication is partitioned to several blocks according to the computation capability of MAC arrays.

E. Computation Process

In this section, we introduce the computing and data reuse process of the whole forward propagation. The backpropagation is similar but in a reverse way. As introduced in Section II, the whole processing pipeline of the forward propagation is comprised of feature extraction, aggregation reduction, and update. The detail forward propagation computation and dataflow are shown in Figure 9.

In summary, the data reuse in Geom can be generally classified into two categories: reuse of non-Euclidean data and Euclidean data. For the computation inside a node (Euclidean data), such as feature extraction and update stages, the feature map data and weight matrix are reused in MAC arrays. For the computation on the node set for aggregation (non-Euclidean data), the feature data is stored in the private cache of every PE for temporal reuse.

Feature Extraction. Feature extraction for nodes is conducted at the beginning of every iteration. During this process, the feature data of nodes are streaming in and streaming out to memory systems. Weight data is stored onto the global buffer and reused for the feature extraction of every node.

Aggregation Reduction. After the completion of feature extraction, Geom conducts aggregation reduction for every node, by loading and computing the feature data of its node neighbors. The feature data is buffered in the private cache of PE. Along with the aggregation for the nodes in the input graph, there is a temporal reuse of feature map data in the private cache after the graph reordering. Such temporal data reuse reduces off-chip memory accesses and the Section V-C

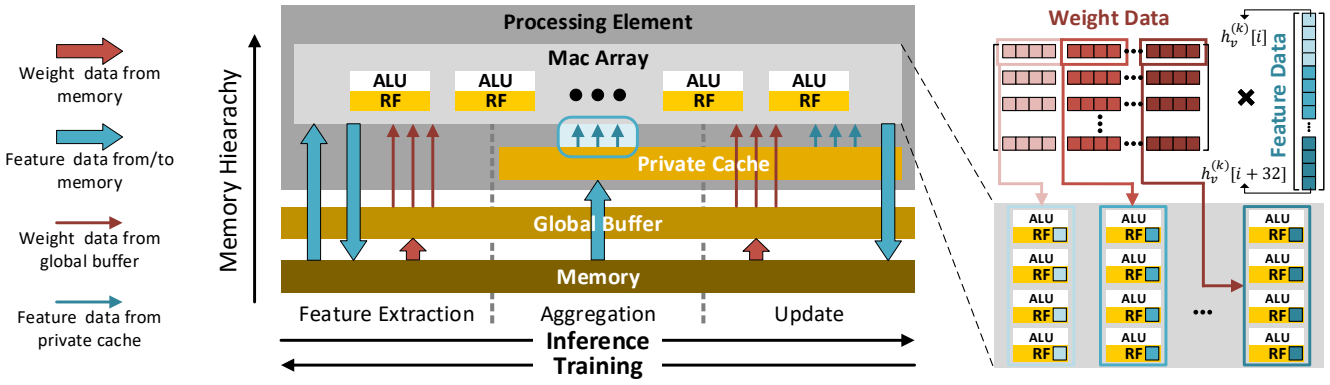

Fig. 9. Dataflow in Geom: an example of weight data, feature data, and intermediate aggregation results reuse across four layers of memory hierarchy.

discussed the effect with a quantitative analysis. Specifically, there are two types of reuse for non-Euclidean data: 1) the feature data of every node, which is the input of aggregation reduction function, is stored in F-Cache. 2) The intermediate aggregation results of node sets, which are stored in IA-Cache. For the graphs with large degrees, IA-Cache is helpful to reduce the computations of aggregation significantly.

Update. With the aggregation results of a node, the update operation is conducted which calculates with the input of the aggregation results and the weight metrics. During the node computation, the weight data and feature data are reused in MAC arrays and the global buffer. The final result of the updated feature data will be written through to off-chip memory directly.

