GLIST: Towards In-Storage Graph Learning

Cangyuan Li, Ying Wang, Cheng Liu, and Shengwen Liang, *SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China;* Huawei Li, *SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China; Peng Cheng Laboratory, Shenzhen, China;* Xiaowei Li, *SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; University of Chinese Academy of Sciences, Beijing, China*

https://www.usenix.org/conference/atc21/presentation/li-cangyuan

This paper is included in the Proceedings of the
2021 USENIX Annual Technical Conference.

July 14–16, 2021

978-1-939133-23-6

Open access to the Proceedings of the
2021 USENIX Annual Technical Conference
is sponsored by USENIX.

GLIST: Towards In-Storage Graph Learning

Cangyuan Li [1, 2], Ying Wang [1, 2], Cheng Liu [1, 2], Shengwen Liang [1, 2], Huawei Li [1, 2, 3], Xiaowei Li [1, 2]
SKLCA, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China [1]
University of Chinese Academy of Sciences, Beijing, China [2]
Peng Cheng Laboratory, Shenzhen, China [3]

Abstract

Graph learning is an emerging technique widely used in diverse applications such as recommender system and medicine design. Real-world graph learning applications typically operate on large attributed graphs with rich information, which do not fit in the memory. Consequently, the graph learning requests have to go across the deep I/O stack and move massive data from storage to host memory, which incurs considerable latency and power consumption. To address this problem, we developed GLIST, an efficient in-storage graph learning system, to process graph learning requests inside SSDs. It has a customized graph learning accelerator implemented in the storage and enables the storage to directly respond to the graph learning requests. Thus, GLIST greatly reduces the data movement overhead in contrast to conventional GPGPU based systems. In addition, GLIST offers a set of high-level graph learning APIs and allows developers to deploy their graph learning service conveniently. Experimental results on an FPGA-based prototype show that GLIST achieves 13.2× and 10.1× average speedup and reduces the power consumption by up to 98.7% and 98.0% respectively on a series of graph learning tasks when compared to CPU and GPU based solutions.

1 Introduction

Graph is a fundamental data structure widely seen in modern computer systems and applications. Real-world social networks, molecular graph structures, biological protein networks, social networks, and data from many other fields can be modeled as graphs, particularly the attributed graphs (AGs), which carry richer property information than well-studied plain graphs [19, 40, 42]. Attributed graphs occupy a growing proportion of storage space in the datacenters of service providers such as Facebook, Amazon and Alibaba, and the trend will continue especially with the popularity of graph database and graph analytics platforms for citation networks and recommender systems [4, 7, 43, 58]. Taobao, one of the largest online consumer-to-consumer (C2C) platforms, for example, manages attributed graphs that consist of one billion users and two billion items [43]. Therefore, as the machine learning technology advances, the question of how to make prediction, discover new patterns, and mine useful information from such rich attributed graphs, which is known as the area of Graph Learning (GL), is gradually becoming important in private and public cloud datacenters where the massive graph data can be ingested to learn the basic classification, clustering, visualization and prediction functionality [5, 29, 35, 43, 52–54].

Conventionally, common graph learning tasks require numerous CPU or GPU nodes to deal with large-scale graph learning problem and the related user queries, which directly translates to sheer growth of power and cost overhead. For instance, a typical GL-based recommender system in Alibaba [43] employs hundreds of GPUs in service to mine billion-scale attributed graph data associated to numerous customers and shopping items. To investigate more cost-effective GL systems, in this work we first characterize the real-world GL applications by building a conventional single-node GPU+SSD based graph learning system. In this system, several critical tasks found in realistic datacenter infrastructures are implemented and simulated (See Section 3). We discovered that there are several important impactive performance in these mainstream graph learning tasks. (1) For typical graph learning systems that respond to graph analysis requests as shown in Table 2, the storage-and-compute decoupled systems are bottlenecked by I/O operations, and they are not energy efficient in dealing with the GL requests due to the costly data movement from the storage to CPUs/GPUs. (2) Large-scale graph learning tasks exhibit poor data locality, which can hardly be exploited in the limited on-chip or even off-chip memory due to the large footprint of attributed graphs such as social networks or recommender systems. (3) We found that, although graph learning tasks are much more complicated than plain-graph processing, they are generally solvable by emerging graph neural networks (GNN), which means a compact specialized GL accelerator is a viable alter-

native to GPUs and CPUs in storage-centric GL systems.

To replace the power-hungry CPU/GPU based solutions and eliminate the unnecessary power consumed by graph data movement, we propose a near data computing system to realize efficient Graph Learning In-STorage (GLIST). As depicted in Figure 1(b), GLIST is a combination of in-SSD computing and customized graph learning accelerator architecture, and it enables the storage device to directly respond to attributed graph analysis requests and queries, making the data warehouse machines more energy efficient.

However, fitting large-graph learning tasks into compact storage devices remains very challenging and worth investigating. First, large graphs generally have too large footprint to fit in the DRAM memory or the caching memory of storage devices [20], thus processing a large attributed graph on request tends to have poor locality, which must be well exploited in the design of GLIST. In the graph learning process, how to efficiently and directly fetch graph learning model parameters and the graph itself from the flash devices, how to preserve locality in the working-set of GL, and how to exploit the abundant channel-level flash bandwidth in SSDs is also very important.

Second, fitting large-scale graph learning workloads into storage SSDs is challenging due to the limitation of power and computing resource inside the SSDs that generally have embedded CPU or MCU for flash device management, because deep learning technology based graph analysis workloads are bandwidth and computational intensive at the same time. This calls for a more efficient architecture to practice in-storage graph learning with SSDs.

Third, though analyzing a single graph request does not exhibit good memory locality, it is found that the inter-request locality does exist as the working-sets of temporally correlated requests overlap with one another to some degree as will be discussed in Section 3. Thus, to achieve the best efficiency of the in-storage computing, fully exploiting the concurrency and the inter-request locality in the graph analysis requests is also important. As a consequence, more consideration should be given to the working-set caching and the request scheduling strategy in the GLIST controller to reuse GL model and graph data in storage.

In all, we make the following contributions in this paper:

- We profiled real-world GL workloads in different categories and obtained two main observations for optimizing the architecture of GNN systems in terms of data locality, especially for systems with block-based storage devices.
- Based on our observations, we proposed the GLIST architecture to enable high-throughput graph learning services. We handle concurrent requests issued to the power-limited graph learning storage with specialized caching system and locality-centric request scheduling policy to exploit the data locality in and between the attributed graph analysis requests. The graph analysis requests are processed inside SSDs with a unified hardware accelerator to handle various graph learning tasks instead of going across a deep I/O stack. To the best of our knowledge, GLIST is the first in-storage acceleration system for graph learning workloads such as recommender systems and automated customer service.
- We build a GLIST prototype on the Cosmos Plus OpenSSD platform [1]. Experimental results show that the GLIST caching and scheduling policy can improve the performance by up to 13.2× and 10.1× compared to CPU+SSD, GPU+SSD based system, respectively.
- GLIST provides a software abstraction with a set of programming APIs that enable developers to create and deploy their graph learning models and analysis service into the in-storage graph learning system.

2 Background

2.1 Graph Learning Tutorial

Graph neural network applications can be modeled as an encoding-decoding method [15, 58]. The encoding function encodes the vertices in a graph into latent representation (also called embedding) that summarizes both the location and neighboring information. The decoding function decodes the embedding to the original vertex information, which is directly related to graph learning tasks, such as labeling a vertex in classification task.

