Hyperscale FPGA-as-a-Service Architecture for Large-Scale Distributed Graph Neural Network

Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan
Heng Liu, Linyong Huang, Zhaoyang Du, Fei Xue, Yuanwei Fang
Hongzhong Zheng, and Yuan Xie *
{shuangchen.li,dimin.niu,hongzhong.zheng,y.xie}@alibaba-inc.com
Computing Technology Lab, DAMO Academy
Alibaba Group

ABSTRACT

Graph neural network (GNN) is a promising emerging application for link prediction, recommendation, etc. Existing hardware innovation is limited to single-machine GNN (SM-GNN), however, the enterprises usually adopt huge graph with large-scale distributed GNN (LSD-GNN) that has to be carried out with distributed in-memory storage. The LSD-GNN is very different from SM-GNN in terms of system architecture demand, workflow and operators, and hence characterizations.

In this paper, we first quantitively characterize the LSD-GNN with industrial-grade framework and application, summarize that its challenges lie in graph sampling, including distributed graph access, long latency, and underutilized communication and memory bandwidth. These challenges are missing from previous SM-GNN targeted researches. We then propose a customized hardware architecture to solve the challenges, including a fully pipelined access engine architecture for graph access and sampling, a low-latency and bandwidth-efficient customized memory-over-fabric hardware, and a RISC-V centric control system providing good programmability. We implement the proposed architecture with full software support in a 4-card FPGA heterogeneous proof-of-concept (PoC) system. Based on the measurement result from the FPGA PoC, we demonstrate a single FPGA can provide up to 894 vCPU's sampling capability. With the goal of being profitable, programmable, and scalable, we further integrate the architecture to FPGA cloud (FaaS) at hyperscale, along with the industrial software framework. We explicitly explore eight FaaS architectures that carry out the proposed accelerator hardware. We finally conclude that off-the-shelf *FaaS.base* can already provide 2.47× performance per dollar improvement with our hardware. With architecture optimizations, *FaaS.comm-opt* with customized FPGA fabrics pushes the benefit to 7.78×, and *FaaS.mem-opt* with FPGA local DRAM and high-speed links to GPU further unleash the benefit to 12.58×.

*This work was done when Yuhao Wang, Wei Han, and Heng Liu were affiliated with Alibaba.

ISCA '22, June 18–22, 2022, New York, NY, USA

ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527439

CCS CONCEPTS

• **Computer systems organization** → **Neural networks**; **Heterogeneous (hybrid) systems**; **Cloud computing**.

KEYWORDS

Graph Neural Network, FPGA-as-a-service, Accelerator

ACM Reference Format:
Shuangchen Li, Dimin Niu, Yuhao Wang, Wei Han, Zhe Zhang, Tianchan Guan, Yijin Guan, Heng Liu, Linyong Huang, Zhaoyang Du, Fei Xue, Yuanwei Fang, Hongzhong Zheng, and Yuan Xie . 2022. Hyperscale FPGA-as-a-Service Architecture for Large-Scale Distributed Graph Neural Network . In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3470496.3527439

1 INTRODUCTION

Graph neural network (GNN) is emerging as a powerful extension of deep neural network (DNN) for analyzing non-Euclidean graph data. In particular, GNN's inductive learning [31] introduces a trainable generator using DNNs to compute node and edge embeddings. This generator is then capable of inferring embeddings for new unseen nodes and edges. Such a feature makes it successful with demonstrations of cutting-edge results in various applications, such as node classification [24], link prediction [80], relation extraction [86], recommendation [75], quantum chemistry [74], and risk control [48].

Combining with big data, large-scale GNN's application at enterprise (such as Alibaba's e-commerce and e-payment business) encounters unique challenges. Such a GNN usually works on billions-node graphs with rich attributes, demanding 10-100TB memory footprint. We denote GNNs at this scale as **Large Scale Distributed GNN (LSD-GNN)**, since the graph has to be carried out with a CPU-centric distributed in-memory storage. LSD-GNN can only work with mini-batch training and hence adopts a two-step workflow with sampling first and then dense neural network (NN). This is very different from a **Single Machine GNN (SM-GNN)**, in which the graph data is small enough to fit in a GPU's device memory or a single machine's host memory. Therefore, it does not have issues with graph storage nor inter-machine communication. SM-GNN can afford full/global-batch training, so it usually adopts sparse matrix multiplication (sparseMM) operators.

Although lots of work have demonstrated novel hardware for GNN [1], they are all limited to SM-GNN. Unfortunately, techniques that dedicated to SM-GNN, do not apply to LSD-GNN that actually

deployed in the enterprises, for the following reasons. First, operators and workflows are different. SM-GNN optimizes sparseMM operations on local memory hierarchy, following global/full-batch training strategies. However, it is not adaptable for LSD-GNN that uses mini-batch training strategies, in which 2-step sampling and NN are used instead of sparseMM. Second, bottlenecks and focuses are different. SM-GNN focuses on intra-node dataflow, but LSD-GNN is naturally heterogeneous and distributed with bottlenecks on inter-node graph sampling. Third, design goals are different. LSD-GNN highlights the demands for programmability and scalability, since the GNN model is actively developing while the demands for larger data never stop (e.g., most current solutions still use trimmed data). On the opposite, most SM-GNN work's model-specific scaling-up solutions cannot meet such demands.

Meanwhile, we find hyperscale FPGA-as-a-Service (FaaS) emerges as a promising solution. Although customized model-specific ASIC (e.g., GNN-dedicated) chips draw a lot of attention, they suffer from weak programmability and low return-on-investigation (ROI) ratio. Instead, FPGA cloud is naturally scalable and programmable. The infrastructure is off-the-shelf, and hence provides near-zero NRE cost with decent performance (i.e., high performance/dollar).

The goal of this paper is to provide a practical solution to the LSD-GNN's challenges at hyperscale that can be profitable, scalable, and programmable. To this end, we first study the LSD-GNN workload, and quantitively demonstrate its difference from the SM-GNN. With characterization of typical industrial-grade framework, data, and applications, we present the key observations that LSD-GNN is facing the challenge of the sampling stage in GNN and hence bottlenecked by the long latency irregular memory access over fabric. Then, we solve such problems with a customized hardware architecture, including a fully pipelined access engine with a producer-consumer architecture for graph access and sampling, a low-latency and bandwidth-efficient customized memory-over-fabric hardware, and a RISC-V centric control system providing programmability. Further, to deploy the solution at hyperscale, we integrate the customized hardware with an industry LSD-GNN framework to provide a near-transparent user interface, and then apply it to the FPGA cloud to provide scalability, accessibility, and performance per cost improvement. As a highlight, we also present a comprehensive exploration of the FaaS system architecture for deploying the proposed hardware. The exploration includes currently available *FaaS.base*, *FaaS.cost-opt* with network interface integrated on-FPGA, heterogeneous GPU-FPGA cloud system with GPU memory extension, *FaaS.comm-opt* with inter-FPGA dedicated fabric, and finally *FaaS.mem-opt* with dedicated local DRAM on FPGA. Finally, to evaluate the architecture, we implement the proposed hardware and its corresponding software with a 4-card FPGA as a proof-of-concept (PoC) system. Based on the measurement result from the FPGA PoC, we show the effectiveness of the hardware architecture, as well as various FaaS system deployment solutions' speedup and performance/dollar improvement. We conclude that off-the-shelf *FaaS.base* can already provide 2.47× performance per dollar improvement with our hardware. With further modification on FaaS architecture, the improvements can reach 7.78× and 12.58× with *FaaS.comm-opt* and *FaaS.mem-opt*, respectively. Our contribution is summarized as follows.

- We introduce the uniqueness of LSD-GNN that the previous SM-GNN-focused solutions have overlooked, and make the observations of LSD-GNN's challenges at hyperscale system.
- We propose a customized scalable and programmable hardware architecture to cope with LSD-GNN's challenges, and integrate the hardware with an industrial-grade framework.
- We propose a practical solution of leveraging FaaS with proposed hardware for achieving accessibility, scalability, and flexibility.
- We extensively explore a variety of FaaS system architecture solutions, posting the suggestions for future FaaS system designs.
- We evaluate the proposed hardware with a 4-card FPGA implementation. Based on the measurement result, we project the various FaaS performance and conclude their pros and cons.

2 BACKGROUND

In this section, we introduce general GNN model workflow and the cloud service background.

2.1 Graph Neural Network (GNN)

We introduce a general Graph Neural Network (GNN) background following AliGraph [87]'s programming model. *Sample:* Given a root node v, GNN samples a subset $S(v)$ in the neighbor set $N(v)$ with a sampling method (e.g., random). The attributes (a.k.a. features) of sampled nodes are then fetched. The sampling goes iteratively for multiple hops. *Aggregate:* The sampled attributes at the same hop are then aggregated to one. The aggregation operator (e.g., *Max*, *MLP*) is flexibly defined by model. *Combination:* The aggregated attribute is further combined with the attribute of its source vertex. The combination operator is also flexible. The k^{th} layer formulation is described as follows.

$$a_v^k = \textbf{Aggregate}(h_u^{(k-1)} : u \in S(v) \cup v), \tag{1}$$

$$h_v^k = \textbf{Combine}(a_v^k), \tag{2}$$

in which $S(v)$ is a sampled subset of $N(v)$, and h_v^k denotes the attribute/embedding of v.

This is an industrial-grade programming model with sufficient representatives for a large variety of GNN models. Figure 1 shows an example Operator (OP) graph for GNN. The *Aggregate* and *Combination* (combined as *GNN-NN* stages) are executed alternatively layer by layer, after a *GNN-sample* stage.

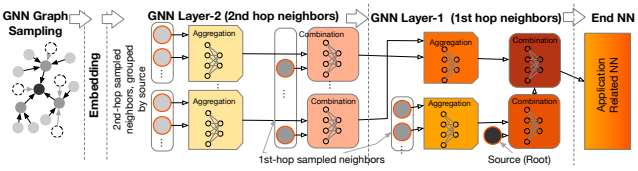


Figure 1: Typical 2-hop (layer) GNN workflow and operator breakdown.

2.2 Full-batch v.s. Mini-batch

Full-batch training processes GNN for all nodes at once, which is a sequence of sparse-dense matrix-matrix multiplications (sparseMM, $N - N$ adjacent matrix and $N - l$ attribute matrix). On the contrary, mini-batch works at the individual node level, no matrix-matrix anymore. It takes one row of the binary sparse adjacent matrix and

multiples it to the attribute matrix. Instead of applying a sparse vector-matrix operator, it prefers a gather (sampling) operator.

Full-batch training costs the minimal possible computations, however, at the price of at least doubled memory footprint. It is preferable at small scale, but not practical for large graphs at 10TB level [43].

2.3 Cloud FPGA Service

FPGA infrastructure as a service (FaaS) is widely provided by major cloud providers [2, 7, 9, 12]. FPGA acceleration on cloud offers unique advantages over GPU/NPUs [21, 21, 22], including low latency, customizable data path for irregular operators, etc. An instance contains a combination of virtual CPU (vCPU) cores, FPGA board(s), DRAM quota, and virtual network channels from a physical machine. The machine might have dedicated inter-FPGA connections [12] and customized network interface card (NIC) [3, 8].

2.4 Framework Support

We use AliGraph (a.k.a. Graph-learn [28]) [87] framework as the baseline and user interface. It is an open-source [28] industrial framework, serving publicly through Alibaba Cloud [4]. The AliGraph framework consists of the graph sampling and the GNN operator support. It is optimized for multi-threading and hyperscale system. It supports a large variety of GNN models, including heterogeneous graph and dynamic graph. It is demonstrated with real-world e-commercial in-house graph data and applications, e.g., a large graph with 492.9M-vertex, 6.82B-edge. The framework outperforms competitors by 4.12% to 17.19% [28].

