Hardware/Software Co-Programmable Framework for Computational SSDs to Accelerate Deep Learning Service on Large-Scale Graphs

Miryeong Kwon, Donghyun Gouk, Sangwon Lee, and Myoungsoo Jung,
Computer Architecture and Memory Systems Laboratory,
Korea Advanced Institute of Science and Technology (KAIST)

https://www.usenix.org/conference/fast22/presentation/kwon

This paper is included in the Proceedings of the
20th USENIX Conference on File and Storage Technologies.
February 22–24, 2022 • Santa Clara, CA, USA
978-1-939133-26-7

Open access to the Proceedings
of the 20th USENIX Conference on
File and Storage Technologies
is sponsored by USENIX.

Hardware/Software Co-Programmable Framework for Computational SSDs to Accelerate Deep Learning Service on Large-Scale Graphs

Miryeong Kwon Donghyun Gouk Sangwon Lee Myoungsoo Jung
Computer Architecture and Memory Systems Laboratory
Korea Advanced Institute of Science and Technology (KAIST)
http://camelab.org

Abstract

Graph neural networks (GNNs) process large-scale graphs consisting of a hundred billion edges. In contrast to traditional deep learning, unique behaviors of the emerging GNNs are engaged with a large set of graphs and embedding data on storage, which exhibits complex and irregular preprocessing.

We propose a novel deep learning framework on large graphs, *HolisticGNN*, that provides an easy-to-use, near-storage inference infrastructure for fast, energy-efficient GNN processing. To achieve the best end-to-end latency and high energy efficiency, HolisticGNN allows users to implement various GNN algorithms and directly executes them where the actual data exist in a holistic manner. It also enables RPC over PCIe such that the users can simply program GNNs through a graph semantic library without any knowledge of the underlying hardware or storage configurations.

We fabricate HolisticGNN's hardware RTL and implement its software on an FPGA-based computational SSD (CSSD). Our empirical evaluations show that the inference time of HolisticGNN outperforms GNN inference services using high-performance modern GPUs by 7.1× while reducing energy consumption by 33.2×, on average.

1 Introduction

Graph neural networks (GNNs) have recently emerged as a representative approach for learning graphs, point clouds, and manifolds. Compared to traditional graph analytic methods, GNNs exhibit much higher accuracy in a variety of prediction tasks [28, 49, 64, 90, 94, 101], and their generality across different types of graphs and algorithms allows GNNs to be applied by a broad range of applications such as social networks, knowledge graphs, molecular structure, and recommendation systems [8, 21, 29, 52]. The state-of-the-art GNN models such as GraphSAGE [27] further advance to infer unseen nodes or entire new graphs by generalizing geometric deep learning (DL). The modern GNN models in practice sample a set of subgraphs and DL feature vectors (called *embeddings*) from the target graph information, and aggregate the sampled embeddings for inductive node inferences [27, 95]. This *node sampling* can significantly reduce the amount of data to process, which can decrease the computation complexity to infer the results without an accuracy loss [9, 27, 96].

While these node sampling and prior model-level efforts for large graphs make the inference time reasonable, GNNs yet face system-level challenges to improve their performance. First, GNNs experience a completely different end-to-end inference scenario compared to conventional DL algorithms. In contrast to the traditional DLs, GNNs need to deal with real-world graphs consisting of billions of edges and node embeddings [19, 73]. The graph information (graph and node embeddings) initially reside in storage and are regularly updated as raw-format data owing to their large size and persistence requirements. As GNNs need to understand the structural geometry and feature information of given graph(s), the raw-format data should be loaded into working memory and reformatted in the form of an adjacency list before the actual inference services begin. These activities take a significant amount of time since the graph information often exceeds hundreds of GBs or even a TB of storage [18, 76]. We observe that the pure inference latency, that all the previous GNN studies try to optimize, accounts for only 2% of the end-to-end inference service time when we execute diverse GNN models in a parallel system employing high-performance GPUs [13, 14] and an SSD [12]. We will analyze this performance bottleneck issue with detailed information in Section 2.3.