V. Experimental Results

In this section, we first introduce the experimental setup and then compare the performance and energy of Geom to graph accelerator, NN accelerator, and GPU platforms. After that, we analyze the performance influence of graph reordering and mapping methodologies. Finally, we analyze the impact of embedding size and graph degree on performance and show that Geom is capable of adapting diverse applications on the hardware platform.

A. Experimental Setup

GCN Datasets. Our graph accelerator evaluation covers a wide spectrum of mainstream graph datasets, including benchmark datasets for graph kernels [26], and datasets commonly used by previous studies [13], [21] in related domains. Details of these datasets are listed in Table II. Citeseer-S is a synthetic benchmark of the citeseer used in graph analytical and GCN benchmark [40], which has 227320 vertices with the dimension of 3703. Such a relatively large graph with a high dimension is built to test the hardware capability.

GCN Models. In this work, we mainly test on two commonly-used graph convolutional neural network models: GIN [43] and GraphSage [21]. We use the default configuration in broadly-used GCN library (Pytorch geometric library [17]), where GraphSage has 2 sageConv layers with hidden dimension = 256, GIN has 5 sageConv layers and 2 linear layers with hidden dimension = 128.

TABLE II
Graph Datasets

Dataset	#G	Avg.#V	Avg.#E	Dim	#Class
COLLAB	5000	74.49	2457.78	492	3
BZR	405	35.75	38.36	53	2
IMDB-BINARY	1000	19.77	96.53	136	2
DD	1178	284.32	715.66	89	2
Dataset	*#G*	*#V*	*#E*	*Dim*	*#Class*
CITESEER-S	1	227,320	814,134	3,703	41
REDDIT	1	232,965	114,615,892	602	6

Hardware Platforms. For the power and area evaluation of NN, Graph, and Geom accelerators, we break down the circuit model estimation to the compute logic, memory array, and hierarchical wires. We use results from RTL synthesis by Design Compiler using 45nm technology for MAC array and control logic. We use CACTI [42] and Micron Power Calculators for SRAM and DRAM estimation. We use McPAT [31] to estimate the NoC interconnection area and power. We conservatively run the accelerator at 500Mhz which comfortably satisfies the timing restraints. Specifically, we compare the performance and energy efficiency of the following hardware architectures, with the detailed configurations shown in Table III:

1) NN-like Accelerators (NN): We implement an NN accelerator similar to Eyeriss [12]. Compared to Geom, it has larger MAC arrays in every PE and has no private cache for graph traversal data buffering. The dataflow is similar to Eyeriss which enables MACs to support data reuse. The configuration of the NN accelerator is shown in Table III.

2) Graph-like Accelerator (Graph): We tailor the graph-like accelerator to execute the graph convolutional neural networks. The graph accelerator closely resembles a prior graph accelerator [20] which is equipped with a large on-chip buffer and the processing array to deal with the matrix-vector multiplication. Compared with Geom, it has a larger on-chip buffer with reduced computation capability. Graph accelerator has 16 MACs in every processing element with on-chip buffer and private cache of storing feature data and intermediate aggregation results. The graph accelerator has the same task mapping strategy as that in Geom for a fair comparison.

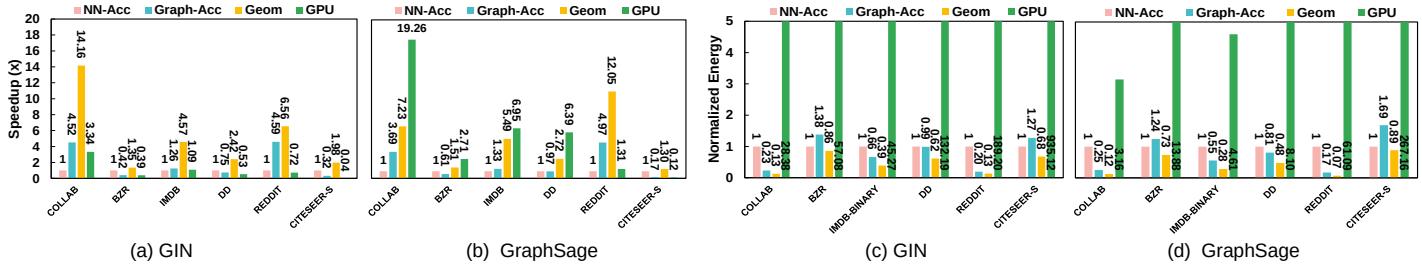

Fig. 10. Speedup and energy comparison for NN-like accelerator, Graph-like accelerator, Geom, and GPU.