Table 1: GNN Notations.

Notations	Description	Notations	Description
G	attributed graph G(V,E)	Nb(v)	vertex v's neighbor set
$\mathbf{h}_v$	the embedding vector of vertex v	N(v)	subset of vertex v's neighbor set
e(i,j)	the edge between v_i and v_j	R	analysis result

Typically, the encoder function is composed of three types of functions including **Sample**, **Aggregate**, and **Combine**. **Sample** controls the scope of the information to be processed in a graph. As formulated in Eq. 1, it samples a subset of the neighbor vertices and constructs a new sub-graph for embedding [6, 14, 53]. The notations used in the formulation is summarized in Table 1. **Sample** can also be omitted according to GCN [21] and GIN [48]. In this case, all the neighbor vertices are used for embedding calculation.

$$\mathbf{S}_v = \mathbf{Sample}^k(Nb(v)) \tag{1}$$

Aggregate aggregates the features of all the incoming vertices to update the feature of current vertex v.

$$\mathbf{h}'^k_v = \mathbf{Aggregate}(\{\mathbf{h}^{(k-1)}_u\}_{u\in N_v}) \tag{2}$$

where $\mathbf{h}^{k'}_v$ is the feature of vertex v aggregated from features of neighbor vertices $\mathbf{h}^{(k-1)}_u$ at the *(k - 1)*th layer.

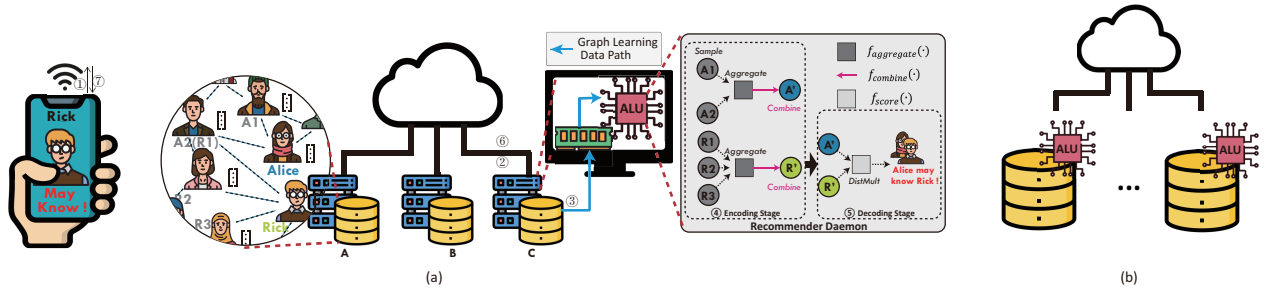

Figure 1: **A processing example of typical GL-based social network recommendation service [14, 21, 46]** When Alice tries to extend her connection via a social network App, a request is generated by the App and sent to the data center①. The request is then converted to multiple graph analysis operations to predict users who Alice may be interested in. One of the operations is assigned to Server C② to predict the potential connection between Alice and Rick. With conventional storage, the relevant sub graph of the huge social network must be loaded from the external storage to main memory and will be processed with an encoder function on host. The processing is to generate embedding vectors that can represent the two users' social network characteristics③. Then a DNN-based predictor is invoked as a decoder to determine whether the two users may agree to connect④. Finally, the recommendation is obtained based on all the prediction results and sent to the user App⑤. (b) GLIST can simplify the graph learning processing. There is no data movement between the storage and the host system.

In order to obtain the updated feature of vertex v at layer k i.e. $\mathbf{h}_v^k$, **Combine**, which is essentially an MLP operation, is applied.

$$\mathbf{h}_v^k = \mathbf{Combine}(\mathbf{h}'^k_v) \tag{3}$$

With multiple iterative processing, the obtained embedding vectors are fed to the decoder function to perform graph analysis tasks. The graph learning tasks can be categorized into three types [47]: **Node-level analysis**, **Edge-level analysis**, **Graph-level analysis**. Meanwhile, the decoder function varies from specific graph learning tasks, and they will be detailed as follows.

Node-level analysis aims to classify nodes without labels in graphs. It can also be applied for the classical community classification task in online social network analysis [3, 26], which essentially classifies nodes into several communities. The decoder function of **Node-level analysis** can be formulated as Eq. 4 [21].

$$\mathbf{R}_v = \mathbf{Decoder}(\mathbf{h}_v) \tag{4}$$

Edge-level analysis focuses on the prediction of missing edges or edges' attributes. A typical use is to predict the potential connections between users and items in recommender systems, revealing a user's interest in an item. The decoder function of **Edge-level analysis** can be formulated as Eq. 5 [38].

$$\mathbf{R}_{e(i,j)} = \mathbf{Decoder}(\mathbf{h}_i, \mathbf{h}_j) \tag{5}$$

Graph-level analysis operates on the entire graph as formulated below [48] and mainly targets for graph classification.

$$\mathbf{R}_\mathbf{G} = \mathbf{Decoder}(\{\mathbf{h}_u\}_{u \in G}) \tag{6}$$

GNN can be used for many applications as summarized in Table 2. We take a GL-based social network recommender system shown in Figure 1 (a) as an example to illustrate the use of GL. When the server receives an edge analysis request to predict the potential relationship between two users②, the algorithm will load the relevant sub-graphs of the social network in external storage with a **Sample** function to host memory, and then generates embedding vectors for the analyzed users③ with an **Aggregate** function and a **Combine** function. This will lead to high processing latency mainly caused by (1) random data access to storage and (2) massive data transfer across the bandwidth-limited PCIe bus and deep OS software stack. Then, the two generated embedding vectors are used by a DNN-based predictor to determine the existence of social connection between the two users④. Finally, the prediction result, which is usually a scalar, is sent back to the host machine and the user App eventually to make a recommendation. The proposed GLIST system, as shown in Figure 1 (b), however, performs the graph learning tasks only in SSDs to mitigate the drawbacks mentioned above.

2.2 In-storage Graph Processing

By enabling computation in storage that can avoid massive data movement between storage devices and host memory, in-storage computing (ISC) has become a promising computing paradigm for big data processing [8, 11, 17, 18, 28, 32, 37]. Graph processing on large-scale graphs is considered to be I/O intensive and requires frequent accesses to the graph in storage, so it fits well to the ISC paradigm. A number of prior works have intensively investigated the use of ISC for graph processing and demonstrated competitive performance and energy efficiency [18, 22, 23, 25, 31, 34, 41, 56]. GraphSSD [34] proposed a semantic-aware translation layer for efficient data access in graph processing. GraphOne [23] proposed an efficient dynamic graph store to facilitate both runtime graph update and processing. It supports various graph processing operations from distinct perspectives. G-Store [22] and MO-

Table 2: Graph learning tasks, algorithms, and datasets.