Second, GNNs consist of various computing components, which are non-trivial to fully accelerate or parallelize over conventional computing hardware. As GNNs are inspired by conventional DL algorithms such as convolution neural networks and representative learning [27, 64, 79], several data processing parts of GNN computing are associated with dense matrix computing. While these matrix multiplications can be accelerated by existing data processing units (DPUs) such as systolic architectures, the graph-natured operations of GNNs can neither be optimized with DPU's multiplication hardware nor with GPUs' massive computing power [5].

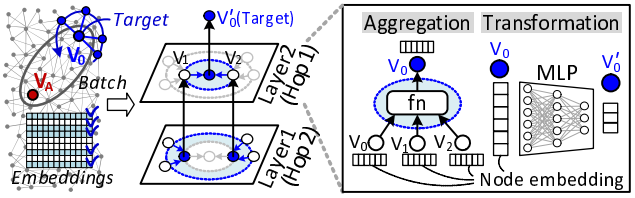
A promising alternative to address the aforementioned challenges is employing in-storage processing (ISP) to serve GNN inferences directly from the underlying storage. While ISP is very a well-known solution heavily studied in the literature for the past few decades [25, 34, 43, 46, 58, 65], it has

unfortunately not been widely adopted in real-world systems [6]. There are several reasons, but the most critical issue of ISP is ironically its concept itself, which co-locates flash and computing unit(s) into the same storage box. As flash is not a working memory but a block device, it is integrated into the storage box with complicated firmware and multiple controllers [11, 36, 99]. These built-in firmware and controllers are not easily usable for the computation that users want to offload as it raises many serious technical challenges such as programmability, data protection, and vendor dependency issues. In this work, we advocate a new concept of *computational SSD* (CSSD) architectures that locate reconfigurable hardware (FPGA) near storage in the same PCIe subsystem [20, 68, 80]. In contrast to ISP, CSSD can maximize peer-to-peer acceleration capability and make it independent from specific storage firmware and controller technologies. However, it is challenging to configure everything that users want to program and/or download in the form of completely full hardware logic into FPGA from scratch.

We propose *HolisticGNN*, a hardware and software co-programmable framework that leverages CSSD to accelerate GNN inference services near storage. HolisticGNN offers a set of software and hardware infrastructures that execute GNN tasks where data exist and infer the results from storage in a holistic manner. Generally speaking, the software part of HolisticGNN enables users to program various GNN algorithms and infer embedding(s) directly atop the graph data without the understanding complexities of the underlying hardware and device interfaces. On the other hand, our hardware framework provides fundamental hardware logic to make CSSD fully programmable. It also provides an architectural environment that can accelerate various types of GNN inferences with different hardware configurations.

For fast and energy-efficient GNN processing, our framework is specifically composed of three distinguishable components: i) graph-centric archiving system (*GraphStore*), ii) programmable inference client and server model (*GraphRunner*), and iii) accelerator building system (*XBuilder*). The main purpose of GraphStore is to bridge the semantic gap between the graph abstraction and its storage representation while minimizing the overhead of preprocessing. GraphStore manages the user data as a graph structure rather than exposing it directly as files without any intervention of host-side software. This allows diverse node sampling and GNN algorithms to process the input data near storage immediately. GraphStore also supports efficient mutable graph processing by reducing the SSD's write amplification.

To accommodate a wide spectrum of GNN models, it is necessary to have an easy-to-use, programmer-friendly interface. GraphRunner processes a series of GNN inference tasks from the beginning to the end by allowing users to program the tasks using a computational graph. The users can then simply transfer the computational graph into the CSSD and manage its execution through a remote procedure call (RPC). This does not require cross-compilation or storage



(a) Preprocessing. (b) GNN processing. (c) Layer's details.

Figure 1: Overview of basic GNN algorithm.

stack modification to program/run a user-defined GNN model. We enable RPC by leveraging the traditional PCIe interface rather than having an extra hardware module for the network service, which can cover a broad spectrum of emerging GNN model implementations and executions in CSSD.

On the other hand, XBuilder manages the FPGA hardware infrastructure and accelerates diverse GNN algorithm executions near storage. It first divides the FPGA logic die into two regions, *Shell* and *User*, using the dynamic function exchange (DFX) technique [82]. XBuilder then secures hardware logic necessary to run GraphStore and GraphRunner at Shell while placing DL accelerator(s) to User. The Shell and User hardware are programmed to CSSD as two separate bitstreams, such that we can reprogram the User with a different bitstream at any time. To this end, XBuilder implements a hardware engine in Shell by using an internal configuration access port, which downloads a bitstream and programs it to User.