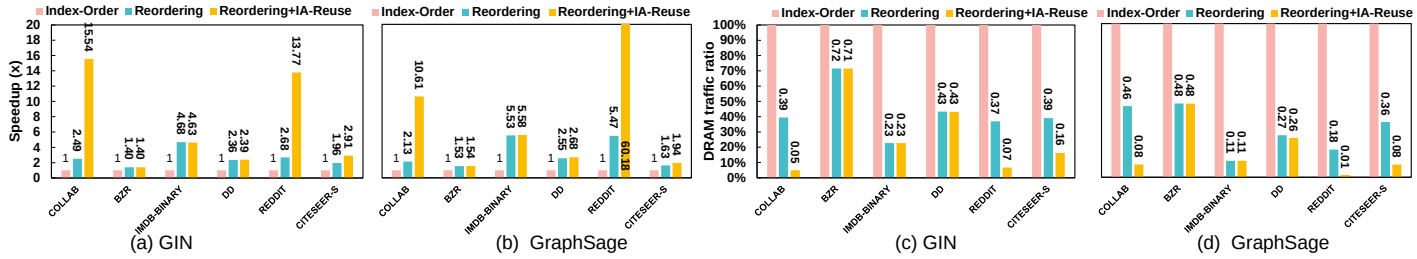

Fig. 11. Speedup and off-chip memory traffic reduction under different mapping Strategies.

TABLE III
HARDWARE PLATFORM CONFIGURATIONS

		NN	Graph	Geom	GPU
Comp	PE Array	8x8 PEs	8x8	8x8	3840 Cores
	MAC Array	16x16 MACs	1x4	4x8	
Mem	Mem BW	32GB/s			432GB/s
	Global buffer	2 MB	4 MB	2 MB	L2: 3MB
	Private Cache	–	256KB/PE	128KB/PE	L1:48KB/SM
	RegisterFile	16KB/PE	256B/PE	2KB/PE	RF: 48K/SM

3) Geom: We implement a cycle-accurate simulator to evaluate the total execution latency (cycles). This simulator models the modules in the architecture design, including PE, NoC, on-chip buffer, private cache, etc, as introduced in Section 4. The configuration of Geom is shown in Table III.

4) GPU: In addition to the accelerators, we also evaluate the GCN performance on NVIDIA Quadro P6000 platform with 3840 CUDA cores, peak single-precision performance of which is about 432GB/s. It is equipped with 24GB device memory capacity through GDDR5X, the bandwidth of which is about 432 GB/s. The GCN implementations are based on Pytorch Geometric library [17]. The GPU performance is estimated by accumulating the kernel execution time obtained from NVIDIA profiling tool suite [35] which eliminates the memory copy time and system stack overhead.

B. Overall Experimental Results

We compare all architectures mentioned in Table III in terms of performance and power. Due to the maturity of the NN accelerators and their success in real-world productions, we set the NN accelerator as the baseline of all studied architectures. The detailed breakdown analysis of the improvement from our mapping methods is discussed in Section V-C.

Performance. We evaluate the execution latency of training the entire graph for one epoch and compare it with the baselines, as shown in Figure 10(a) and (b). Overall, Geom shows speedups of 1.43x to 6.19x, 1.35x to 14.16x compared to *graph* and *NN* accelerators when running GIN model. Meanwhile, Geom achieves 1.96x to 7.74x, 1.30x to 12.05x of speedup when running GraphSage. Specifically, we provide the following insights: Diverse input graphs with different embedding size and degree distributions favor different architectures.