3 CHARACTERIZATION AND CHALLENGES

In this section, we present LSD-GNN's characterization, with the highlight of its difference against SM-GNN.

3.1 Industry LSD-GNN Use Cases

With a summary in Table 1, we highlight the difference between LSD-GNN and SM-GNN at high level as follows. These findings are supported by quantitative studies later in Section 3.3.

1) Huge data need hyperscale: E-commerce graphs often contain billions of vertices with rich attributes (at 10TB-level). The scale of the data is too big to fit in GPUs or a single machine. Even worse, the data size keeps expanding. Distributed in-memory storage is the only option.

2) Workflow difference: As described in Section 2.2, full-batch training costs the minimal possible computations, however, at the price of at least doubled memory footprint. SM-GNN can afford full-batch training and implement sparse matrix operations on GNN, but LSD-GNN cannot due to full-batch's expensive memory cost. On the opposite, LSD-GNN adopts mini-batch training with decoupled graph sampling and then dense matrix computing, in order to avoid the cost [43], which is adopted by popular industry frameworks like AliGraph [87] and AGL [79].

3) End-to-end heterogeneity: Unlike SM-GNN research that uses a single chip for end-to-end GNN application, LSD-GNN is more complex and heterogeneous and most efficient when using (a) CPU for distributed graph sampling, then (b) CPU for optional trainable embedding layer, and (c) finally GPU for dense matrix computing. This paper focuses on the bottleneck stage, i.e., the graph sampling, leaving the embedding and dense matrix stages that hold strong demands for programmability to the well-optimized existing hardwares.

	LSD-GNN	SM-GNN
Graph Size	>10TB	<1TB
Mem. Demand	Distributed system memory	GPU device memory
Workflow	Mini-batch, sample then denseMM	Full/global-batch, sparseMM
Bottleneck	Graph sampling	Sparse-NN
End-App DNN	Heavy compute, diverse	Small/None
Comp. Demand	Distributed hetero-system	Standalone Accel.

Table 1: LSD-GNN v.s. SM-GNN

3.2 Characterization Setup

We present characterizations on various LSD-GNN sampling cases. Table 2 shows six graph datasets we derived from typical internal workload in Alibaba. We rename the graph name according to its scale (e.g., *ml* represents the graph with medium-scale nodes and large attribute length). *syn* is a synthesized large graph, which has the same node/edge numbers from real application, but a synthesized adjacent matrix scaled from a smaller graph. The fifth column (*model*) in Table 2 shows the sampling application setup, in which we use a typical 2-hop random sampling method with 512 mini-batchsize and sample rate of 10. The last column shows the minimal number of servers needed to carry out the corresponding graph data. Table 3 shows an end-to-end GNN application setup. It uses the graph data *ls* and sampling method described in Table 2, 128 length embedding size, graphSage-MAX GNN model, which is followed by a DSSM [35] end application.

We use Table 2 and Table 3 to capture the characterization of LSD-GNN's features mentioned in Table 1: The graph data is processed using mini-batch sampling at a distributed system; The end-to-end application combines with embedding and an end application.

	Node	Edge	Attr. Len	Model	Server #
ss	65.2M	592M	72	batchsize: 512,	1
ls	1.9B	5.2B	84	negative sample rate: 10,	5
sl	67.3M	601M	128	hop (layer): 2,	1
ml	207M	5.7B	136	sample rate: 10/10,	5
ll	702M	12.3B	152	random sample, hidden/	15
syn	5.9B	105B	152	embedding size: 128	128

Table 2: LSD-GNN sampling graph data configurations (Note: sample rate 10/10 mean both layers sample 10 neighbors per node).

Graph	Embedding	GNN-NN	End-model
ls in Table 2	128	graphSAGE-max	DSSM [35] 128-128
Instance: 5-server 120-worker			

Table 3: An end-to-end LSD-GNN application.

We run the characterization with AliGraph [87] framework (see Section 2.4) in a datacenter serverless environment (an internal

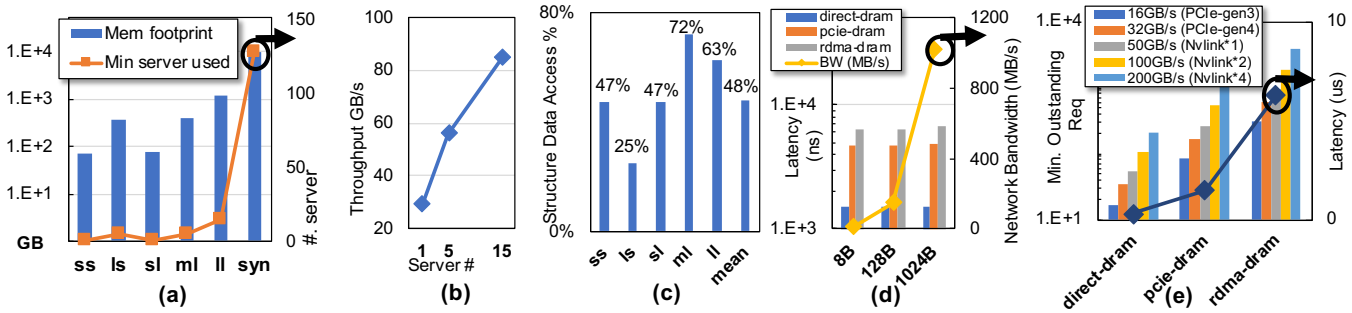


Figure 2: (a) Memory footprint for LSD-GNN applications and minimal servers to carry it. (b) Performance scaling with number of servers (average across all benchmarks). (c) Memory access request distribution. (d) Round-trip latency/bandwidth of remote memory access for various request length. (e) Outstanding request demands to fill the bandwidth.

version of Alibaba Cloud PAI [4]). AliGraph has the logic *server* and *worker* concepts for resource assignment. Servers (a process with a group of vCPU at hyperscale, not physical servers) work on attribute fetching while workers process graph traversal and NN. The cloud infrastructure carrying our experiments uses a dual-socket Intel Xeon 8163 CPU.

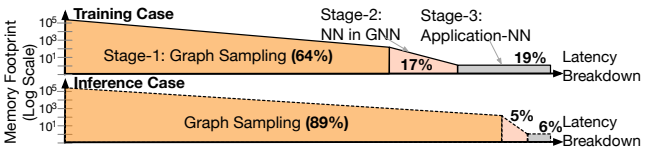


Figure 3: End-to-end LSD-GNN characterization.

3.3 LSD-GNN Characterization

Figure 3 and Figure 2(a)-(e) show the result, in which Figure 3 is an end-to-end case study with the application described in Table 3. Figure 2(a) shows the memory footprint of the six graph data in Table 2. Figure 2(b) shows the performance scalability when the number of servers increases. Figure 2(c) shows the memory access pattern in the six datasets, i.e., the percentage of fine-grained graph structure data indirect access in the entire memory traffic. Figure 2(d) shows the relationship between the communication latency/bandwidth and the access data granularity. Figure 2(e) shows the demanded outstanding request to hide the latency and fully utilize the bandwidth, in various hardware configurations.

Observation-1: Sampling is the bottleneck. Figure 3 shows the end-to-end training and inference case. Sampling stage is the latency bottleneck, which takes 64% and 88% in breakdown for training and inference, respectively. This is different from SM-GNN, since LSD-GNN randomly samples data in distributed system.

LSD-GNN sampling is also the bottleneck in graph storage, taking 5 orders of magnitude larger than NN in Figure 3. Figure 2(a) further shows memory footprint for six graph datasets shown in Table 2, and hence the minimal number of servers to carry the graph. It shows that the large graph footprint forces the adoption of multiple nodes.

Observation-2: Sampling is bounded by communication, but bandwidth is underutilized. Figure 2(b) shows the server scalability by using 1, 5, and 15 serves. The sampling throughput suffers a sub-linear performance scaling with the increase of servers, due to the inter-node communication overhead. We further find the inter-node bandwidth is underutilized due to the random yet fine-grained (remote) memory access pattern. Figure 2(c) shows the memory access pattern in the six graph data. It shows that, on average, 48% of the memory accesses are for graph structure (e.g., pointer, node ID), which are indirect pointer chasing fine-grained (8 - 64Btye) access. Remote access bandwidth for small packages is 100× smaller than its peak bandwidth (compared to 8B and 1024B access data size), as shown in Figure 2(d) [54]'s right y-axis.

Observation-3: Long latency worsens the performance. Figure 2(d) shows the latency and bandwidth of three hardware configurations (by legends) with various package sizes [54]. The left y-axis and the bars show the latency. The local direct DRAM access (blue bar) takes the minimal latency, but the PCIe connected local host DRAM access (orange bar) shows a significant latency increase, followed by the RDMA access remote DRAM that takes the longest latency. The x-axis shows the size of the access data from 8Byte to 128 Byte. It demonstrates that long latency is expected when using PCIe-connected DRAM or remote DRAM, even though access data is small (the 8Byte case). The right y-axis and the yellow line further show the bandwidth of the RDMA-access remote DRAM case. It shows that smaller data size (e.g., 8Byte) seriously hurts the bandwidth utilization, due to its long latency.

In order to hide the latency for improving bandwidth utilization, we need to have enough on-the-fly (a.k.a., outstanding) requests. Figure 2(e) shows required outstanding requests fitting various link (by legends) bandwidth. For example, in the case of direct local DRAM access, since the latency is smaller (right side of the y-axis), we need fewer concurrent requests to fulfill the corresponding bandwidth (represented by the legends from 16GB/s to 200GB/s); but we need more outstanding request in the case of remote DRAM with longer latency to hide.

3.4 Limitation of existing solutions

System for large graphs: Software-based optimizations for distributed graphs has already been applied to LSD-GNN framework (AliGraph). The limitation is the efficiency and cost: too much vCPU resource for sampling is required to meet the demands of sampling throughput. It motivates this paper to offload the task to FPGAs for cost-down, while existing graph accelerators are not capable of processing 10TB-level graphs.

Sparse tensor accelerators: Sparse accelerators focuses on single-machine sparseMM, which is suitable for full-batch SM-GNN (see Section 2.2). However, mini-batching LDS-GNN demands cross-machine gathering for sampling operators, rather than general sparse tensor algebra.

3.5 Summary: New Challenges and Goals

To summarize, we observe that the challenges for LSD-GNN are very different from those of the SM-GNN (as shown in Table 1). In LSD-GNN, the bottleneck turns out to be the mini-batch sampling on graph data at hyperscale. The specific challenges of distributed graph data sampling are:

Challenge-1: The long remote access latency and the inefficient latency hiding capabilities. The long latency could result in either the failure of meeting real-time deadline in some inference scenarios, or the underutilization of memory/network bandwidth that reduces the throughput in training. Unlike improving throughput, latency cannot be improved by purchasing more existing hardware (e.g., more vCPUs). To tackle this challenge, we design a customized and efficient processing hardware (see Section 4.2) that can (1) bypass unnecessary software layers and memory hierarchy to save latency, and (2) provide massive concurrent memory requests to hide the latency.

Challenge-2: The underutilized remote access bandwidth. In addition to the latency, the underutilization is also caused by inefficient interconnection and GNN's random and fine-grain memory access pattern. As mentioned in Observation-3 and Figure 2(d) that smaller data access over PCIe or network still takes very long latency and hence results in serious bandwidth underutilization. Instead of general-purpose remote memory access optimization, a dedicated and customized protocol/hardware is demanded to solve these problems.