Analysis level	Model	Graph	#Vertices/#Graphs	#Edges(per graph)	Application
Node-Level	GCN [21]	ogbn-products (OP) [16]	2,449,029	61,859,140	Product category prediction
	GS-Pool [14]	soc-LiveJournal1 (SL) [3, 26]	4,847,571	68,993,773	On-line community classification
		twitter (TW) [24]	61,578,417	1,468,365,182	User classification in social network
		ogbn-papers100M [16]	111,059,956	1,615,685,872	Research papers classification
Edge-Level	GS-Pool [14]	ogbl-citation2 (OCi) [16]	2,927,963	30,561,187	Missing citations prediction
	PinSage [53]	ogbl-wikikg2 (OW) [16]	2,500,604	17,137,181	Knowledge graph completion
		SOC-Friendster (SF) [51]	65,608,366	1,806,067,135	Missing relationships prediction in social network
Graph-Level	GCN [21]	ogbg-molpcba (OM) [16]	437,929	28.1	Molecular property prediction
	GIN [48]	ogbg-code(OCo) [16]	452,741	124.2	Code summarization
		ogbg-ppa (OP) [16]	158,100	2,266.1	Taxonomic prediction

SAIC [31] also achieved efficient in-storage graph processing with redundant data elimination methods and locality optimizations. Graphene [30] and FlashGraph [57] were proposed to address the I/O challenge in graph processing by managing frequently accessed data in DRAM.

However, due to the power constraint, the low-end processors in storage usually have limited computing capability to deal with complicated and demanding tasks. In this case, many powerful hardware accelerators are built in the context of ISC in recent years. GraFBoost [18] develops a specialized accelerator to coalesce the random accesses to the storage in large-scale graph processing and achieves server-class performance with small memory and low power footprint. ExtraV [25] utilizes a cache-coherent hardware accelerator to achieve both high performance and high flexibility for plain graph analysis.

While prior in-storage graph processing works mainly target to analyze plain graphs which only have simple scalar attributes, they cannot fulfill the processing requirements of the graph learning workloads that mostly operate on graphs with large vector attributes, because the graph learning tasks have distinct data access patterns and computation intensity. In addition, the primitive operations used in graph learning can also be unique. For example, the **Sample** function is not supported by any of the conventional plain graph processing abstractions [33, 36, 55]. Thus, we are motivated to investigate a novel ISC architecture for cutting-edge learning tasks on large and spare graphs.

3 GL Workload Study for GLIST Design

3.1 Single Workload Characterization

Experimental setup In order to characterize and gain insight of various graph learning workloads, we conduct an in-depth study on a series of real-world representative GL applications on GPU [44]. The details of the applications and the datasets used for evaluation are illustrated in Table 2. The models and evaluation datasets are all stored in a 1 TB Samsung 970 PRO NVMe SSD. The computation device is an NVIDIA V100 GPU (Volta) equipped with 16 GB HBM2 memory.

Result analysis The latency of a graph learning task is broken down into three parts: GPU compute time(Computation)

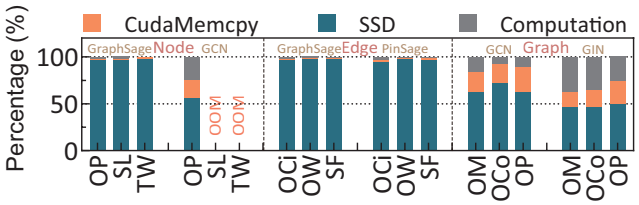

Figure 2: Performance breakdown of compute and I/O time for different graph learning workloads.

which implies the computation overhead, CudaMemcpy time(CudaMemcpy) which represents the time spent on the data movement between GPU and host, and the SSD to DRAM communication time (SSD). Figure 2 shows the profiling results, from which we can safely land two conclusions: First, the I/O bottleneck significantly penalizes the performance of graph analysis requests for most graph learning tasks. As we can see, all the graph learning tasks evaluated in this experiment spend more than half of their execution time on I/O operations, which means that GL workloads are limited by I/O bandwidth.

Second, it is hard for real-world large graphs to fit the memory. For example, twitter (TW) and soc-LiveJournal1 (SL) in the tested datasets cannot be fully loaded into 16 GB GPU memory. Thus, the algorithms have to load data from last-level storage on demand (Sampling based algorithms, i.e. GS-Pool and PinSage) or even cannot run (Non-Sampling based algorithms, i.e. GCN and GIN). Huge graphs not only increase the I/O access overhead, but also seriously restrict the throughput of general purpose processors, due to the memory capacity limitation. However, if the working set of graphs can be preprocessed where they originally stay and only the relevant sub-graph are moved, the data movement and processing overhead can be significantly reduced.

3.2 Locality in Graph Learning Workloads

To enable graph learning inside storage and service GL requests from users, we must exploit the locality in graph learning workloads to alleviate the SSD bandwidth limitation while preventing the long-latency flash accesses from penalizing the response performance. Two major exploitable locality observations that help fit the graph learning workloads into SSDs will be illustrated as follows:

1. *There exists working set reusability in between graph learning requests.*

Because SSD typically has much longer latency and coarser access granularity like pages and blocks, it is essential to take advantage of the limited DRAM or SRAM cache in it and exploit the locality in between the graph analysis requests for performance improvement. There are two potential types of locality in between the graph learning requests. The first type is Graph Data Locality (GDL) while the second is model parameter locality (MPL). For GDL, processing each vertex in the graph will involve a working set consists of its neighbors' property data. Graph analysis requests that happen to hit vertices in the proximate regions in the graph probably share a common working set. For MDL, many graph learning requests like node classification may utilize the same model parameters, so it will be beneficial to select and combine the graph learning requests with the same model parameters from all the batched requests.

2. *The layout of graph data in flash channels significantly impacts the locality of in-storage graph learning.*

Each single vertex/edge feature vector in attributed graphs is usually at the size of hundreds of bytes or few KB [19, 39, 50] and is smaller than a flash page size (i.e. 16 KB), the minimum operation granularity of flash devices. However, recent graph neural networks that respond to GL requests usually adopt **Sample** function which samples a subset of the target vertices' multi-hop neighbors [10, 14, 53]. This means that there may exist bandwidth under-utilization when the vertices located in the same flash page are not sampled simultaneously because the multi-hop structural correlation may not be captured by the **Sample** function.

For *Observation 1*, the request scheduling and caching strategy should be designed to fully exploit the temporal data locality that exists between requests. For *Observation 2*, the feature data layout in flash devices should be reorganized to improve the data reuse in a flash page.

4 GLIST Design

4.1 System Overview

To move the graph learning ability into storage devices, a state-of-the-art GL framework must support for system designers to develop and deploy the service of GL functions, e.g. GNN-based recommender systems and vertex classification, in storage devices. Inspired by the GL framework described in [58], we construct a multi-layered system architecture for the GLIST system, including user interface, run-time management, and specialized hardware as shown in Figure 3.

For the purpose of processing various graph data with GLIST, users can interact with it using the provided commands (see Table 3) via the GLIST Application Interface to define or invoke the specific GL functions in storage devices. Except the interface of defining and calling graph analysis functions in storage, GLIST also implicitly performs locality-aware graph reorganization for the newly registered and updated graphs on the host machine, so that the GLIST system can improve storage operations and the response efficiency when processing the received user analysis requests.

Take the recommender system shown in Figure 1 as an example. At the offline stage, the social network graph which embeds the users' friendship information with connected edges is registered and stored in the flash devices of the GLIST system by *GraphRegister()*. The API also quantizes the vertex feature vector and chooses appropriate bit-width for edge data representation. The registered graph will further be used to make recommendation via GNN algorithms by predicting the existence of an edge. The GNN-based recommender model, e.g. PinSage from Pinterest [53] is trained and obtained by the application developers, and is then registered and kept in storage via *ModelRegister()*. It will be later invoked on requests.