1) *Training input graphs with large feature sizes and lower degree shifts to NN computing mode and favors more computing resources*. For example, BZR, DD, and Citeseer-S have an average degree of 1.1, 2.5, 3.6, *NN* accelerator performs better than the graph accelerators.

2) *Optimization for non-Euclidean data reuse plays a much more important role for training input graphs with a larger average degree*. For example, COLLAB, IMDB-binary, and Reddit have an average degree of 32.8, 4.8, and 492. Thus, *Graph* accelerator performs better than *NN* accelerator. The proposed architecture, *Geom* outperforms the *graph* and *NN* accelerators no matter how the workload sizes vary.

We further compare Geom with the GPU platform and provide the following observations.

1) *Larger graphs with high dimension size and node volumes are more performance-sensitive to the data reuse optimizations*. When training GraphSage models, Geom achieves 9.18x and 10.76x of speedup for Reddit and Citeseer-S with a large graph scale. While GPU outperforms Geom when training small graphs, such as COLLAB, BZR, IMDB-BINARY, and DD. The key reason is that their memory footprint is too small and most feature data and weight data can be held in the on-chip memory hierarchy thus training GCNs becomes computing-bound. For larger graphs, the feature data of nodes cannot be held in the on-chip hierarchy. Additionally, in GCNs, most of the operations are based on matrix-vector multiplication, which has a much larger mem/compute ratio than that of matrix-matrix multiplications. Thus the data reuse optimization plays a more

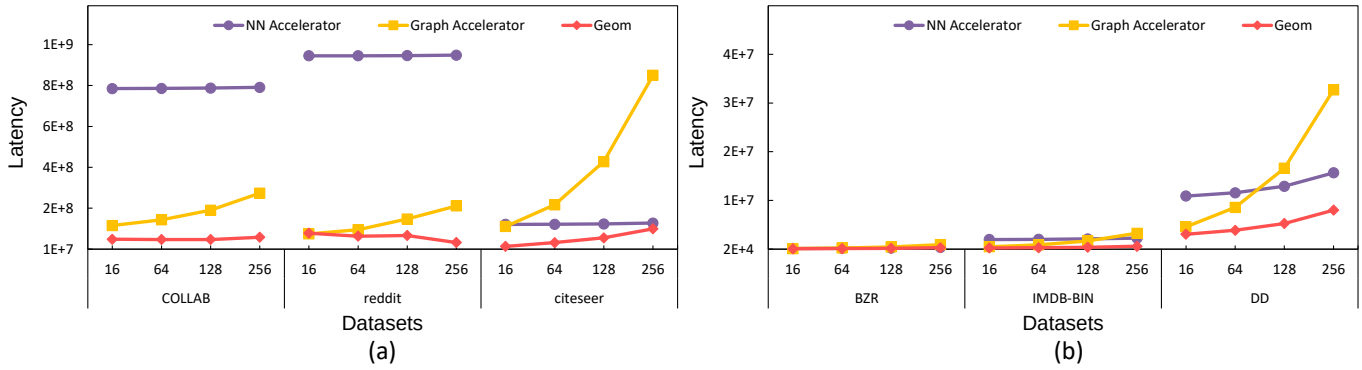

Fig. 12. Impact of feature dimension size for NN-like accelerator, Graph-like accelerator, and Geom.

important role for larger graphs. Consistently, Geom achieves a larger speedup compared to GPU platform for the dataset Reddit and Citeseer-S.

2) *Deeper GCN models are more performance-sensitive to the data reuse optimizations*. GIN model, which has deeper layers (5 Sageconv layers and 2 linear layers) than that in GraphSage (2 SageConv layers), Geom achieves the speedup of 3.42x to 4.52x compared to the GPU platform even on small graphs (COLLAB, BZR, IMDB-BINARY, and DD). Overall, Geom achieves the speedup of 3.42 to 46.7x of speedup across the various datasets when training GIN models.