Our goal is to tackle these challenges with customized hardware and system architecture. Beyond this, we prefer a practical and short-term solution and hence fulfilling the guidelines of being **profitable, scalable, and programmable**. (1) To be profitable, the solution must have a reasonable return-on-investigation (ROI). Specifically, since GNNs are emerging but not dominating (unlike CNN), a dedicated customized ASIC chip may not make economic sense. (2) Scalability is a strong demand, since the large amount of ever-increasing graph data needs a scalable solution. Furthermore, today's GNN models are limited by current hardware and have to adopt some pruning of the graph data. We therefore envision that the demand for scalability will keep increasing. (3) As GNN models keep actively developing, it require flexibility and programmability. Any hardware must be compatible with the software/framework (e.g., AliGraph) to support algorithm development.

4 ARCHITECTURE

In this section, we propose a customized hardware architecture to cope with the LSD-GNN's challenges.

4.1 Overview

We adopt customized domain-specific hardware to overcome the LSD-GNN's challenges (Section 3), targeting at optimizing memory accesses acceleration for sampling. Different from most domain-specific accelerators which are compute-centric, in the sampling for LSD-GNN, we target at memory accesses. For *Challenge-1*, we propose an access engine (AxE) architecture with customized hardware for low latency and massive parallelism for latency hiding (Section 4.2). For *Challenge-2*, we adopt a near-data processing system architecture along with a customized memory-over-fabric (MoF) (Section 4.3). In addition to solving the challenges, we follow the design guidelines of profitable, programmable, and scalable. For *programmability*, we adopt a RISC-V based control interface. For *scaling out*, the MoF is designed for supporting multi-node communication. For *scaling up*, AxE is a multi-core based architecture, the number of cores and core IO components can be adjusted according to applications.

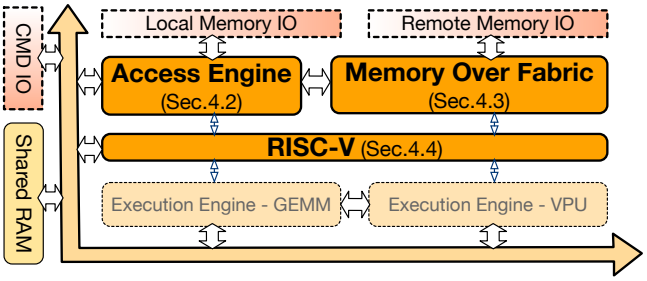


Figure 4: An overview of the decoupled access-execution hardware architecture for graph sampling.

Figure 4 shows the overview of the decoupled access-execution hardware architecture. We have Access Engine (AxE), an accelerator module, which deals with graph traversal and sampling on the graph data from both the AxE direct-connected local memory and remote machine. AxE is connected to Memory-over-Fabric (MoF) module, through which it requests or responds to remote machine for graph data partition at the remote side. Both the AxE and the MoF are connected to the central RISC-V processor with queues for controlling. There is a central SoC bus (e.g., AXI smart connect [61]) connecting AxE, RISC-V, a shared memory, and IO component (e.g., PCIe IP), for data sharing.

Additionally, an optional FP32 (application demands high precision) general matrix-multiplication engine (GEMM) [55] and an optional vector processing unit (VPU) [64] can be added to the design. Although FPGA's FP32 TFlops is not competitive with GPU or even CPU, GEMM/VPU might be useful in latency-sensitive inference tasks with simpler model, in which case data movement from FPGA to local or remote GPU can be eliminated. Also, the FPGA compute units are preferable for reductions in the sampling stages in order to reduce communication overhead, such as the case for GCN. The use case and scheduling for GEMM and VPU in an end-to-end LSD-GNN pipeline is beyond the scope of this paper.

The architecture is highly parametrizable. First, AxE, MoF, GEMM, and VPU are all scalable (e.g., core/channel numbers, array/lane width) according to high-level specification and resource availability. Second, the architecture can be integrated with various IO components. Table 8 shows example configurations to support various scenarios. With different IO components and bandwidth settings, AxE and MoF can be adjusted accordingly.

4.2 Access Engine (AxE)

Compared to C-based software solution, Access Engine (AxE)'s goal is to support an order of magnitude more sampling requests in flight in the memory system. Conceptually we achieve this by building a dedicated memory access engine that is closely coupled to the memory controller, aware of the access patterns needed to be generated for sampling graph nodes. AxE adopts a customized hardware approach to overcome the long latency and inefficient latency hiding challenges (*Challenge-1*). At the same time, it provides flexibility for various configurations with different IO component integration solutions and core configurations. To reach those goals, we propose four micro-architecture techniques, including fine-grained async-pipelining, streaming-based sampling, OoO-supported massive request generation, and LSD-GNN application-specific caching.

Module Description and Workflow: Figure 5 shows an overview of AxE. The central data path in AxE (dotted light gray) is a homogeneous multi-core architecture, providing the **scaling capability**. For each core, one sampling task is for a batch of nodes. Task on each core is independent so that there is no data sharing or coherency issues. The command comes from the RISC-V through the *decoder* module, and the top scheduler module initiates the process by setting the *Control/status registers (CSRs)* and distributing the task to cores accordingly. The data movement unit fetches the input node ID from command IO (e.g., PCIe) to a target core's *buffer*. In each core, the *GetNeighbor* module fetches the neighbor node ID according to the root node ID stored in the *buffer*, as well as the edge weight/attribute if applicable. The *GetSample* module samples the neighbors according to the configured sample method. The sampled nodes are optionally written back to the *buffer* for multi-hop sampling. The following *GetAttribute* module fetches the sampled node's feature/attribute. All these modules are connected to the *LoadUnit* for both local and remote memory access. The results are written to either the shared on-chip RAM or the command IO (e.g., PCIe). AxE then issues responses through *encoder* to indicate the task is done.

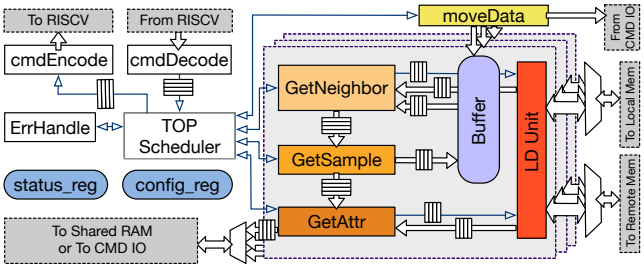


Figure 5: Access engine architecture.

Table 4 shows typical commands (not a complete list) the AxE takes to support AliGraph framework [28] (Section 5) for **programmability**.

Micro Architecture - Pipelining: Targeting at efficiently hiding the long latency, we adopt **Tech-1: fine-grained FIFO-connected asynchronous producer-consumer streaming architecture** to achieve very deep pipelining. Figure 6 zooms in module *GetNeighbor*, in which five sub-level modules are pipelined, some of which have one more hierarchy with finer level pipelines. Figure 7 shows the significance of the pipeline depth in terms of processing throughput: deeper pipeline, better performance. As an additional benefit, such loosely coupled dataflow naturally supports the supernode scenario.

command	input	arguments
set/read CSR		
sample n-hop	root nodes	sample method/rate, w/ or w/o edge/node attribute
read node attribute	nodes ID	batch size
read edge attribute	node pairs ID	batch size
negtive sample	node pairs ID	sample rate

Table 4: E.g. AxE command to support AliGraph.

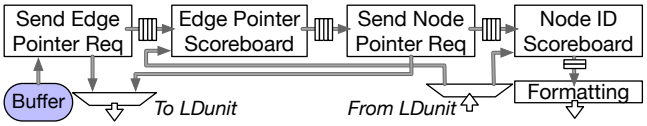


Figure 6: GetNeighbor submodule micro-architecture.

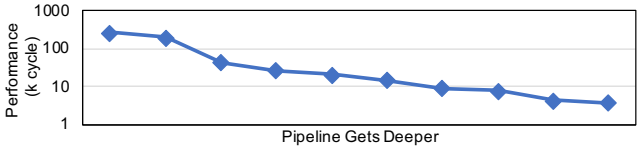


Figure 7: Measured performance (latency) v.s. pipeline depth.

We propose **Tech-2: streaming sampling** method to overcome the long latency and heavy resource usage by conventional sampling hardware. Random sampling is the most frequently used GNN sampling method, and it is the base for many other sampling methods, such as degree-based sampling. Conventional random sampling hardware suffers from long latency and heavy resource utilizations. To randomly sample K nodes out from N nodes, it needs N space to store all candidate nodes, and in total $N+K$ cycles to complete the sampling. We propose a step-based approximate random sampling method, in which we divide the N nodes into K groups, by the order they come in (e.g., 1 to N/K in the first group). In each group, we generate a random number $k \in N/K$ for each group, and select the k^{th} node in the group as sampled. The step-based sampling does not need extra storage or synchronization, and it is a streaming-based approach which naturally fits in the pipeline architecture. It reduces the latency from $K+N$ cycles to N and saves 91.9% LUT and 23% registers (on FPGA). The proposed method has an ignorable impact on the model accuracy (e.g., steaming sampling reaches 0.548 on PPI, while standard method reports 0.549).

Micro Architecture - Load Unit: We further customize the load unit for memory latency reduction and hiding by proposing **Tech-3: OoO-supported massive outstanding request generation** with massive memory-level parallelism (MLP). First, we maintain the request dependency by embedding context into the memory request and response. Usually, the context is maintained by a thread in a processor, causing non-trivial context storage and switching overhead. Instead, our design reduce the size of context into a

128-bit tag to the memory request/response package, reducing the overhead and eliminating the MLP limitation caused by context storage. Second, we enable the out-of-order (OoO) memory access by maintaining the order and synchronization with the score-board hardware. As shown in Figure 6, we have two score-boards to maintain the order after OoO requests are issued. The first score-board maintains the order of the root node, required by GNN training loss computing. The second one maintains the order of neighbors, so that neighbors from different roots are synchronized. Our implementation shows that the OoO design can improve the throughput by 30×.

Another AxE innovation is **Tech-4: LSD-GNN application-specific caching**. Different from study on SM-GNN [39, 40, 44, 77], which leverages reuse between nodes, we find very limited reuse in LSD-GNN, because the ratio of the small number of batch size (e.g., 512) over the large number of nodes (e.g., 10 billion) is too low. In addition, framework (i.e., AliGraph) already provides system-level caching for the most frequently used nodes. Therefore, we find caching temper reuse is not efficient in the hardware. Instead, we propose to design only 8KB cache just for coalescing. Such a small cache is good enough to capture the spacial reuse for continuously stored edge and attribute. It provides maximal reuse coverage but with minimal resource.

4.3 Memory Over Fabric (MoF)

The MoF is a customized lightweight interconnection between FPGAs. It provides data-link capability with high reliability without much software overhead, and hence overcomes the bandwidth underutilization problem. We further propose two techniques. First, we define the MoF frame with **Tech-1: multi-request in single package**, since GNN memory access patterns are fine-grained (Figure 2(c)). Such small data size makes the package header and the read package (with 64-bit address per request) a significant overhead. Although GENZ [26] has package with 4 requests, it is still not enough. Table5 demonstrates a comparison. To send 128 16B/64B packages, GENZ needs 64 packages while our proposed method only uses 2 (64 requests per package). Therefore, the header and address overhead is reduced and package utilization is significantly improved. The advantage is more obvious when sending small data (e.g., 16B data), which is exactly the case of fine-grained memory access in GNN sampling.

	request size	number of packages	Header Overhead	Address Overhead	Data (utilization)
genz	128×16Byte	64	51.02%	10.20%	32.65%
proposed		2	2.36%	19.53%	78.11%
genz	128×64Byte	64	25.77%	8.25%	65.98%
proposed		2	0.09%	5.88%	94.03%

Table 5: Bandwidth utilization compared with GENZ multi-read package.