After the model deployment, the users' clicks on the relevant App are converted to GL queries and sent to the data center machines. Particularly for the friend recommendation queries, essentially they belong to typical link predictions over the social network graph and will be handled by the daemon process running on the host of the GLIST system. On receiving the requests, the daemon process calls *GraphAnalysis()*. To exploit the data reuse between requests and ensure the request processing latency at the same time, the GL requests are batched in fixed time windows before being issued to the computing storage in GLIST ①. In the computing storage, a runtime environment is maintained to manage the incoming link prediction requests②. It translates each link prediction request to primitive analysis commands including a vertex embedding command that invokes the encoder function and a prediction command that executes the decoder function. The link prediction can be obtained after the execution of the corresponding primitive analysis commands.

In addition, the GLIST runtime also provides optimizations to exploit the data reuse within the batched requests and roughly includes two parts: (1) It reorders the primitive vertex analysis commands that generate flash accesses (i.e. vertex embedding requests) to explore the graph data reuse and fits the flash accesses to the flash channel-level parallelism. (2) It groups the reordered primitive vertex analysis commands into small batches to increase the bandwidth utilization of ways and channels, instead of sequentially handling each graph analysis command with limited footprint [14, 53]. After the commands are received and handled by the GLIST runtime, they are further decoded and sent as instructions to the GLIST processor that eventually executes and accelerates the graph learning functions. The instructions are served by the Sampler first, which fetches the feature vectors from the flash devices ⑦-⑧ or the Page Cache directly ⑨ from the on-board DRAM. Then, it constructs a larger sub-graph by merging multiple small sub-graphs obtained from the grouped

Table 3: GLIST APIs

Category	APIs
Graph Update	AddEdge, RemoveEdge, AddVertex, RemoveVertex, UpdateVertex
Graph Registeration	GraphRegister, Graph Unregister
Model Registeration	ModelRegister, ModelUnregister
Graph Analysis	GraphAnalysis, GetAnalysisResult

sampling functions [6, 14, 43]. The newly assembled graph is further loaded to the on-chip buffers of GLA. When all the required data are ready, the processing element array is instructed by the commands to execute the invoked vertex analysis model. Afterwards, when the feature vectors of the queried edge's endpoints are ready, the primitive prediction function is scheduled onto the GLA to predict the link probability between target vertices. Finally, the GLA triggers the GLIST Runtime to collect the results and return the analysis results to the daemon process via *GetAnalysisResult()*.

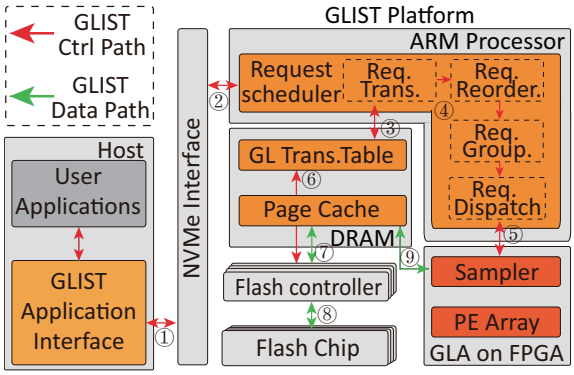

Figure 3: System architecture of GLIST.

4.2 The GLIST Runtime

The GLIST runtime is designed to decode, schedule, and issue the input requests to flash devices and the GLA. It manages the incoming requests as commands from the host machine, and also exploits the locality in between concurrent requests, and re-schedule the requests to maximize the available locality. To improve the flash bandwidth utility and exploit the inter-request locality, the GLIST runtime maintains two key structures, the Page Cache and the Graph Learning Translation Table, which enable the reuse of graph data and GNN models fetched from the flash devices in and between consecutive requests.

GLIST Page Cache is adopted to exploit temporal data locality between user requests. It caches the edges, vertex feature vectors and model data touched by the previously executed requests. Besides, the intermediate data such as the embedding vectors of vertices are also cached. We adopted the Least Recent Use strategy as replacement policy. The Page Cache works in the process of request response, and it is also used to hide the latency of operations correlated to the GNN function deployment stage, e.g. registering new function models.

Graph Learning Translation Layer, denoted as GL-TL, is provided to index reusable objects in SSDs including the graph property data, edge information, and analysis model parameters. GL-TL replaces the conventional LBA-to-PPN (logic block address to physical page number) paging used in commercial SSDs. GL-TL includes three translation tables, i.e. the Vertex Mapping Table, the Property Mapping Table, and the Model Mapping Table. The Vertex Mapping Table records the mapping between the vertex ID and the flash page which keeps its neighbors. Besides, it also records other metadata of each vertex such as the number of adjacent vertices. Similarly, the Model Mapping Table and the Property Mapping Table keep the logical object index and physical block address. All the tables are kept in the DRAM when GLIST is activated.

4.2.1 In-storage Graph Learning Request Scheduling

Though the GLIST Page Cache and GL-TL enable the reuse of graph in between requests, how to group the requests into batch of concurrently executed commands impacts the efficacy of locality enhancement. When being requested, GNNs usually at first sample the large graph and operate on certain sub-graphs. Due to the random sampling strategies, analyzing a single vertex usually touches several flash pages to fetch the feature vectors of the sampled sub-graph, which has very unpredictable locality and sometimes causes a huge waste of flash bandwidth. However, if multiple analysis requests are concurrently processed and tactically reordered by the GLIST runtime, the flash bandwidth utilization will be improved. Nevertheless, due to the limited size, the DRAM in storage cannot accommodate the whole working set of a large request batch. Reordering and grouping the requests will help improve the cache reusability. In this way, multiple groups are served sequentially to reuse the shared data including the attribute information and intermediate data of vertices, because the groups of different requests may overlap with one another and share the intermediate or the input property data in the requests. Moreover, the requests in each group are fused and processed as a batch can better utilize both the flash bandwidth and the PE array of GLA.

The process of GL requests scheduling is shown in Algorithm 1. To exploit the intermediate data reuse, GLIST leverages an encoding-decoding manner by splitting the GNN workflow into vertex embedding phase and prediction phase. In vertex embedding phase, the intermediate data can be reused by other analysis requests. For example, as shown in Figure 1, the latent representation of each user obtained from this phase can be used to generate any recommendations related to that user. The operations in the prediction phase, however, are highly dependent on each specific user request, which hardly share intermediate data. Therefore, GLIST parses the requests (Line 3), so that only the primitive vertex embedding requests are re-scheduled. After re-scheduling, the primitive

Algorithm 1: Request Scheduling

Input: Graph G, Request Ri, Group Size S
Output: Scheduled Requests Ro, Embedding-Prediction Mapping Table EP_MT

```
1  req_mapping_table = dict()
2  par_ri = usr_req_partition(Ri)
3  for user_req ← par_ri do
4      if user_req.type == "Edge" then
5          primitive_req_mapping_table =
             extract_edge(primitive_req_mapping_table, user_req)
6      else if user_req.type == "Graph" then
7          primitive_req_mapping_table =
             extract_graph(primitive_req_mapping_table, user_req)
8      else
9          primitive_req_mapping_table =
             extract_vertex(primitive_req_mapping_table,user_req)
10 reordered_primitive_req, primitive_req_mapping_table =
     reorder_primitive_req(task_primitive_req, primitive_req_mapping_table)
11 Ro, EP_MT = request_grouping(Config, reordered_primitive_req,
     primitive_req_mapping_table)
12 return Ro,EP_MT
```

requests are reordered, and an embedding-prediction mapping table (EP_MT) will be used to record the mapping information between the re-ordered embedding phase and the intact prediction phase, so that GLIST can correctly execute the prediction phase.