Energy Consumption. In addition to performance comparison, we compare the energy consumption of Geom, graph accelerator, NN accelerator, and GPU. Energy consumption is calculated by multiplying the average power and the execution time. GPU power is sampled by *nvidia-smi*, which is the tool suite provided by NVIDIA CUDA driver. Compared to GPU, Geom improves energy efficiency by 26.3x to 1375.2x across different datasets and GCN models. Compared to NN-like accelerators, Geom improves energy efficiency by 1.47x to 7.92x for GIN and 1.13x to 8.20x for GraphSage. For graph-like accelerators, Geom improves energy efficiency by 1.60x to 1.87x for GIN and 1.69x to 2.52x for GraphSage. Such a relatively smaller energy consumption gap from the graph-like accelerator than the gap from the NN-like accelerator is caused by the large proportion of energy consumption on the on-chip cache and DRAM memory access.

Area. Geom mainly consists of the following components: computation logic, on-chip buffer and queues, hierarchical interconnection, and control logic. The computation units mainly comprise of the MAC arrays. The on-chip buffer and queues include the LSQ, instruction queue, F-Cache, IA-Cache, global buffer, and register file, as described in Table III. In summary, under the technology process of 45nm, Geom has an area of 36.86 mm^2.

C. Mapping Optimization

Geom incorporates both the hardware accelerator design and mapping methodology based on the reordered graph. In this section, we analyze the impact of graph reordering which aims to improve the data reuse of non-Euclidean data. Specifically, we compare the following three strategies on Geom platform: 1) $Index-order$: compute with the index order of nodes; 2) *Reordering* : compute the nodes in the reordered order after graph reordering; 3) $Reordering\&IA-reuse$: based on the reordered nodes, reuse the intermediate aggregation (IA) results.

Performance Comparison. We compare the speedup of the latter two strategies over the first one, as shown in Figure 11(a) and (b). We make the following observations: 1) Reordered graph generally improves the performance with the speedup of about 3.14x and 2.59x for GraphSage and GIN, across the datasets with different degree distributions and feature dimension sizes. 2) For input graphs with larger degrees, reusing the intermediate results ($Reordering\&IA-reuse$) brings significant speedup. As shown in Figure 11, COLLAB has an average degree of 32 and it achieves 15.5x speedup by reusing the aggregation results during GIN training.

Off-chip Memory Traffic Reduction. We further analyze the off-chip memory access reduction with dataflow optimization. The off-chip memory access volume of these three strategies is shown in Figure 11(c) and (d). Generally, compared to index-order execution, graph reordering reduces 69% and 58% of the off-chip memory access traffics for GraphSage and GIN. For the large sparse graphs with the large average degree, such as COLLAB and Reddit, the intermediate aggregation reuse eliminates more than 90% of memory accesses in the further step. These results show that optimization for non-Euclidean data significantly reduces the memory traffic and improves the memory access efficiency.

D. Impact of Graph Features

In this subsection, we analyze the impact of graph topology and feature dimensions on different hardware platforms. Specifically, we compare the six datasets with different scales and feature dimensions on *Graph* accelerator, *NN* accelerator, and *Geom* accelerator, as shown in Figure 12. We came with the following observations: 1) *Graph* accelerator performance is very sensitive to the feature input and output dimensions. With the increase of the feature dimension, the execution latency increases significantly, especially for the graphs with large input feature dimensions. For example, Citeseer-S has feature data with 3703 dimensions, with the increase of the output dimension, the *Graph* accelerator execution latency significantly deteriorates because the GCN demands more MACs for matrix-vector multiplication; 2) *NN* accelerator

is memory bound during GCN model execution. Taking the Reddit in Figure 12(a) as an example, the execution latency of NN accelerator stays still even the output dimension scales from 16 to 256. The NN accelerator performance does not change with the varying dimension size, which indicates that the computation capability is under-utilized and the bottleneck is the memory overhead incurred by the graph irregularity.