Second, we propose **Tech-2: address compression**. In addition to conventional data compression, we further compress the address in the request package. This is because that, for fine-grained remote memory accesses, request package with 64-bit address has the comparable bandwidth consumption with the data (e.g., 8Byte). Table 6 shows a simple Base-Delta-Immediate Compression (BDI) compression result on data and address for sending 128 8Byte data. With MoF's Tech-1, we already reduce header overhead and send only 1600Byte instead of 6336Byte data. With data compression, the saving further goes to 46%. After adding the address compression, we only need to send 779Byte data.

	GENZ	MoF	MoF w/ data comp.	MoF w/ addr. comp.
Byte to Send	6336	1600	864	779
Saving	–	75%	46%	9.8%

Table 6: BDI compression on 8B×128 read package.

4.4 RISCV-based Controller

To reach the best **programmability**, we leverage a RISC-V based central controller. The RISC-V hooks up all customized accelerator modules and controls them through ISA extension. It forms the software/hardware interface, i.e., user can write C code on RISC-V to control the entire hardware. Users can move data around, send configurations and commands to corresponding modules, check status, and maintain data dependency.

	MMIO	ISA-ext	*QRCH*
Interaction	~100 cyc	~1 cyc	~10 cyc
Programmability	Bad (coarse-grain)	Good (fine-grain)	Fair (small OP level)
ToolChain Dev	Hard	Fair	Easy
Extensibility	Bad	Fair	Good

Table 7: *QRCH* v.s. other design choices.

We propose queue-based RISC-V coprocessor communication hub (*QRCH*) to connect customized hardware with RISC-V instruction extension. Different from loosely coupled memory-mapped IO (MMIO) [49, 52] solution connecting accelerators through the bus (e.g., AXI), or tightly coupled instruction extension [41] solution integrating accelerators into the processor's pipeline, *QRCH* demonstrates its advantages in Table 7, considering the tradeoff among interaction latency, programmability, and extensibility. *QRCH* provides a fairly fast interaction speed, which consists of the instruction latency to fill the queue, and the accelerator's latency to read the queue (~10 cycles depending on queue width and accelerator design) cycles. Second, it provides decent programmability leveraging the instruction extension and fine-grained control commands. Third, it is easy to extend to various accelerators, since the extended instructions and its corresponding tool-chain modifications are independent of the queues and accelerators.

Figure 8 shows the micro-architecture design to implement *QRCH* with Xuantie 906 RISC-V core [59]. With modification of frontend for decoding the customized instructions, the *QRCH* logic is integrated into the execution stage. We describe more details in our technical report [20].

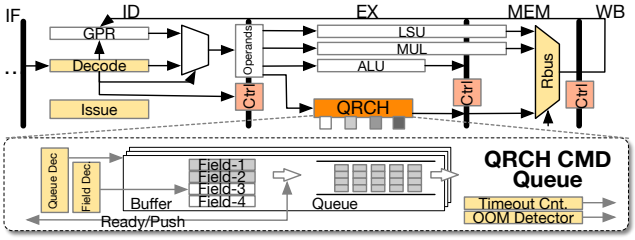


Figure 8: Integrating *QRCH* to RISC-V pipeline.

5 SOFTWARE STACK

To provide enough programmability for the model development, we integrate our hardware with AliGraph [28, 87] (see Section 2.4). We expose various levels of programming interface, in addition to the framework. We provide various level programmability, including (1) ISA level for RISC-V *QRCH* , (2) accelerator operator-level (e.g., read/write CSR), (3) algebra operator-level (e.g., GEMM), and (4) GNN operator-level (e.g., 2-hop sampling). Beyond that, we also provide frequently used fixed model APIs (e.g., graphSAGE). All above are finally integrated with the AliGraph framework interface. Details are described in our technical report [20].

6 SYSTEM EXPLORATION

In this section, we explore approaches to deploy the proposed hardware and software from system perspectives.

6.1 Motivation and Architecture Taxonomy

We choose the FPGA-as-a-Service (FaaS) hyperscale solution to meet our goal of being profitable and scalable. Although customized ASIC could provide better performance, its significant non-recurring engineering (NRE) cost cost makes the ROI less attractive than the off-the-shelf FaaS solutions. To fully unleash the performance of the *FaaS.based* solution, we present a design space exploration (DSE) for the FaaS system architecture. We envision our DSE will provide a guideline for FaaS system's future developments.

We explicitly explored *eight* FaaS architectures by introducing a 2-level taxonomy, summarized in Table 8. The first taxonomy variable is design constraints (*base, cost-opt, comm-opt, mem-opt*), about which we have four types of architectures. The second taxonomy variable is the coupling method. If the FPGA and GPU are integrated in one single heterogeneous server, we denote such architecture as *tc (i.e., tightly coupled)*. If the FPGA and GPU are decoupled into separated all-FPGA and all-GPU machines, we denote these as *decp (i.e., decoupled)*. That results in eight architectures, including the off-the-shelf baseline architecture (*base (tc/decp)*, Section 6.2), the cost-optimized FPGA w/ integrated NIC architecture (*cost-opt (tc/decp)*, Section 6.4), the communication-optimized FPGA w/ dedicated fabric architecture (*comm-opt (tc/decp)*, Section 6.3), and the memory-optimized FPGA w/ dedicated local DRAM architecture (*mem-opt (tc/decp)*). In the remaining sections we describe in detail each of the design points.

6.2 Off-the-shelf FaaS (base)

Figure 9 shows the system architecture to leverage the off-the-shelf Alibaba Cloud FaaS. Figure 9 (a) shows *base.decp*, in which FPGA and GPU are decoupled into different servers and communicate with network. Figure 9 (b) shows *base.tc*, in which FPGA and GPU are tightly coupled in one single heterogeneous server that carries both cards.

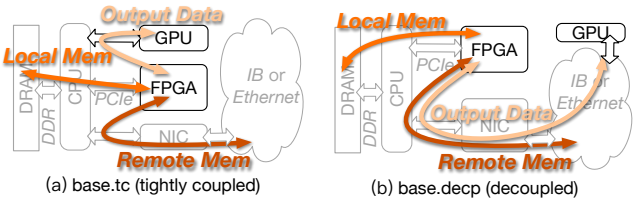


Figure 9: *base*: Off-the-shelf FaaS architecture.

We describe how to adapt the architecture proposed in Section 4 to implement *base.decp* and *base.tc*. First, we configure *CMD/Data*, *Local Mem*, and *Remote Mem* IOs accordingly. As summarized in Table 8, we use PCIe for *CMD/Data IO*, PCIe-connected host DRAM as *Local Mem IO*, PCIe-NIC accessed remote node as *Remote Mem IO*. We then tailor or tune the MoF and AxE micro-architecture to suit the performance requirement of *base.decp* and *base.tc*, while addressing the long latency problem mentioned in *Challenge-1*. MoF is tailored to adapt to the PCIe-NIC interface, with a customized DMA engine to bypass the host interference. To hide the long latency caused by the PCIe-NIC connected remote memory access, we leverage the scalability of the architecture and tune the number of AxE core. We decide the number of AxE cores by calculating the number of outstanding requests required to hide the latency. We have

$$O_i = \frac{B_i}{\sum_k C_k \cdot P_k} \cdot L_i, i \in \{\text{local, remote}\}, k \in \{\text{all access pattern}\} \quad (3)$$

in which B_i, L_i denote the local/remote memory access bandwidth and round trip latency, C_k and P_k denote the data length and probability for each memory access pattern, O_i denotes the outstanding request demand. Note that B_i is the effective (not peak) bandwidth, taking considerations of overall system bottlenecks. We conclude that three AxE cores are needed for *base.decp* and *base.tc*.

6.3 On-FPGA integrated NIC (cost-opt)

To fulfill the goal of profitability, we propose *cost-opt.tc* and *cost-opt.decp* FaaS architectures, shown in Figure 10. They leverage FPGA logic and PHY resource to build an on-FPGA NIC, which eliminates the demands for a dedicated NIC card. Figure 10 (a) shows the tightly coupled same-server FPGA/GPU case, and Figure 10 (b) shows the decoupled case.

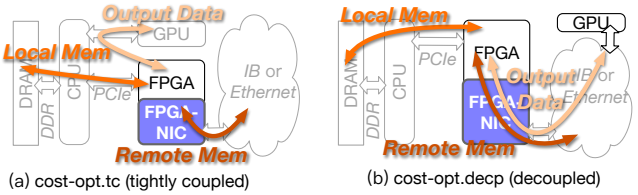


Figure 10: *cost-opt*: FaaS architecture with on-FPGA integrated NIC.

We describe how to adapt the architecture proposed in Section 4 to implement *cost-opt.decp* and *cost-opt.tc*. Similar to *base*, we use PCIe for *CMD/Data IO*, and PCIe-connected host DRAM as *Local*

Primary Design Constraints	*Integration*	FPGA-FPGA Connection	Local Mem Access	Remote Mem Access	FPGA-GPU Connection
base	**.tc**	PCIe→NIC	PCIe→HostMem (16GB/s)	PCIe→NIC→PCIe →HostMem (16GB/s)	in-server PCIe P2P (16GB/s)
	.decp				across-server PCIe→NIC (16GB/s)
cost-opt	**.tc**	**[opt]** on-FPGA NIC (16GB/s)	PCIe→HostMem (16GB/s)	PCIe→NIC→PCIe →HostMem (16GB/s)	in-server PCIe P2P (16GB/s)
	.decp				across-server on-FPGA NIC (16GB/s)
comm-opt	**.tc**	**[opt]** Dedicated Customized Fabric (MoF, **100GB/s**)	PCIe→HostMem (16GB/s)	MoF→FPGA→PCIe →HostMem (16GB/s)	in-server PCIe P2P (16GB/s)
	.decp				across-server PCIe→NIC (16GB/s)
mem-opt	**.tc**	**[opt]** Dedicated Customized Fabric (MoF, **100GB/s**)	**[opt]** FPGA local DRAM (**102.4GB/s**)	**[opt]** MoF→FPGA local DRAM (**100GB/s**)	**[opt]** in-server fast link (**300GB/s**)
	.decp				across-server PCIe→NIC (16GB/s)

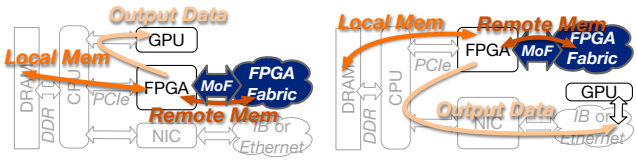
Table 8: Summary of eight FaaS architectures in the DSE. (Tightly Coupled: *tc*. Decoupled: *decp*.)

Mem IO, but on-FPGA NIC as *Remote Mem IO* to access remote node, as summarized in Table 8. We modify the MoF so that it interfaces with FPGA-NIC PHY without any host interference, running the NIC protocol but still adopting the packing and compression techniques described in Section 4.3. Such design bypasses PCIe and hence further copes with *Challenge-1*, reducing the long latency. For AxE tuning, now we only need two AxE cores according to the modeling in Equation 3, since the on-FPGA NIC reduces the remote memory latency.

We briefly explain how on-FPGA NIC design reduces server purchasing cost (not necessarily operating cost) for three reasons: (1)The elimination of the dedicated NIC; (2) The saving of NIC form factor provides potential for a denser integration; (3) The shorter latency due to on-FPGA NIC reducing the number of AxE cores.