Requests decomposition and reordering. The sub-graphs associated to the GL requests may overlap with each other to different extent, and contribute to different degree of locality. Thus, to maximize the temporal data locality in-between embedding requests that hit the same graph, the runtime scheduler reorders the primitive requests according to the affinity of their sampled regions in the graph. Though there are different categories of reusable data worth exploiting, the scheduler prioritizes the reuse of intermediate embedding data over that of input property data. For example, the requests on graph edge analysis, all begins with the embedding of the endpoints that can be reused while the latter prediction can be done independently. Thus, the Edge-level analysis request is decomposed by the scheduler into two primitive **Node-level Analysis** requests and one prediction request. In this way, the requests in embedding phase can be scheduled to maximize data locality and the prediction phase of different requests can reuse the embedding vectors of vertices in the DRAM cache. As shown in Algorithm 1, the batch of requests is initially separated into the primitive requests on vertices according to the requested graphs and the type of requests (Line 2). Then, the scheduler scans through the requests and reorders the primitive requests that may hit different sub-graphs (Line 4 — Line 9). Because each primitive request is correlated to a vertex as the analysis target, estimating the locality in between requests is to measure the size of overlapping area of the corresponding vertex sub-graphs, which includes all the vertices that are n-hops away from the target vertex. Thereby, in scheduling, the *reorder_primitive_req*() function is used to obtain the vertex whose sub-graph share more vertices with the previous scheduled requests than others. In practice, we

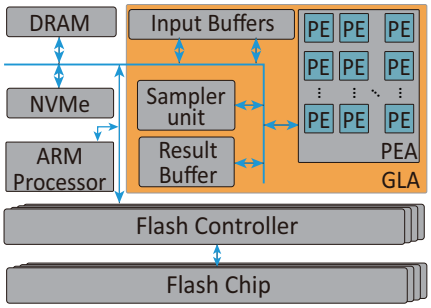

Figure 4: The GLA architecture and its integration to the hardware system.

implement the function by simply finding the vertex that has the minimum distance from the previously requested vertex.

Requests grouping. As mentioned above, the sub-graph obtained by the Sample function for each requested vertex usually causes random but small-footprint memory access, which tends to cause low flash bandwidth. In addition, for most of the GNN models, the bottom layers are witnessed to contribute less computation overhead than the top layers [6,14, 43]. As a result, a single request hitting a vertex may not fully utilize the Processing Element (PE) resources of the GLA especially in the last layer of the GNN models. Therefore, GLIST batches all the reordered requests obtained from the request reordering stage into several groups as described in Line 11 of Algorithm 1. By fusing multiple requests of the same tasks into a batched task, both the utilization of flash bandwidth and the GLA are improved.

4.3 In-storage Graph Learning Accelerator

4.3.1 The Accelerator Architecture

Based on the design presented in [27], the Graph Learning Accelerator presented in Figure 4 is composed of a graph Sampler, on-chip buffers, and a Processing Element Array (PEA) to perform graph neural network inference.

The Sampler unit. For attributed graph analysis, the Sampler unit samples the vertices and edges from a large graph according to the predefined manner, before invoking GNN inference. It supports uniform distribution sampling or other predefined sampling functions [6, 14, 53]. In the sampling stage, the property data and their connection of the sampled vertices under request are loaded from flash devices to the DRAM of the GLIST embedded platform and further to the corresponding on-chip buffers. Besides, the Sampler in GLIST also supports non-sampling based GNN models [21, 49] by loading tiled graph sequentially according to the predefined tiling configuration.

According to the GNN framework introduced in Figure 2.1, the **Combine** and **Aggregate** functions should also be supported by the GLA. An Processing Element Array (PEA) is designed to address the matrix operations in **Combine** function. Each column of the PEA handles a single dimension

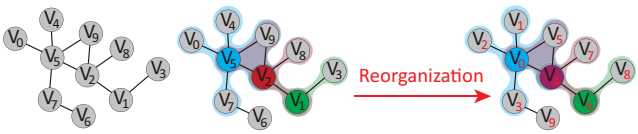

Figure 5: Graph reorganization.

of the input property for all the input vertices while the PEs in the same row are dedicated to one single vertex, so that the PEA structure is independent to the dimension of graph vertex properties.

To support the **Aggregate** function, we adopted a full mesh topology in our design by fully connecting columns of PEs in the array to achieve high-throughput message passing. Each PE in the same column broadcasts its data to all other columns and select data from other PEs' output according to the control signal generated by the controller in PEA.

Algorithm 2: Graph Reorganization

```
   Input: Graph G, Hop count h, Degree threshold Td, Center vertex number
          threshold Cn
   Output: Reorganized Graph
1  important_vertices = Top Cn nodes whose in degree < Td of G
2  for i → important_vertice do
3  |   workingset[i] = Sample from workingset[i]
4  end
5  vertex_sequence = []
6  i = 0
7  while workingset is non-empty do
8  |   Add vertex j that has the maximum intersection with vertex i in
   |     workingset[] to vertex_sequence
9  |   Remove workingset[i] from workingset
10 |   i = j
11 end
12 map_table = map()
13 for i → vertex_sequence do
14 |   Assign new ID to vertex i and its h hop neighbors
15 |   Record the mapping information in map_table
16 end
17 new_graph = Construct new graph with G and map_table
18 return new_graph
```

4.3.2 Graph Reorganization

When analyzing a vertex in a Sampling based graph learning workflow, its closer neighbors are more likely to be accessed, which shows the existence of spatial locality in GL workloads. However, the property of the vertices usually takes hundreds of bytes or few KB, which is much smaller than a flash page size and may cause flash bandwidth under-utilization. Therefore, we designed a heuristic algorithm to re-index the vertex IDs in a graph to maximize the spatial locality of GL requests as Algorithm 2 shows. Firstly, the reorganization algorithm selects the top Cn highest in-degree vertices with in-degree below the threshold Td as important vertices, where the threshold Td is used to exclude excessively high degree vertices since their neighborhood footprint often outsize the flash page and their locality can hardly be exploited. After that, it fetches each important vertex's h hop neighbors as its working set. To reduce the complexity, the algorithm usually randomly samples a subset of the true working set (i.e. $sqrt(N)$ from N vertices in our implementation) to represent the whole set. Then the important vertices are sorted according to the size of overlapping working set with others so that the potential spatial locality associated to the vertices are kept in the vertex sequence. Finally, the chosen important vertices and their corresponding h hop neighbors are assigned new IDs in sequence.

A tiny example shown in Figure 5 illustrates the graph reorganization procedure with given parameters: $h = 1, Cn = 3, Td = 0$. The procedure chooses three important vertices: $V5$, $V2$, and $V1$ according to the number of adjacent vertices and their one-hop neighbors are recorded as working-set respectively, as the shades shown in Figure 5. Then the algorithm sorts the three important vertices according to the size of overlapping working-set and obtains the sequence: $V5 \rightarrow V2 \rightarrow V1$ ($V5$'s working set has three common vertices with $V2$'s, and $V2$'s working set has two common vertices with $V1$'s). After that, each important vertex and the corresponding working set are assigned new IDs in the previously sorted order. Specifically, $V5$ and its five one-hop neighbors $V0, V4, V2, V7$, and $V9$ are assigned new IDs: $V0 \sim V5$. Then $V2$ and $V1$ follows. Finally, the procedure finds the remaining vertex that does not belong to any working set ($V6$) and assigns new ID to it to make sure that all the vertices in the graph are re-indexed.