In summary, GCNs favor *NN* accelerators with powerful computation capabilities and optimizations for spatial data reuse when the input graph has a high feature dimension, while GCN demands shift to *Graph* accelerators with larger on-chip memory when input graphs exhibit high irregularity and complex topologies. *Geom* accelerator, accompanied by the mapping methodology, handles the diverse demand across these datasets, which well adapts the diverse applications to the hardware platform.

VI. Discussion

Graph-Reordering Overhead. In this work, the graph reordering is happening in the pre-processing stage for only one time. It is based on row and column transformation according to the LSH clustering results. LSH clustering is lightweight and friendly for hardware parallelization. For the Reddit dataset with 232,965 nodes, the graph reordering only requires several seconds to complete. We compare the performance between GPU and Geom with/without preprocessing overhead under the training scenario with 100 epochs, as shown in Figure 13. Without preprocessing overhead, Geom achieves 46.7x and 9.06x of speedup for Citeseer and Reddit applications. With preprocessing overhead, Geom still achieves 37.4x and 8.66x speedup compared to GPU.

In addition, such an LSH-based technique can be extended to support on-line graph reordering for batching and sampling techniques. The LSH-clustering has the time complexity of $O(n*nz*|H|)$, where $|H|$ is the number of the hashing functions and nz is the average non-zero elements in the adjacency matrix. Supporting the on-line graph reordering will be our future work.

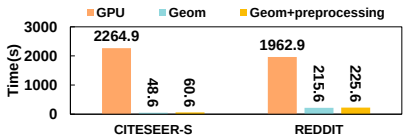

Fig. 13. Preprocessing overhead.

Batching and Sampling Influence. Batching and sampling strategies are proposed to train the graph model to alleviate the memory and computation burden for training the entire graph data in one epoch and improve the convergence speed as well [13], [21]. The state-of-art algorithm work [13] observes that the training node sets with more edges are very important for improving the convergence rate of the GCN models during sampling or batching. Our reordering methodology greatly helps to target the node sets with large dense connections, thus enables a more efficient batching and sampling method. Additionally, the reordered graph remains useful even for random batching or sampling because the order for temporal data reuse stays still in the subgraphs.

VII. Related Work

Graph acceleration In observing that the graph applications exhibit the high cache miss rates and under-utilization of memory bandwidth, abundant work have been proposed to accelerate graph analytics applications. They can be classified as the following categories: **1) Graph Preprocessing**: In order to improve the data access efficiency, it is necessary to preprocess graph data that adapts the graph structure to the hardware accelerators. For example, graph layout reorganization, graph ordering [4], and graph partitioning [9]. Our work incorporates the graph ordering techniques to improve the data reuse of non-Euclidean dataflow during GCN training. **2) Hardware acceleration**: Customized architectures have been proposed to accelerate graph applications. Previous work designs hardware modules to implement the gather, apply, scatter phases in graph computing [20], [37]. Graphicionado adopts large on-chip eDRAM for storage of the graph data to eliminate the random accesses, and another work [37] designs dedicated cache hierarchy for different graph data. However, such on-chip design cannot efficiently handle the spatial data reuse inside the NN-based computation. Additionally, the computing units in graph accelerators are too lightweight for the NN-based computation of GCN applications.

DNN Accelerators Academia and industry have proposed various architectures for general acceleration of DNNs [11], [12], [36], [39], which can be classified as the temporal architectures and spatial architectures. The spatial accelerators are based on dataflow processing, which the processing element or ALUs form a processing datapath for directly communicating with each other. Many advanced dataflow optimization strategies are proposed, such as input stationery, weight stationery, and row stationery, *etc*. Such dataflow designs eliminate the overhead of loading or storing data from and into memory hierarchy. However, the dataflow optimizations are only applicable to Euclidean data processing with regular data reuse directions or datapaths. For the irregular graph data, there is no uniform data reuse datapaths. Therefore, our work propose memory hierarchy design to support both of these two dataflows to improve the data access efficiency.