6.4 Customized inter-FPGA fabric (comm-opt)

Recall *Challenge-1* and *Challenge-2* in Section 3.5 that LSD-GNN is bottlenecked by communication latency and bandwidth. To remove such bottleneck in the FaaS system, we propose the FaaS architecture with dedicated customized inter-FPGA fabric, as shown in Figure 11. Such inter-FPGA fabric (e.g., Direct Attach Copper cables with SFP or QSFP interface) provides larger bandwidth and smaller latency than typical NICs in cloud infrastructures. Figure 11 (a) and (b) show *comm-opt.tc* with same-server FPGA/GPU and *comm-opt.decp* with decoupled FPGA/GPU, respectively.



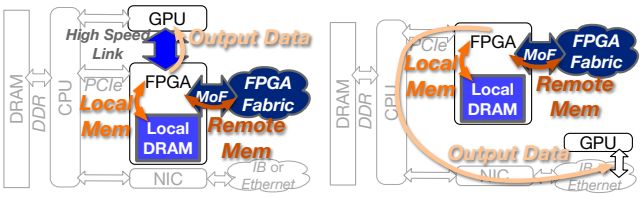
(a) comm-opt.tc (tightly coupled) (b) comm-opt.decp (decoupled)

Figure 11: *comm-opt*: FaaS architecture with customized inter-FPGA fabric.

To implement *comm-opt.decp* and *comm-opt.tc* by adapting the architecture proposed in Section 4, we use PCIe for *CMD/Data IO*, and PCIe-connected host DRAM as *Local Mem IO*, but dedicated FPGA fabric as *Remote Mem IO*, as summarized in Table 8. We adopt MoF without modification, which directly connects to FPGA fabric QSFP PHY, running the customized low latency protocol described in Section 4.3. We then tune the AxE with a two-core configuration, according to calculation using Equation 3, as a result of the latency reduction of the remote memory access.

6.5 FaaS w/ FPGA local DRAM (mem-opt)

After addressing the communication problems, the bottleneck shifts to the DRAM memory bandwidth. We propose *mem-opt* architectures to further unleash FaaS's performance, as shown in Figure 12. In *mem-opt*, FPGA owns its local DDR memory to store the graph data. Figure 12 (a) shows the tightly coupled case, in which FPGA and GPU are in the same server. Note that in *mem-opt.tc*, FPGA and GPU are connected with high-speed GPU links, instead of PCIe. Figure 12 (b) shows the decoupled case.



(a) mem-opt.tc (tightly coupled) (b) mem-opt.decp (decoupled)

Figure 12: *mem-opt*: FaaS architecture with FPGA local DRAM.

Based on the architecture proposed in Section 4, we implement *mem-opt* by (1) adopting PCIe for *CMD IO* but GPU high-speed link (e.g., NVlink-like solutions) for *DATA IO* in the tightly coupled case, (2) building FPGA on-board DDR4 DRAM as *Local Mem IO*, and (3) using dedicated FPGA fabric as *Remote Mem IO*, as summarized in Table 8. MoF is tailored the same way as shown in the *comm-opt* case. For AxE tuning, we implement two AxE cores for *mem-opt.decp*, according to Equation 3. We then need ten AxE cores for *mem-opt.tc*, to fully utilize the increased effective local memory bandwidth and FPGA-GPU bandwidth.

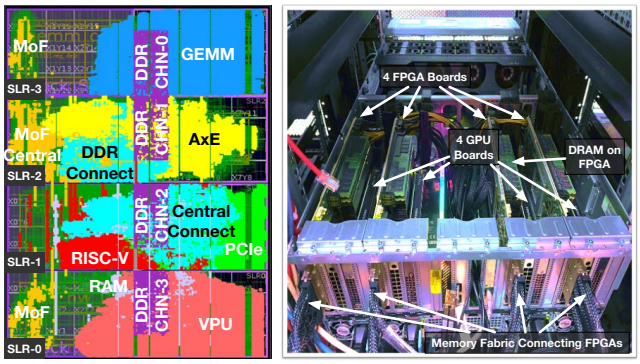
7 EVALUATION

In this section, we first demonstrate an FPGA proof-of-concept (PoC) implementation. With the measured results from the PoC, we then project the FaaS DSE and quantitively demonstrate their pros and cons.

7.1 A Proof-of-Concept (PoC) Implementation

The goals of the PoC system are: (1) proofing that the architecture design is implementable, functionality correct, and capable for scaling; (2) setting an anchor point for later performance and resource projection; (3) providing an application-level demonstration and a platform for system and software development. Note that the PoC does not target at demonstrating advanced performance or large capacity since it is the halfway to the products (e.g., FaaS). To explain the connection between the PoC and the targeting FaaS architectures, Table 9 summarizes the PoC configuration in the format as Table 8 for comparison, showing that how PoC helps to validate the FaaS architecture.

As shown in Figure 13 (right), the PoC adopts a 4U server. We use four Bittware VV8 [23] cards, each of which has a Xilinx VU13P [65], 16 lane PCIe-Gen3, 4×128GB DDR4, and 4×QSFP-DD cages, each supporting 2×100GbE interface. The four FPGA boards are connected P2P with Direct Attach Copper (DAC) fabrics. Four Nvidia T4 [56] cards are added for tightly coupled heterogeneous NN computing, connecting to FPGAs through PCIe P2P. On each FPGA, the PoC implementation configuration is shown in Table 10, the resource utilization is shown in Table 11, and the post-synthesis layout for the 4-SLR FPGA is shown in Figure 13 (left). The heterogeneous system is fully integrated with AliGraph [28, 87] framework. It is capable of demonstrating end-to-end industrial LSD-GNN applications.



(a) FPGA post-syn layout (b) 4-card FPGA testbed

Figure 13: FPGA proof-of-concept demonstration.

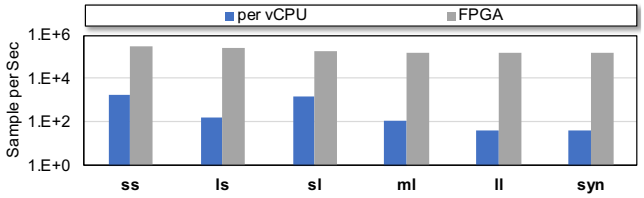


Figure 14: Measurement on the demonstration system.

Figure 14 shows the GNN sampling rate (sample/second) measurement results on the PoC, with the benchmarks described in Table 2. The results are compared with CPU-based AliGraph software solution, whose performance is normalized by per vCPU. Note that the server described in Table 2 is a software concept similar to a service, instead of a physical server. The vCPU each server actual consumes is dynamic and application dependent. We consider the average vCPU one server consumes over the whole sampling process and normalizes the sampling throughput to per-vCPU for a fair and clear comparison. We conclude that, on average, one PoC FPGA board can provide 894 vCPU's sampling capability. This performance can be further improved with architecture tuning and optimizations during the transition from PoC to products.

7.2 Evaluation Setup: from PoC measurement to FaaS DSE

To close the gap between the PoC implementation (Table 9) and the FaaS architecture DSE (Table 8), we build an in-house performance analytical model to capture FPGA hardware behavior, memory access, and inter-FPGA communications. The simulator is capable of exploring performance given a set of architecture configurations (e.g., Table 10). Figure 15 validates the simulator with the measurement results from the PoC implementation. In this figure, we validate various configurations, including (1) Number of AxE cores shown in x-axis, varies from 1, 2, and 4-core; (2) Memory configurations shown in legends, denoted as *PCIe* (i.e., PCIe connected host memory), *1-chn, 2-chn, 4-chn* (i.e., 1/2/4 channel FPGA local DDR4); (3) Number of FPGA nodes shown in legends, denoted as *1n*, *4n* (i.e., single FPGA and 4-node FPGAs). We have the measurement results with these configurations on the PoC presented by the dots, while the modeling number is presented by lines. It validates the effectiveness of the simulator by showing 0.974% error. In addition, to show the capability of the model, we present the modeling result without PCIe output limitation with the bar and the right y-axis. Because most of the configurations and measurements supported by the PoC are eventually bottlenecked by the PCIe bandwidth for results output, we have to rely on the simulator for such design cases without PCIe bottleneck, such as the case in *mem-opt.tc*.

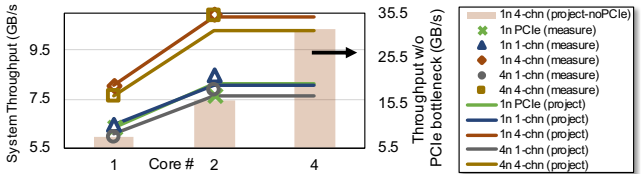


Figure 15: Validating the performance model with PoC measurement.

We also build a FaaS instance cost model for various instance configurations (including FPGA and GPU) by collecting available data from Alibaba Cloud price calculator [5]. We construct a linear regression model with number of vCPU, DRAM capacity, and number of FPGA and GPU card. We validate the cost model with the price calculator, as shown in Figure 16. The table under the figure shows the instance configurations with its product ID in Alibaba Cloud. The blue line with right y-axis shows the errors. Despite of the error in the *ecs-ram-e* (906GB memory) instance, which is the advanced instance with the largest memory that leads to extra cost under-estimated by our linear cost model, it shows our cost model is generally effective.

We summarize the setup for the following FaaS evaluation.

- **Benchmarks**: GNN sampling (excluding NN part) described in Table 3 on graph data (*ss*, *ls*, *sl*, *ml*, *ll*, *syn*) described in Table 2. The performance metric is samples (i.e., batches) per second.

	FPGA-FPGA Connection	Local Mem Access	Remote Mem Access	FPGA-GPU Connection
Proof-of-concept implementation	**[opt]** Dedicated Customized Fabric (MoF, **100GB/s**)	PCIe→HostMem (16GB/s) or **[opt]** FPGA local DRAM (**100GB/s**)	**[opt]** Dedicated Customized Fabric (MoF, **100GB/s**)	in-server PCIe P2P (16GB/s)

Table 9: Comparing the PoC implemented configurations with FaaS architectures in Table 8.

AxE (250MHz)	dual-core, 4-channel DDR4-1600 (12.8GB/s×4, 512GB) as local MemIO, MoF as remote MemIO, and PCIe as CmdIO
MoF (250MHz, 322MHz PHY)	3x QSFP-DD (200Gb/s×3) per card, 4-card P2P connection, 200Gb/s×6×2 bidirection bandwidth for whole system
RISC-V (100MHz)	E906 [59] with 32KB I-cache and D-cache, 512KB I-MEM and D-MEM, and JTAG interface
Subsystem (250MHz)	8MB×2 bank shared memory, 8KB buffer, 32×32-bit CSR, 1MB MMU, and PCIe-gen3x16 with Xilinx QDMA
Interconnect (250MHz)	hierarchical AXI with Smart-Connect, with dedicated AXI for AxE and 4-channel DDR for speed consideration

Table 10: PoC configuration.

CLBs (K)	LUTs (K)	CLB Reg (K)	BRAM (Mb)	URAM (Mb)	DSP
216	1728	3456	94.5	360	12288
60.53%	35.07%	22.48%	39.29%	40.00%	12.50%

Table 11: Resource utilization of VU13P FPGA.

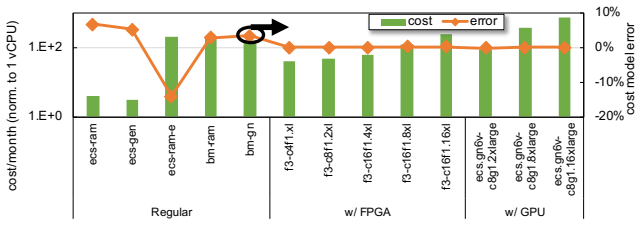


Figure 16: Validating the cost model with public available data.

	vCPU	Memory	FPGA	NIC	MoF
-small	2	8GB	1 chip	10Gb	100Gb (if applicable)
-medium	2	384GB	1 chip	20Gb	200Gb (if applicable)
-large	2	512GB	2 chips	50Gb	800Gb (if applicable)

Table 12: FaaS configurations.