Table 4: FPGA Resource Uasge

Module	LUT	FF	BRAM	DSP
Flash Controller	44141	30156	80	0
NVMe Interface	8586	11455	28	0
GLA Accelerator	66287	51527	172	514
In Total	136506	117261	293	514
Percent(%)	62.45	26.82	53.76	57.11

5 Evaluation

5.1 GLIST Overall Evaluation

Experiment Setup. The Cosmos Plus OpenSSD platform was employed for the proposed GLIST system implementation, and it consists of an XC7Z045 FPGA chip (ARM-FPGA), 1 GB DRAM, an 8-channel NAND flash interface, an Ethernet interface, and a PCIe Gen2 8-lane interface. We implemented the GLA with Chisel [2] and integrated it in the hybrid ARM-FPGA processor as the major GL processing engine. The hardware project was synthesized and implemented with Vivado 2016.2 and the design works at 150MHz. Table 4 shows the logic resource usage of our hardware project. The firmware of the prototype runs on Dual 1GHz ARM Cortex-A9 core of XC7Z045. The board was connected with the host server via a PCIe link. We also profiled the prototype system and built a simulator for scalable evaluation.

We take a set of **Node-level**, **Edge-level**, and **Graph-level** GL workloads shown in Table 2 as benchmarks. The models used for benchmark are all quantized to 8bit fixed point. We

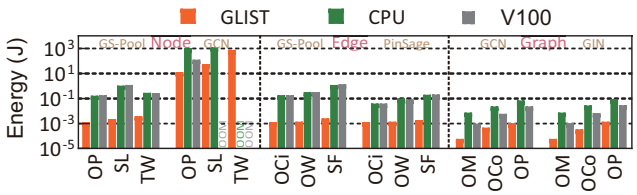

Figure 6: Energy consumption of a single request on different GL systems.

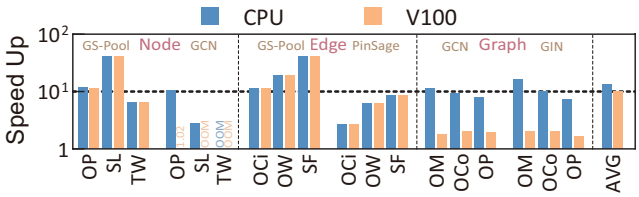

Figure 7: Single node performance of GLIST

have the benchmark implemented with GLIST on Cosmos Plus OpenSSD platform to gain insight into the advantages of the in-storage graph learning. We compared GLIST with DGL [45] on a CPU-based platform and a GPU-based platform respectively. The CPU-based platform is equipped with two Intel Xeon E5-2690 V3 processors and 64 GB DRAM. The GPU-based platform includes two Intel Xeon E5-2690 V3 processors and an NVIDIA V100 GPU. Both platforms have all the graphs and GNN models initially stored in a Samsung 970 EVO 1 TB SSD with 3.5 GB/s peak read bandwidth because the large graphs used in many graph learning applications can exceed the capacity of the main memory. To evaluate the different systems, we randomly generated 10,000 graph learning requests over the graph and measured the average processing latency and energy consumption.

Performance. The performance of the proposed GLIST-based GL system is illustrated in Figure 7. It shows 13.2× and 10.1× average speedup compared to the CPU baseline and GPU baseline, respectively. Particularly, GLIST shows significant higher performance speedup on GS-Pool and PinSage which need to sample over the large input graphs. The main reason is that the random sampling over large input graphs incurs substantial random accesses to the flash and rather low flash bandwidth utilization when GS-Pool and PinSage are deployed on the CPU platform and the GPU platform. We also measured the flash bandwidth, and it shows only 100 MB/s, which is much lower than the peak bandwidth of the device and dramatically bottlenecks the computing capability of CPUs and GPUs accordingly. As a result, the performance of the CPU platform and the GPU platform is also similar. In contrast, GLIST with intensive data layout optimization and intra-request reuse optimization greatly improves the data reuse and reduces the random accesses over the flash. Thereby, it benefits most on GS-Pool and PinSage. Different from GS-Pool and PinSage, GCN and GIN operate on the entire graph instead of sampling sub-graphs. In this case, the graph will be accessed sequentially and the flash bandwidth can be fully utilized. With sufficient data supply from the flash, the GPU platform with more parallel processing engines shows much higher performance over the CPU platform according to the experiment. GLIST takes advantage of the specialized accelerator and still outperforms the CPU platform and the GPU platform given the same flash bandwidth provision.

Energy Consumption. In this experiment, we utilized a power meter to measure the power consumption of the proposed GLIST system, the CPU-based, and the GPU-based graph learning systems respectively. Then, we obtained the energy consumption by calculating the production of the average power got by power meter and the benchmark execution time. The resulting energy consumption of the different benchmark GNN models are illustrated in Figure 6 and the per-request average power and benchmark time of different settings are listed in Table 5. It shows that GLIST reduces the average energy consumption by 98.7% and 98.0% respectively when compared to the CPU-based platform and the GPU-based platform. The significant energy reduction can be attributed to both the lower power consumption brought by the dedicated GLA in GLIST and the much lower execution time of GLIST as discussed in prior subsection. At the same time, we also noticed that the GPU-based platform shows higher energy consumption on PinSage and GS-Pool over the CPU-based platform. This is mainly caused by the fact that GPU fails to exploit its massive parallel processing engines due to the massive random access induced flash bandwidth bottleneck and much higher power consumption over the CPU-based platform. When the flash bandwidth utilization is improved for GCN and GIN that include more sequential data accesses, the execution time dominates the energy consumption in the GPU-based platform. Hence, the GPU-based platform exhibits lower energy consumption in these cases.

5.2 The GLIST Optimizations

Experimental Setup. To gain insight into the advantages of the GLIST optimization including graph reorganization (**R**), request scheduling (**S**), request grouping (**G**), and caching (**C**), we conducted a request generation server to continuously issue different graph analysis requests to the GLIST system for evaluating the above optimization strategies. In order to make the distribution of requests issued by generation servers closer to the real production system, we first analyze the real-world request trace from the commercial data center. The analysis results indicate that the requests have different levels of locality depending on the services provided by the data center and the data types accommodated in the warehouse nodes. Thereby, for simplicity, we introduce the *N%-Locality*, denoted as *N-L*, to describe the degree of locality in between the batch of requests that arrives in a fixed time window sent to GLIST. This term represents the N% neighbor vertices and edges can be reused between any adjacent request on average,

Table 5: Per-request average power and benchmark time of different platforms.