VIII. Conclusion

The graph convolutional network (GCN) is a promising approach to learn the inductive representation of graphs from many application domains. To meet the demands of this new learning method mixing the computation of graph analytics and neural network, we propose the geometric learning accelerator based on spatial architectures for graph neural network models, Geom, and enhance memory hierarchy design to support the data reuse of both the Euclidean and non-Euclidean data. We further develop a lightweight graph reordering strategy to improve the temporal reuse of non-Euclidean data and eliminate workload. Finally, we evaluate Geom accelerator design, Geom, and compare it with the existing architectural design of NN accelerator and graph accelerator on representative GCN models and datasets. Evaluation results demonstrate that Geom together with our mapping method achieves significant speedup and better energy efficiency compared with prior designs.

REFERENCES

[1] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327494

[2] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 1225–1233. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969239.2969376

[3] L. Backstrom *et al.*, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '06. ACM, 2006, pp. 44–54.

[4] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 203–214.

[5] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie, "Analysis and optimization of the memory hierarchy for graph processing workloads," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 373–386.

[6] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[7] R. v. d. Berg, T. N. Kipf, and M. Welling, "Graph convolutional matrix completion," *arXiv preprint arXiv:1706.02263*, 2017.

[8] T. D. Bui, S. Ravi, and V. Ramavajjala, "Neural graph learning: Training neural networks using graphs," in *Proceedings of the Eleventh ACM International Conference on Web Search and Data Mining*, ser. WSDM '18. New York, NY, USA: ACM, 2018, pp. 64–71. [Online]. Available: http://doi.acm.org/10.1145/3159652.3159731

[9] D. Chakrabarti, "Autopart: Parameter-free graph partitioning and outlier detection," in *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 2004, pp. 112–124.

[10] J. Chen, T. Ma, and C. Xiao, "FastGCN: Fast learning with graph convolutional networks via importance sampling." [Online]. Available: https://arxiv.org/abs/1801.10247v1

[11] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ACM Sigplan Notices*, vol. 49, no. 4. ACM, 2014, pp. 269–284.

[12] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2016.

[13] W.-L. Chiang, X. Liu, S. Si, Y. Li, S. Bengio, and C.-J. Hsieh, "Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, ser. KDD '19. New York, NY, USA: ACM, 2019, pp. 257–266. [Online]. Available: http://doi.acm.org/10.1145/3292500.3330925

[14] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, ser. SCG '04. New York, NY, USA: ACM, 2004, pp. 253–262. [Online]. Available: http://doi.acm.org/10.1145/997817.997857

[15] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proceedings of the 30th International Conference on Neural Information Processing Systems*, ser. NIPS'16. USA: Curran Associates Inc., 2016, pp. 3844–3852. [Online]. Available: http://dl.acm.org/citation.cfm?id=3157382.3157527

[16] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in neural information processing systems*, 2015, pp. 2224–2232.

[17] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch geometric." [Online]. Available: http://arxiv.org/abs/1903.02428

[18] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proceedings of the national academy of sciences*, vol. 99, no. 12, pp. 7821–7826, 2002.

[19] P. Goyal and E. Ferrara, "Graph embedding techniques, applications, and performance: A survey," *Knowledge-Based Systems*, vol. 151, pp. 78–94, 2018.

[20] 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)*. IEEE, 2016, pp. 1–13.

[21] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., pp. 1024–1034. [Online]. Available: http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs.pdf

[22] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[23] M. Henaff, J. Bruna, and Y. LeCun, "Deep convolutional networks on graph-structured data," *arXiv preprint arXiv:1506.05163*, 2015.