- **FaaS Architectures**: As described in Section 6.1, tightly coupled same-instance FPGA/GPU *base.tc*, *cost-opt.tc*, *comm-opt.tc*, *mem-opt.tc*, and decoupled FPGA/GPU *base.decp*, *cost-opt.decp*, *comm-opt.decp*, *mem-opt.decp*.
- **Instance Configurations**: As described in Section 3, we use three types of instance configurations: *small (s)*, *medium (m)*, and *large (l)*, as shown in Table 12.
- **GPU considerations**: We assume the application requires one V100 GPU for every 12GB/s sampling throughput (75% of the PCIe bandwidth).

7.3 Limitation of This Work

Our work has three following modeling and evaluation limitations.
Limitation-1: In terms of performance we only consider the sampling portion of the GNN workload. The reason for this is that the NN portion is quite diverse and highly GPU dependent. As shown in Figure 3, the NN portion accounts for 36% to 11% of the application time. Indeed, when the sampling portion is substantially speeded up, the NN portion will now contribute to relatively larger portion. For example, in some case if we replace all CPU instances with same amount of FPGA instances but does not change GPU instance configurations, sampling would then take up only 1% to 2% of the new execution time, with NN time dominating the entire application. A comprehensive treatment of full application speedups considering all the system effects is future work.
Limitation-2: We make the simplifying assumption to the GPU dollar cost that the application requires one V100 GPU for every 12GB/s sampling throughput (75% of the PCIe bandwidth). This simplification affects Figures 17 through 21. As of the application requires more GPU instance in a very deep and complex NN model scenario, e.g., ten V100 for every 12GB/s sampling throughput, our performance per dollar benefit shown in Figure 21 could be offset from 12.58× to 1.48×.
Limitation-3: We do not model the FPGA and interconnect dollar costs at sufficient detail to determine the cost difference between the eight FaaS configurations – as a result, with our simple dollar cost model, all eight FaaS configurations cost the same. Future work can refine this issue. This simplification affects Figures 18, 21, and 20.

7.4 FaaS Architecture Exploration Evaluation

Result overview: Figure 17 and 19 shows the result of eight FaaS solutions introduced in Section 6 on six graph dataset introduced in Table 2. For each FaaS solution, there are three instance configurations considered, introduced in Table 12. Figure 17 presents the GNN sampling throughput/instance, and Figure 19 shows the normalized performance/dollar (to CPU geomean). Figure 18 and 21 show the geomean of Figure 17 and 19 over six data sets, respectively. Figure 20 shows the normalized (to *ss* CPU cost) minimal CPU and *FaaS.base* service cost to carry out and run applications on the six graph dataset. It serves as a connection between performance in Figure 17 and performance/dollar in Figure 18. As mentioned in the Section 7.3, Figure 17 to 21 only consider GNN's sampling part throughput, not the NN part. Figure 18 to 21 assume one V100 GPU is required for every 12GB/s GNN sample throughput.
Compare *FaaS.base* with CPU baseline: We find that all FaaS solutions, even with the off-the-shelf *FaaS.base*, significantly outperforms the CPU-based baseline solution. The performance of a FPGA in the decoupled FPGA and GPU cases (*decp*) is equivalent to 67× vCPU, and that of the tightly coupled cases (*tc*) is 129.6×. The performance/dollar is also improved by 2.47× and 4.11× in *decp* and *tc*, respectively. This demonstrates the major advantage

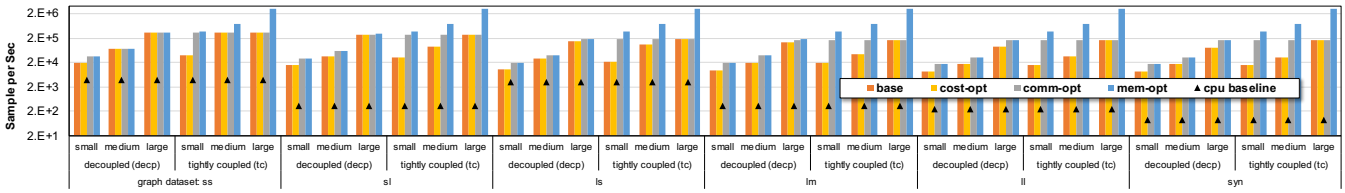


Figure 17: GNN sampling performance/instance of eight FaaS architectures (*base*, *cost-opt*, *comm-opt*, *mem-opt*, with decoupled *.decp* or tightly coupled *.tc* option) on six graph dataset (*ss*, *ls*, *sl*, *ml*, *ll*, *syn*) with three instance configurations (*small*, *medium*, *large*).

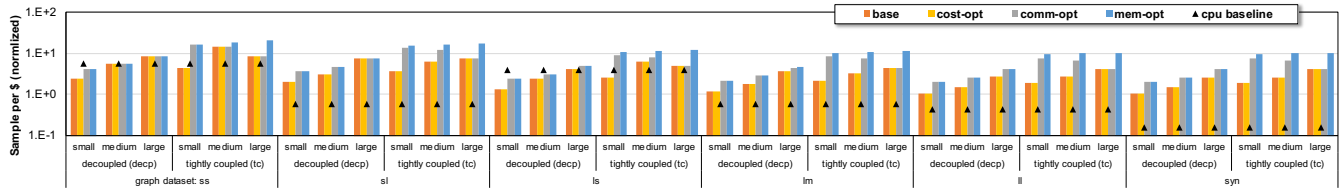


Figure 18: Normalized performance/dollar of GNN sampling from eight FaaS architectures (*base*, *cost-opt*, *comm-opt*, *mem-opt*, with *.decp* or *.tc*) on six graph dataset (*ss*, *ls*, etc) with three instance configurations (*small*, *medium*, *large*).

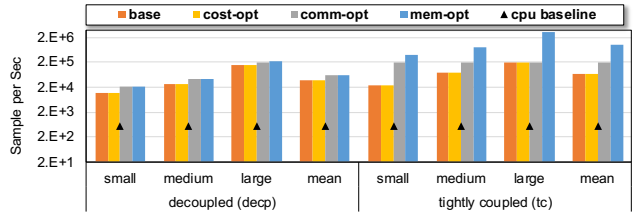


Figure 19: Average GNN sampling performance/instance of eight FaaS architectures (*base.decp, cost-opt.decp, comm-opt.decp, mem-opt.decp, base.tc, cost-opt.tc, comm-opt.tc, mem-opt.tc*) with *small*/*medium*/*large* instance configurations (geomean over six graph dataset in Figure 17).

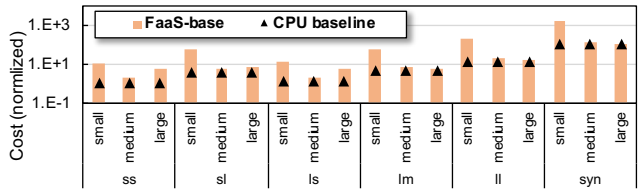


Figure 20: Minimal cost (cloud service cost, not building server cost) of the FaaS.base with *small*/*medium*/*large* instance configurations on six graph dataset (*ss*, *ls*, etc).

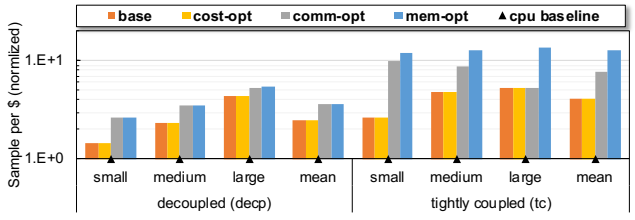


Figure 21: Average normalized performance/dollar of GNN sampling from eight FaaS architectures (*base.decp, cost-opt.decp, etc*) with *small*/*medium*/*large* instance configurations (geomean over six graph dataset in Figure 18).

of our architecture. The performance gain comes from addressing *Challenge-1* and *Challenge-2* summarized in Section 3.5, by improving memory access and communication efficiency. The cost down comes from offloading the CPU unfriendly workload to the more efficient yet affordable FPGA resource.

Figure 20 shows the comparison between FaaS and CPU from cost perspective, providing a guideline for users to choose their instance accordingly. For example, in the *lm* dataset with *small* instance, we need at least 49 CPU or *FaaS.base* instances to carry out the graph at the cost of 5.44 or 69.81. If users do not care about performance at all, CPU is still the best solution with lowest cost. However, the 49 *FaaS.base* instances give 28.8× sampling throughput than the CPU instances though more expensive (shown in Figure 17). Therefore, if users are in the demand of high performance, *FaaS.base* would be the choice. Also, if users care about performance/dollar, *FaaS.base* is $\frac{28.8/69.81}{1/5.44} = 2.25\times$ better (also shown in Figure 18). That is for the same cost, users can get 2.25× improvement in performance. The further advantage of FaaS is that, since it scales better, one could build a more expensive FaaS system, that provides even higher performance, whereas scaling up performance with a pure CPU system is suboptimal because of the memory access and inter-node communication challenges.

***FaaS.base* scalability:** Comparing six graph datasets (*ss, sl, etc*), we find that FaaS performance scales well with the size of the graph data. Referring to Figure 2(a) that shows the memory footprint of the six graph data, the graph with smaller size (i.e., *ss, ls*) trends to have a weak performance improvement over vCPU (29% and 18% of the geomean, in Figure 17's *FaaS.base.decp* case). This weak performance improvement leads to worse-than-CPU performance per dollar in *ss, ls* of Figure 18. However, as graph data get bigger when the CPU baseline faces the challenges of inefficient scaling for larger graphs, our customized FPGA design shows its benefit. In most industry use cases, graph is much larger than *ss, ls* and eligible for taking FaaS's advantage.

Comparing three instance configurations (*small, medium, large*), we find that FaaS performance scales well with the instance resource

assigned. For example, in the *FaaS.base.decp* case, *medium/large* instance is 2.4× and 14.0× faster than the *small* instance. That is to say a larger and more integrated instance would always be beneficial, if cost allows.

***FaaS.cost-opt*'s pros and cons:** We find that the *FaaS.cost-opt* architecture (described in Section 6.3) looks unattractive to cloud users since it does not provide better performance and performance/dollar than *FaaS.base*. Such architecture uses an on-FPGA NIC, instead of a PCIe standalone NIC, for communication and remote memory access. The reason for the unchanged performance is that the on-FPGA NIC does not provide extra bandwidth. The FaaS does not benefit from the lower latency either, since it is already not the system bottleneck. The reason for the unchanged performance/dollar is that we are considering the cloud user-side retail service price, instead of the cost of building/operating the infrastructure (though it is possible if a cloud provider potentially reduces the FaaS service price according to the saving on infrastructures).

However, we envision that such architecture would be preferred by cloud providers, for two reasons. First is the cost saving on building and operating the infrastructure discussed in Section 6.3. Second, we also see a datacenter technology trend of customized NIC (e.g., MoC [3] in Alibaba Cloud and Nitro [8] in AWS). Such trend (an existing FPGA customized NIC) would be in favor of implementing the *FaaS.cost-opt* architecture.

***FaaS.comm-opt*'s pros and cons:** We find *FaaS.comm-opt* architecture (Section 6.4) shows significant improvement. Such architecture targets at addressing *Challenge-2* by adapting a customized faster and wider FPGA-to-FPGA fabric for communication and remote memory access. Compared with *FaaS.base*, *FaaS.comm-opt.decp* and *FaaS.comm-opt.tc* provides 1.6× and 2.9× extra performance, respectively. The average performance per dollar demonstrates a similar improvement (1.5× and 1.9×).