	Node						Edge							Graph				
Dataset	OP	SL	TW	OP	SL	TW	OCi	OW	SF	OCi	OW	SF	OM	OCo	OP	OM	OCo	OP
Model	GS-Pool			GCN			GS-Pool			PinSage				GCN			GIN	
P_{GLIST}(W)	25	25	25	26	25	26	25	24	25	26	26	24	25	25	26	25	26	25
T_{GLIST}(ms)	0.05	0.09	0.16	497.25	2252.24	30060.62	0.05	0.06	0.11	0.05	0.05	0.08	23.77	122.09	408.09	22.65	132.94	570.97
P_{CPU}(W)	280	281	280	201	200	-	311	282	280	292	202	288	282	202	219	202	202	219
T_{CPU}(ms)	0.60	3.80	1.03	5159.58	6191.28	-	0.58	1.17	4.37	0.13	0.33	0.68	0.027	0.11	0.32	0.037	0.13	0.42
P_{GPU}(W)	316	316	266	250	-	-	316	271	312	296	256	301	225	247	298	242	258	304
T_{GPU}(ms)	0.58	3.79	1.03	506.54	-	-	0.58	1.17	4.37	0.13	0.33	0.68	0.0043	0.024	0.079	0.0045	0.027	0.096

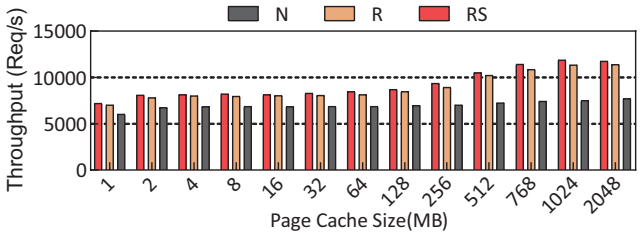

Figure 8: Throughput of GLIST w.r.t. page cache size.

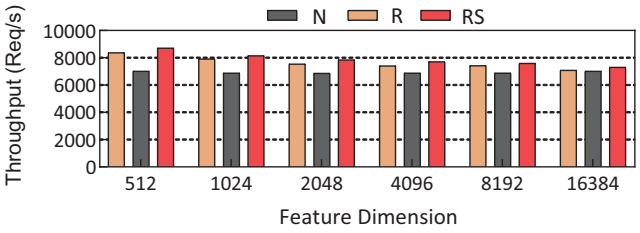

Figure 9: Throughput of GLIST w.r.t. property dimension.

which is defined as the follows:

$$N\text{-}L = \frac{N_{Common\ vertices\ of\ subgraph\ related\ to\ the\ two\ requests}}{N_{Vertices\ in\ subgraph\ related\ to\ the\ latter\ request}} \quad (7)$$

We randomly generated 10,000 vertex classification requests on the ogbn-papers100M dataset, whose property data is over 50 GB in size. We shuffled the requests to evaluate the advantages of the above optimization approaches. First, as discussed in Section 4.2, the benefits brought by the graph reorganization (**R**) and request scheduling (**S**) optimization methods are impacted by the cache size and the feature dimension. Thereby, in order to evaluate the influence of these factors on system performance, we evaluate the throughput of the GLIST system under different Page Cache sizes. After that, we adjusted the dimension of the vertex property data to distinguish the gains brought by the graph reorganization and request scheduling optimizations. Second, due to the locality level that can influence the performance of request scheduling, we fixed the property size of the vertices to 16 KB and adjust the locality level of the generated requests to show the variation of the gain of the request scheduling. Third, we explored the performance variation of the GLIST system under the different group size configurations of the GLIST runtime. Finally, we fixed the Page Cache size and measured the performance speedup of the GLIST system under different combinations of optimization methods compared to the GLIST system without optimization methods.

Evaluation. Figure 8 shows the throughput of the GLIST system under the three configurations including GLIST without optimization (**N**), GLIST with graph reorganization (**R**), and GLIST with graph reorganization and request scheduling (**RS**). It can be observed that the throughput of the system with **RS** and **R** increases as the Page Cache size increases because larger cache size can avoid vertex access being dispatched to flash memory and reduces access latency. Meanwhile, the GLIST system that adopts reorganization methods **R** possesses higher spatial data locality, which highly exploits the data reusability in each flash pages and alleviates the penalty of Page Cache misses, thereby, it performs better than the system without any optimizations, though the Page Cache is large. Furthermore, when both the request scheduling and graph reorganization are adopted in **RS** configuration, the spatial data locality of each single request can be exploited and released. In this case, as shown in Figure 8, the GLIST system with **RS** methods gains the highest throughput compared to **N** and **R** settings.

Figure 9 illustrates the system throughput of the GLIST system under the three configurations mentioned in the above paragraph with respect to various configurations on the property dimension. It can be observed that the GLIST with **R** and **RS** optimization methods achieve 19.5% and 24.2% higher throughput compared to the system without optimization **N**, respectively. As the feature dimension increases, a flash page can only accommodate a few property vectors, which results in the graph reorganization methods fails to exploit spatial data locality. Thereby, the system performance with **R** and **RS** optimization methods drops sharply and the **RS** still outperforms **R** because of the gain brought by request scheduling method. In addition, as the dimension of property vector increases, the performance gap between the **R** and **N** optimization methods gradually disappears until the property dimension arrives at the size of a flash page (16 KB). In this case, the system with request scheduling still maintains 4% higher throughput than the other settings.

Figure 10 (a) shows the speedup of the GLIST system with request scheduling methods **RS** under different locality level configuration compared to a system without optimization methods. The increase of locality level indicates the rise of data reusability, thus the GLIST system adopting request scheduling method achieves performance improvement up to 2.65×. As shown in Figure 10 (b), the GLIST system achieves performance improvement compared to the system without

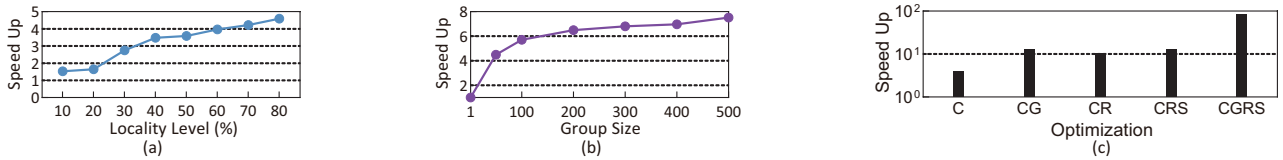

Figure 10: (a) Speedup of request scheduling w.r.t. locality level. (b) Performance improvement of different group size in request grouping. (c) Performance improvement of different optimization enabled.

request grouping optimization with the increase of group size. The reason can be attributed to two folders: (1) larger group size can fully utilize the internal bandwidth provided by multiple flash channels; (2) large group size can fully exploit the data parallelism and thus making the computation unit of the graph learning accelerator in high utilization. In addition, the limitation of internal flash bandwidth makes performance improvement slow down when the group size is larger than 200. Figure 10 (c) shows the performance speedup of the GLIST system under the combination of various optimization (**R**, **S**, **G**, and **C**) compared to the system without optimization (**N**). It can be observed that neither the combination of node embedding caching and request grouping (12.94×), denoted as **CG**, nor the combination of node embedding caching, graph reorganization, and request scheduling (12.89×), denoted as **CRS** achieves the best performance. This is because **CG** fails to exploit the data locality brought the graph reorganization and request scheduling methods under the high utilization of flash bandwidth and graph learning accelerator, which results in the Page Cache hit rate only reaches to 67.86%. Meanwhile, although **CRS** can exploit the data locality using graph organization and make Page Cache hit rate reach to 94.73%. **CRS** is unable to fully utilize the flash bandwidth and PE-Array in GLA. Not only does the **CGRS** optimization method exploit the data locality but also fully utilize the available resource, making the GLIST system achieve the highest performance improvement.