[24] W. Huang, T. Zhang, Y. Rong, and J. Huang, "Adaptive sampling towards fast graph representation learning," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., pp. 4558–4567. [Online]. Available: http://papers.nips.cc/paper/7707-adaptive-sampling-towards-fast-graph-representation-learning.pdf

[25] 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, D. Killebrew, 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*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/3079856.3080246

[26] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann, "Benchmark data sets for graph kernels," 2016. [Online]. Available: http://graphkernels.cs.tu-dortmund.de

[27] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks." [Online]. Available: http://arxiv.org/abs/1609.02907

[28] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Citeseer, Tech. Rep., 2009.

[29] Y. LeCun, B. L, and Y. Bengio, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, 2015.

[30] R. Levie, F. Monti, X. Bresson, and M. M. Bronstein, "Cayleynets: Graph convolutional neural networks with complex rational spectral filters," *IEEE Transactions on Signal Processing*, vol. 67, no. 1, pp. 97–109, 2018.

[31] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2009, pp. 469–480.

[32] Y. Li, W. Ouyang, B. Zhou, J. Shi, C. Zhang, and X. Wang, "Factorizable net: an efficient subgraph-based framework for scene graph generation," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 335–351.

[33] F. Monti, M. M. Bronstein, and X. Bresson, "Geometric matrix completion with recurrent multi-graph neural networks," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. USA: Curran Associates Inc., 2017, pp. 3700–3710. [Online]. Available: http://dl.acm.org/citation.cfm?id=3294996.3295127

[34] M. Niepert, M. Ahmed, and K. Kutzkov, "Learning convolutional neural networks for graphs," in *International conference on machine learning*, 2016, pp. 2014–2023.

[35] Nvidia., "Cuda toolkit documentation." [Online]. Available: http://docs.nvidia.com/cuda/profiler-users-guide/index.html

[36] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," *Microsoft Research Whitepaper*, vol. 2, no. 11, pp. 1–4, 2015.

[37] 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)*, June 2016, pp. 166–177.

[38] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. New York, NY, USA: ACM, 2014, pp. 701–710. [Online]. Available: http://doi.acm.org/10.1145/2623330.2623732

[39] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G. 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)*, June 2016, pp. 267–278.

[40] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[41] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: http://arxiv.org/abs/1409.1556

[42] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1," Technical Report HPL-2008-20, HP Labs, Tech. Rep., 2008.

[43] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" [Online]. Available: http://arxiv.org/abs/1810.00826

[44] J. Yang, J. Lu, S. Lee, D. Batra, and D. Parikh, "Graph r-cnn for scene graph generation," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 670–685.

[45] 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*, ser. KDD '18. ACM, pp. 974–983, event-place: London, United Kingdom. [Online]. Available: http://doi.acm.org/10.1145/3219819.3219890

[46] 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 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 4800–4810. [Online]. Available: http://papers.nips.cc/paper/7729-hierarchical-graph-representation-learning-with-differentiable-pooling.pdf

[47] V. Zayats and M. Ostendorf, "Conversation modeling on reddit using a graph-structured lstm," *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 121–132, 2018.

[48] J. Zhang, X. Shi, J. Xie, H. Ma, I. King, and D.-Y. Yeung, "Gaan: Gated attention networks for learning on large and spatiotemporal graphs," *arXiv preprint arXiv:1803.07294*, 2018.

[49] Z. Zhang, P. Cui, and W. Zhu, "Deep learning on graphs: A survey." [Online]. Available: http://arxiv.org/abs/1812.04202

[50] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications." [Online]. Available: http://arxiv.org/abs/1812.08434

[51] R. Zhu, K. Zhao, H. Yang, W. Lin, C. Zhou, B. Ai, Y. Li, and J. Zhou, "AliGraph: A comprehensive graph neural network platform." [Online]. Available: http://arxiv.org/abs/1902.08730