FaaS.comm-opt is attractive for both the cloud providers and users, due to the significant performance improvement. However, it would be the perfect solution if we do not have the following practical concerns. Although Microsoft [12] has implemented a similar FPGA fabric, most of the data centers are reluctant to build and maintain a customized, non-standard inter-node fabric, due to the difficulty in, e.g., reliability. In this sense, we envision that next-generation communication infrastructures such as CXL [16] may bridge this gap in the near future.

***FaaS.mem-opt*'s pros and cons:** We find that *FaaS.mem-opt.tc* (Section 6.5) further improves the performance and performance per dollar, demonstrating 3.0× than that of *FaaS.base.tc*. After *FaaS.comm-opt* solves the remote memory bandwidth, the bottleneck is then shifted to the PCIe-connected DRAM bandwidth. *FaaS.mem-opt* addresses this problem by adding the FPGA local DRAM, with which the FPGA can embrace the multi-channel DDR4 bandwidth instead of the narrow PCIe host memory bandwidth. Note that *FaaS.mem-opt.decp* does not provide any extra performance since the performance is bottlenecked by PCIe result data transferring from FPGA to NIC and then the GPU on other servers.

Though with the best performance, *FaaS.comm-opt* may not be a preferable design choice. The on-FPGA DRAM cost is non-trivial for building the infrastructure. Even worse, unlike host memory, such on-FPGA DRAM might not be fully utilized by other general-purpose applications. That said, we envision that *FaaS.comm-opt* architecture might be more feasible when integrated to GPU/Accelerators chips, in which cases the local memory can also serve as a general-purpose extended device memory to GPU/Accelerators.

Tightly coupled (*tc*) v.s. Decoupled (*decp*): We find that heterogeneous cloud instance (*tc*) is generally more beneficial and necessary for GNN. As shown in Section 6.1, unlike conventional decoupled (*decp*) architecture, in which all-FPGA servers and all-GPU servers are separated by network, tightly coupled (*tc*) is one single heterogeneous server carrying both cards. For example, the *FaaS.cost-opt.tc* shows 1.9× better than performance *FaaS.cost-opt.decp*. This is because *tc* architecture no longer needs to send results through the already busy NIC, which is equivalent to a larger NIC bandwidth benefiting the system communication. Considering the *Challenge-2* of the communication bandwidth limitation, the performance and performance/dollar are improved.

We also conclude that the tightly coupled architecture is preferable when FaaS performance gets further optimizations. The benefit of *tc* over *decp* increases from 1.9× in *FaaS.cost-opt* to 3.5× in *FaaS.comm-opt* and 16.6× in *FaaS.mem-opt*. The benefit magnification is that as the performance keeps improving, the system bottleneck has shifted from communication and memory to the result data output (from FPGA to GPU). Therefore, for the best performance *FaaS.mem-opt*, we must have the *tc* option to fully unleash the performance.

8 RELATED WORK

GNN hardware: The researches on customized hardware for SM-GNN have been well established [6, 14, 25, 39, 40, 42, 44, 45, 47, 69, 76–78, 84, 85]. Among them, Zhang *et al.* [78], GraphACT [76], AWB [25], and BoostGCN [77] are FPGA-based solutions. However, these SM-GNN's solutions do not work for LSD-GNN due to the shift of hardware bottleneck to remote access latency and bandwidth efficiency, as explained in Section 3.

In addition, there are three other limitations with previous SM-GNN hardware works. First, almost all works target at inference (except GraphACT [76] and GNNsampler [47]). Our work is designed for both inference and training. Second, most of the works including HyGCN [69], Engn [45], AWB [25], Zhang *et al.* [78], GCNAX [44], GraphACT [76], and BoostGCN [77], are weak at programmability and only support for a few certain GNN models (e.g., GCN). Our work provide flexibility compatible with well-developed framework (AliGraph). Third, not enough attention has been paid to memory access. Some of the works like HyGCN [69], AWB [25], and VersaGNN [60] making effort on sparse matrix multiplication, but it is not the workflow for LSD-GNN. Others such as GCNAX [44], BoostGCN [77], Rubik [14], GraphACT [76], GNNSampler [47], and Grip [39, 40] optimizing data reuse are not applicable to LSD-GNN either, since the chance to find reuse within 512-node mini-batch compared with 10+ billion total nodes is extremely low. Huang *et al.* [34] works on a similar problem like this paper, but under a different disaggregated memory pool context.

Sparse tensor hardware: Architecture innovations from sparse tensor accelerators [32, 58, 62, 81] can be adopted to solve the SM-GNN problems, since the full-batch training of SM-GNN can be well processed with sparseMM on a single machine. LSD-GNN, on the

other hand, uses a mini-batch training scheme (see Section 2.2) and hence has a sampling step, where part of the sparse adjacent matrix multiplies the dense attribute matrix. Such operation is carried out as gather-like memory access instead of sparseMM. Therefore, LSD-GNN demands hardware-accelerated gathering operators for the distributed system rather than the highly optimized sparseMM in a single machine.

GNN system optimization: Software and system optimization for GNN has been well studied. In addition to framework-level effort [33, 51, 63], researchers also have paid attention to distributed system optimization for LSD-GNN [37, 66, 82, 83, 87]. These framework level innovations, such as caching and partition in AliGraph [87], are orthogonal to our underlying hardware evolutions. There are also researches on GPU software optimizations. For example, ZeroCopy [53] leverages both CPU and GPU for the sampling stage on SM-GNN. GNNAdviser [68] and fusedGNN [15] optimize GPU kernel for sparse operators. However, they only cover SM-GNN instead of the multi-node LSD-GNN we target in this paper.

Graph analysis hardware: There are also works on customized hardware for graph analytics, including ASIC [13, 17, 30, 46, 57, 70, 71], FPGA/GPU [18, 19, 50, 67, 73], and processing customizations [10, 72]. However, these works are not well associated with GNN applications nor limited on focusing the single machine cases. Hence, they cannot be adopted to solve the LSD-GNN problems.

9 DISCUSSION BEYOND FPGA

Integrated CPU/GPU Architecture: Similar to *mem-opt.tc* solution, we can replace the FPGA with a processor and connect it with GPU through GPU's high speed link. NVidia Grace [27] presents such a solution with 144 ARM cores, LPDDR5, and 900GB/s NVlink. Intel Sapphire Rapids [36] integrates AI engine inside the processor. While processor-based architectures provide a more general solution, we argue that they are suboptimal for LSD-GNN. First, CPUs are inefficient for sampling compared with the FPGA solution (recall that one FPGA provides 894 vCPU performance). Second, Grace does not use DIMM-based DRAM, which limits the capacity scalability.

Integrated NIC/GPU Architecture: Similar to *cost-opt.tc* solution, data processing unit (DPU, e.g., NVidia Bluefield [11]) provides an alternative solution by providing general processing units on NIC, which can be potentially used for sampling. We argue that DPU solutions are also suboptimal for LSD-GNN, limited by the processing capability. Hence they cannot fully utilize the bandwidth. For example, Bluefield [11] provides 300 CPU core, but the customized FPGA hardware provides performance as much as 894 vCPUs.

ASIC solution: We see weak motivation for a standalone customized ASIC chip solution to replace the FPGA solution. Performance wise, all standalone sampling chip solutions have a performance upper-bound (i.e., the GPU data input bandwidth). Although an ASIC can provide larger throughput than the FPGA solution, both of them will hit the upper-bound. Cost-wise, ASIC requires a significant NRE cost, but there is not enough volume and demand to even it out. Therefore, FPGA is a more reasonable solution after ROI is calculated. That said, we envision integrating the AxE engine into the GPU/AI hardware would be a promising solution, for example, as an extension of NVidia H100 TMA [29] or Google TPU 4D tensor DMA [38].

10 CONCLUSION

In this paper, we bring up the new hardware challenges from Large Scale Distributed GNN (LSD-GNN), and point out its difference from previous well-studied SM-GNN. We then propose a novel architecture to cope with the challenges of large graph storage, long latency, and underutilized bandwidth. The proposed architecture is examined by a four-card heterogeneous proof-of-concept FPGA system with full software integration, demonstrating that one FPGA can provide 894 vCPU's sampling capability. In order to make the solution profitable, programmable, and scalable, we further propose to deploy the architecture with FPGA-as-a-Service (FaaS) platform. We explore various FaaS architecture configurations and conclude that (1) Off-the-shelf *FaaS.base* is capable of providing GNN service, with 2.47× performance per dollar improvement over vCPU solutions; (2) Integrating NIC on the FPGA *FaaS.cost-opt* is attractive not for performance but for potential cost down on the cloud provider side; (3) Further addressing the communication bandwidth by adding dedicated FPGA interconnection fabric (*FaaS.comm-opt*) improves the benefit to 3.65×; (4) Building local DRAM to FPGA *FaaS.mem-opt* and heterogeneous servers with both FPGA and GPU (*tc*) further removes the memory bottleneck and PCIe bottleneck. It shows 12.58× improvement, though the cost of building such infrastructure is non-trivial;

ACKNOWLEDGMENTS

We thank anonymous reviewers for their valuable feedbacks and help to improving this paper. We appreciate a wide variety of Alibaba internal collaborators, especially T-Head IoT team for RISC-V support, Data Analytics and Intelligence Lab and AI Engine team for applications and profilings, Platform of AI team for AliGraph support, and X-Dragon team for FaaS support.

REFERENCES

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2020. Computing graph neural networks: A survey from algorithms to accelerators. *arXiv preprint arXiv:2010.00130* (2020).
[2] AliCloud F3 2022. *Alibaba Cloud Elastic Compute Service, Compute optimized type family with FPGA.* https://www.alibabacloud.com/help/doc-detail/108504.html
[3] AliCloud MoC 2022. Alibaba Cloud X-Dragon NiC. https://www.alibabacloud.com/blog/introducing-the-sixth-generation-of-alibaba-clouds-elastic-compute-service_595716
[4] AliCloud PAI 2022. Alibaba Cloud's Machine Learning Platform for AI (PAI). https://www.alibabacloud.com/product/machine-learning
[5] AliCloud Price 2022. Alibaba Cloud Elastic Compute Service, Price Calculator. https://www.alibabacloud.com/pricing-calculator
[6] Adam Auten, Matthew Tomei, and Rakesh Kumar. 2020. Hardware acceleration of graph neural networks. In *Proceedings of the 2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[7] AWS EC2 2022. Amazon EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/
[8] AWS Nitro 2022. AWS Nitro System. https://aws.amazon.com/ec2/nitro/
[9] Azure FPGA 2022. Microsoft Azure FPGA Inference. https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service
[10] Abanti Basak, Shuangchen Li, Xing Hu, Sang Min Oh, Xinfeng Xie, Li Zhao, Xiaowei Jiang, and Yuan Xie. 2019. Analysis and optimization of the memory hierarchy for graph processing workloads. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 373–386.
[11] Bluefield 2022. BlueField-3, the most powerful software-defined, hardware-accelerated data center infrastructure on a chip. https://www.nvidia.com/en-us/networking/products/data-processing-unit/

[12] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[13] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 581–594.

[14] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. 2021. Rubik: A Hierarchical Architecture for Efficient Graph Neural Network Training. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[15] Zhaodong Chen, Mingyu Yan, Maohua Zhu, Lei Deng, Guoqi Li, Shuangchen Li, and Yuan Xie. 2020. fuseGNN: accelerating graph convolutional neural network training on GPGPU. In *Proceedings of the 2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[16] CXL 2022. Compute Express Link 2.0: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/

[17] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 595–608.

[18] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. 2016. FPGP: Graph processing framework on FPGA a case study of breadth-first search. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 105–110.