5.3 Bit-width Scalability Exploration

The bit-width of vertex feature vector and GNN model parameters has a great impact on performance, resource overhead, and energy consumption of the GLIST prototype. To explore an appropriate setting for quantization, we choose GCN [21] and Cora [39] as target model and dataset respectively to evaluate how the four configurations including **floating-point**, **32 bit fixed-point**, **16 bit fixed point**, and **8 bit fixed-point** impact on accuracy, latency, energy consumption, and resource usage. We leverage a static quantization method which enumerates every possible configuration at each layer and choose the best one with the lowest loss. The results are as shown in Table 6. Though **floating-point** and **32 bit fixed-point** achieve higher accuracy, the logic resource usage is extremely high and can hardly be implemented on current FPGA platform. And wider word size makes it hard to enable high-throughput graph learning services because of high bandwidth

Table 6: Comparison of accuracy and resource utilization

Config.	Acc.	LUT	FF	DSP	Latency	Energy
float-32	79.1%	1891986	192372	1	-	-
fixed-32	77.8%	632777	242506	889	-	-
fixed-16	77.5%	125253	112730	513	2.74ms	5.7×10^{-2}J
fixed-8	77.1%	66287	51527	514	1.80ms	3.6×10^{-2}J

requirements. For the lower bit configurations, the latency decreases by 34.3% and the energy consumption decreases by 36.8% with only a 0.4% loss on accuracy when changing bit width from 16 to 8. Moreover, the resource usage of 8-bit GLA is significantly less than another one, making it possible to implement more GLA cores in the GLIST system to perform higher throughput graph learning services. Note that the quantization method used for evaluation is only a naive one and the accuracy of low-bit configurations will show better results when changing to state-of-the-art methods [9, 12, 13].

6 Conclusion

In this paper, we formulated that the conventional GPU+SSD graph learning platforms are limited by I/O operations after studying a diverse set of graph learning tasks. We then study the data locality that exists in flash-based graph learning applications. To tackle the bottlenecks of conventional graph learning systems, we proposed an in-storage graph learning accelerating system, GLIST, which features multiple optimizations proposed based on our observations to fully exploit data locality. Finally, we implemented a GLIST prototype with FPGA and showed it achieves 13.2× and 10.1× average speedup and reduces the power consumption by 98.7%, 98.0% compared to conventional CPU and GPU based graph learning systems, respectively.

7 Acknowledgement

We are grateful to Professor Zsolt István for his useful comments and suggestions on improving this paper. This work is supported in part by the National Key Research and Development Program of China under grant 2018AAA0102705, and in part by the National Natural Science Foundation of China (NSFC) under grant No.(62090024, 61876173, 61902375). The corresponding authors are Ying Wang and Cheng Liu.

References

[1] The OpenSSD Project. http://openssd.io.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: constructing hardware in a scala embedded language. In *DAC Design Automation Conference 2012*, pages 1212–1221. IEEE, 2012.

[3] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 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*, pages 44–54, 2006.

[4] Bradley R Bebee, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. Amazon neptune: Graph data management in the cloud. In *International Semantic Web Conference (P&D/Industry/BlueSky)*, 2018.

[5] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. A taxonomy and survey of dynamic graph visualization. In *Computer Graphics Forum*, volume 36, pages 133–159. Wiley Online Library, 2017.

[6] Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[7] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[8] Chanwoo Chung, Jinhyung Koo, Junsu Im, and Sungjin Lee. Lightstore: Software-defined network-attached key-value drives. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 939–953, 2019.

[9] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. *arXiv preprint arXiv:1511.00363*, 2015.

[10] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[11] Boncheol Gu, Andre S Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, et al. Biscuit: A framework for near-data processing of big data workloads. *ACM SIGARCH Computer Architecture News*, 44(3):153–165, 2016.

[12] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented approximation of convolutional neural networks. *arXiv preprint arXiv:1604.03168*, 2016.

[13] Philipp Gysel, Jon Pimentel, Mohammad Motamedi, and Soheil Ghiasi. Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks. *IEEE transactions on neural networks and learning systems*, 29(11):5784–5789, 2018.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[15] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *arXiv preprint arXiv:2005.00687*, 2020.

[17] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[18] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, et al. Grafboost: Using accelerated flash storage for external graph analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 411–424. IEEE, 2018.

[19] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2016. http://graphkernels.cs.tu-dortmund.de.

[20] Minsub Kim and Sungjin Lee. Reducing tail latency of dnn-based recommender systems using in-storage processing. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 90–97, 2020.

[21] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[22] Pradeep Kumar and H Howie Huang. G-store: high-performance graph store for trillion-edge processing. In

SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 830–841. IEEE, 2016.

[23] Pradeep Kumar and H. Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 249–263, Boston, MA, February 2019. USENIX Association.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[25] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, H Peter Hofstee, Gi-Joon Nam, Mark R Nutter, and Damir Jamsek. Extrav: boosting graph processing near storage with a coherent accelerator. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.

[26] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[27] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. Engn: A high-throughput and energy-efficient accelerator for large graph neural networks. *IEEE Transactions on Computers*, 2020.

[28] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, Renton, WA, July 2019. USENIX Association.

[29] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[30] Hang Liu and H. Howie Huang. Graphene: Fine-grained IO management for graph computing. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 285–300, Santa Clara, CA, February 2017. USENIX Association.

[31] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a trillion-edge graph on a single machine. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 527–543, 2017.

[32] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. Deepstore: In-storage acceleration for intelligent queries. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 224–238, 2019.

[33] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[34] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. Graphssd: graph semantics aware ssd. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 116–128, 2019.

[35] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic web*, 8(3):489–508, 2017.

[36] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488, 2013.

[37] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, Renton, WA, July 2019. USENIX Association.

[38] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018.

[39] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[40] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.

[41] Linghao Song, Youwei Zhuo, Xuehai Qian, Hai Li, and Yiran Chen. Graphr: Accelerating graph processing using reram. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 531–543. IEEE, 2018.

[42] Aravind Subramanian, Pablo Tamayo, Vamsi K Mootha, Sayan Mukherjee, Benjamin L Ebert, Michael A Gillette, Amanda Paulovich, Scott L Pomeroy, Todd R Golub, Eric S Lander, et al. Gene set enrichment analysis: a knowledge-based approach for interpreting genome-wide expression profiles. *Proceedings of the National Academy of Sciences*, 102(43):15545–15550, 2005.

[43] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. Billion-scale commodity embedding for e-commerce recommendation in alibaba. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 839–848, 2018.

[44] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, et al. Deep graph library: Towards efficient and scalable deep learning on graphs. *arXiv preprint arXiv:1909.01315*, 2019.

[45] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[46] Quan Wang, Zhendong Mao, Bin Wang, and Li Guo. Knowledge graph embedding: A survey of approaches and applications. *IEEE Transactions on Knowledge and Data Engineering*, 29(12):2724–2743, 2017.

[47] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S Yu. A comprehensive survey on graph neural networks. *arXiv preprint arXiv:1901.00596*, 2019.

[48] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[49] Mingyu Yan, Zhaodong Chen, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. Characterizing and understanding gcns on gpu. *IEEE Computer Architecture Letters*, 2020.

[50] Pinar Yanardag and SVN Vishwanathan. Deep graph kernels. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1365–1374, 2015.

[51] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *Knowledge and Information Systems*, 42(1):181–213, 2015.

[52] Hao Yin, Austin R Benson, Jure Leskovec, and David F Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 555–564, 2017.

[53] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[54] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[55] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph dsl. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–30, 2018.

[56] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 45–58, 2015.

[57] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 45–58, Santa Clara, CA, February 2015. USENIX Association.

[58] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019.