[19] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. 2017. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 217–226.

[20] Extended Version 2022. Hyperscale FPGA-As-A-Service Architecture for Large-Scale Distributed Graph Neural Network (extended version). https://shuangchenli.github.io/isca22tr.pdf

[21] FaaS Benefit 2022. Alibaba Cloud FPGA-based ECS Instances Scenarios. https://partners-intl.aliyun.com/help/doc-detail/163848.htm

[22] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[23] FPGA Board 2022. Bittware XUP-VV8. https://www.bittware.com/fpga/xup-vv8/

[24] Alberto Garcia Duran and Mathias Niepert. 2017. Learning graph representations with embedding propagation. *Advances in neural information processing systems* 30 (2017), 5119–5130.

[25] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, et al. 2020. AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing. In *Proceedings of the 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 922–936.

[26] GENZ 2022. Gen-Z: An open systems Interconnect designed to provide memory-semantic access to data and devices via direct-attached, switched or fabric topologies. https://genzconsortium.org/

[27] GRACE 2022. NVIDIA GRACE CPU. https://www.nvidia.com/en-us/data-center/grace-cpu/

[28] GraphLrean 2022. Graph-Learn (a.k.a. AliGraph): An Industrial Graph Neural Network Framework. https://github.com/alibaba/graph-learn

[29] H100 2022. NVIDIA Hopper Architecture In-Depth. https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/

[30] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[31] William L Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. 1025–1035.

[32] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 319–333.

[33] Yuwei Hu, Zihao Ye, Minjie Wang, Jiali Yu, Da Zheng, Mu Li, Zheng Zhang, Zhiru Zhang, and Yida Wang. 2020. Featgraph: A flexible and efficient backend for graph neural network systems. In *Proceedings of the SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[34] Linyong Huang, Zhe Zhang, Shuangchen Li, Dimin Niu, Yijin Guan, Hongzhong Zheng, and Yuan Xie. 2022. Practical Near-Data-Processing Architecture for Large-Scale Distributed Graph Neural Network. In *2022 IEEE Access*. IEEE.

[35] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. 2013. Learning deep structured semantic models for web search using clickthrough data. In *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. 2333–2338.

[36] Intel SPR 2022. Golden Cove - Microarchitectures - Intel. https://en.wikichip.org/wiki/intel/microarchitectures/golden_cove

[37] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the accuracy, scalability, and performance of graph neural networks with roc. *Proceedings of Machine Learning and Systems* 2 (2020), 187–198.

[38] Norman P Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, et al. 2021. Ten lessons from three generations shaped google's tpuv4i: Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1–14.

[39] Kevin Kiningham, Philip Levis, and Christopher Ré. 2020. GReTA: Hardware Optimized Graph Processing for GNNs. In *Proceedings of the Workshop on Resource-Constrained Machine Learning (ReCoML 2020)*.

[40] Kevin Kiningham, Christopher Re, and Philip Levis. 2020. GRIP: a graph neural network accelerator architecture. *arXiv preprint arXiv:2007.13828* (2020).

[41] Yunsup Lee, Andrew Waterman, Henry Cook, Brian Zimmer, Ben Keller, Alberto Puggelli, Jaehwa Kwak, Ruzica Jevtic, Stevo Bailey, Milovan Blagojevic, et al. 2016. An agile approach to building RISC-V microprocessors. *IEEE Micro* 36, 2 (2016), 8–20.

[42] Cangyuan Li, Ying Wang, Cheng Liu, Shengwen Liang, Huawei Li, and Xiaowei Li. 2021. GLIST: Towards In-Storage Graph Learning. In *2021 {USENIX} Annual Technical Conference ({USENIX} {ATC} 21)*. 225–238.

[43] Houyi Li, Yongchao Liu, Yongyong Li, Bin Huang, Peng Zhang, Guowei Zhang, Xintan Zeng, Kefeng Deng, Wenguang Chen, and Changhua He. 2021. GraphTheta: A Distributed Graph Neural Network Learning System With Flexible Training Strategy. *arXiv preprint arXiv:2104.10569* (2021).

[44] Jiajun Li, Ahmed Louri, Avinash Karanth, and Razvan Bunescu. 2021. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 775–788.

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

[46] Jilan Lin, Shuangchen Li, Yufei Ding, and Yuan Xie. 2021. Overcoming the Memory Hierarchy Inefficiencies in Graph Processing Applications. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 1–9.

[47] Xin Liu, Mingyu Yan, Shuhan Song, Zhengyang Lv, Wenming Li, Guangyu Sun, Xiaochun Ye, and Dongrui Fan. 2021. GNNSampler: Bridging the Gap between Sampling Algorithms of GNN and Hardware. *arXiv preprint arXiv:2108.11571* (2021).

[48] Ziqi Liu, Chaochao Chen, Xinxing Yang, Jun Zhou, Xiaolong Li, and Le Song. 2018. Heterogeneous graph neural networks for malicious account detection. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. 2077–2085.

[49] Yun-Chen Lo, Yu-Chun Kuo, Yun-Sheng Chang, Jian-Hao Huang, Jun-Shen Wu, Wen-Chien Ting, Tai-Hsing Wen, and Ren-Shuo Liu. 2019. Physically Tightly Coupled, Logically Loosely Coupled, Near-Memory BNN Accelerator (PTLL-BNN). In *ESSCIRC 2019-IEEE 45th European Solid State Circuits Conference (ESSCIRC)*. IEEE, 241–244.

[50] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated graph processing on a single machine with balanced replication. In *2017 {USENIX} Annual Technical Conference ({USENIX} {ATC} 17)*. 195–207.

[51] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: parallel deep neural network computation on large graphs. In *Proceedings of 2019 {USENIX} Annual Technical Conference ({USENIX} {ATC} 19)*. 443–458.

[52] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A hardware accelerator for tracing garbage collection. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 138–151.

[53] Seung Won Min, Kun Wu, Sitao Huang, Mert Hidayetoğlu, Jinjun Xiong, Eiman Ebrahimi, Deming Chen, and Wen-mei Hwu. 2021. Large Graph Convolutional Network Training with GPU-Oriented Data Communication Architecture. *arXiv preprint arXiv:2103.03330* (2021).

[54] MVAPICH 2022. MVAPICH: MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. https://mvapich.cse.ohio-state.edu/benchmarks/

[55] Dimin Niu, Shuangchen Li, Yuhao Wang, Wei Han, Zhe Zhang, Yijin Guan, Tianchan Guan, Fei Sun, Fei Xue, Lide Duan, Yuanwei Fang, Hongzhong Zheng, Xiping Jiang, Song Wang, Fengguo Zuo, Yubing Wang, Bing Yu, Qiwei Ren, and Yuan Xie. 2022. 184QPS/Watt, 64Mbit/mm2 3D Logic-to-DRAM Hybrid Bonding with Process-Near-Memory Engine for Recommendation System. In *2022 International Solid-State Circuits Conference (ISSCC)*. IEEE.

[56] NVT4 2022. NVIDIA T4. https://www.nvidia.com/en-us/data-center/tesla-t4/

[57] Muhammet Mustafa Ozdal, Serif Yesil, Taemin Kim, Andrey Ayupov, John Greth, Steven Burns, and Ozcan Ozturk. 2016. Energy efficient architecture for graph analytics accelerators. *ACM SIGARCH Computer Architecture News* 44, 3 (2016),

166–177.

[58] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 724–736.

[59] RISC-V 2022. T-head E906 RISC-V core. https://www.t-head.cn/product/E906

[60] Feng Shi, Ahren Yiqiao Jin, and Song-Chun Zhu. 2021. VersaGNN: a Versatile accelerator for Graph neural networks. *arXiv preprint arXiv:2105.01280* (2021).

[61] SmartConnect 2022. Xilinx LogiCORE IP AXI SmartConnect. https://www.xilinx.com/products/intellectual-property/smartconnect.html

[62] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 766–780.

[63] Chao Tian, Lingxiao Ma, Zhi Yang, and Yafei Dai. 2020. Pcgcn: Partition-centric processing for accelerating graph convolutional network. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 936–945.

[64] VPU 2022. An opensource VPU implementation. https://github.com/alibaba/vector-accelerating-unit

[65] VU13P 2022. Xilinx Virtex UltraScale+ VU13P devices. https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html

[66] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. 2021. FlexGraph: a flexible and efficient distributed framework for GNN training. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 67–82.

[67] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

[68] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2020. Gnnadvisor: An efficient runtime system for gnn acceleration on gpus. In *Proceedings of 2020 USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[69] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. Hygcn: A gcn accelerator with hybrid architecture. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 15–29.

[70] Mingyu Yan, Xing Hu, Shuangchen Li, Itir Akgun, Han Li, Xin Ma, Lei Deng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, et al. 2019. Balancing memory accesses for energy-efficient graph analytics accelerators. In *2019 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 1–6.

[71] Mingyu Yan, Xing Hu, Shuangchen Li, Abanti Basak, Han Li, Xin Ma, Itir Akgun, Yujing Feng, Peng Gu, Lei Deng, et al. 2019. Alleviating irregularity in graph analytics acceleration: a hardware/software co-design approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 615–628.

[72] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2021. SpZip: Architectural Support for Effective Data Compression In Irregular Applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1069–1082.

[73] Yifan Yang, Zhaoshi Li, Yangdong Deng, Zhiwei Liu, Shouyi Yin, Shaojun Wei, and Leibo Liu. 2020. GraphABCD: Scaling out graph analytics with asynchronous block coordinate descent. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 419–432.

[74] Shuqian Ye, Jiechun Liang, Rulin Liu, and Xi Zhu. 2020. Symmetrical Graph Neural Network for Quantum Chemistry with Dual Real and Momenta Space. *The Journal of Physical Chemistry A* 124, 34 (2020), 6945–6953.

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

[76] Hanqing Zeng and Viktor Prasanna. 2020. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. 255–265.

[77] Bingyi Zhang, Rajgopal Kannan, and Viktor Prasanna. 2021. BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In *Proceedings of the 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 29–39.

[78] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. 2020. Hardware acceleration of large scale GCN inference. In *Proceedings of the 2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 61–68.

[79] Dalong Zhang, Xin Huang, Ziqi Liu, Zhiyang Hu, Xianzheng Song, Zhibang Ge, Zhiqiang Zhang, Lin Wang, Jun Zhou, Yang Shuang, et al. 2020. Agl: a scalable system for industrial-purpose graph machine learning. *arXiv preprint arXiv:2003.02454* (2020).

[80] Muhan Zhang and Yixin Chen. 2018. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems* 31 (2018), 5165–5175.

[81] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 261–274.

[82] Jun Zhao, Zhou Zhou, Ziyu Guan, Wei Zhao, Wei Ning, Guang Qiu, and Xiaofei He. 2019. Intentgc: a scalable graph convolution framework fusing heterogeneous information for recommendation. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2347–2357.

[83] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: distributed graph neural network training for billion-scale graphs. In *Proceedings of the 2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, 36–44.

[84] Zhe Zhou, Cong Li, Xuechao Wei, and Guangyu Sun. 2021. GCNear: A Hybrid Architecture for Efficient GCN Training with Near-Memory Processing. *arXiv preprint arXiv:2111.00680* (2021).

[85] Zhe Zhou, Bizhao Shi, Zhe Zhang, Yijin Guan, Guangyu Sun, and Guojie Luo. 2021. BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1009–1014.

[86] Hao Zhu, Yankai Lin, Zhiyuan Liu, Jie Fu, Tat-Seng Chua, and Maosong Sun. 2019. Graph neural networks with generated parameters for relation extraction. *arXiv preprint arXiv:1902.00756* (2019).

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