We implement HolisticGNN on our CSSD prototype that places a 14*nm* FPGA chip [87] and 4TB NVMe device [12] under a same PCIe switch. We also prototype the software framework of HolisticGNN on the CSSD bare-metal, and we fabricate/test various GNN accelerator candidates within CSSD, such as a many-core processor, systolic arrays, and a heterogeneous (systolic+vector) processor. Our evaluations show that the inference time of HolisticGNN outperforms GNN inference services using high-performance GPUs by 7.1× while consuming 33.2× less energy, on average.

2 Background

2.1 Graph Neural Networks

Graph neural networks (GNNs) generalize conventional DL to understand structural information in the graph data by incorporating feature vectors (*embeddings*) in the learning algorithms [27, 28, 95]. GNNs can capture topological structures of the local neighborhood (per node) in parallel with a distribution of the neighborhood's node embeddings [7, 9, 27, 96].

General concept. As shown in Figure 1, GNNs in general take three inputs, a graph, the corresponding node embeddings (e.g., user profile features), a set of unseen/seen nodes to infer, called *batch*. Since the internal memory of GPUs is insufficient to accommodate all the inputs, it is essential to reduce the size of the graph and embeddings by preprocessing them appropriately (Figure 1a), which will be explained in Section 2.2. GNNs then analyze the preprocessed structural

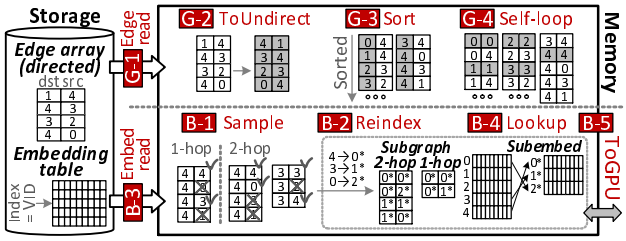


Figure 2: Holistic viewpoint of GNN computing (**G**: Graph preprocessing, **B**: Batch preprocessing).

information with node embeddings over multiple computational layers (Figure 1b). Each layer of GNNs is composed of two primary execution phases, called neighborhood *aggregation* and node *transformation* [79, 90], which are all performed for neighbors at different hop distances (connected to a target node in the batch). Specifically, as shown in Figure 1c, the aggregation is a simple function to accumulate node embeddings of the target node's neighbors, whereas the transformation converts the aggregated results to a new node embedding using one or more traditional *multi-layer perceptrons* (MLPs [31, 32]). Therefore, the aggregation processes data relying on graph structures and mainly exhibits irregular, graph-natured execution patterns. In contrast, the transformation computing procedure is very similar to that of conventional neural networks (e.g., CNNs and RNNs), but it does not require heavy computation. For example, GNNs mostly use only 2∼3 layers [15, 42, 72, 75, 90], whereas Google BERT employs more than 24 layers and needs to perform heavy matrix multiplications [17].

Note that, while the massive parallel computing of GPUs is very well-optimized for many DL algorithm executions, these characteristics of GNNs (e.g., irregular execution pattern and relatively lightweight computation) allow other processing architectures to be a better fit for GNN acceleration.

Model variations. Based on how to aggregate/transform embeddings, there is a set of variant GNNs, but *graph convolution network* (GCN [42]), *graph isomorphism network* (GIN [90]), and *neural graph collaborative filtering* (NGCF [75]) are the most popular GNN models used in node/graph classification and recommendation systems [21, 29, 49, 94, 96, 101].

Specifically, GCN uses an *"average-based aggregation"* that normalizes the embeddings by considering the degree of neighbor nodes. This prevents cases where a specific embedding has excessively large features, thereby losing other embeddings that have relatively small amounts of data in the aggregation phase. In contrast, GIN uses a *"summation-based aggregation"* that does not normalize the embeddings of both the target node (self-loop) and its neighbors. In addition, GIN gives a learnable self-weight to the target node embedding to avoid unexpectedly losing its feature information due to the heavy states and features of the target node's neighbors. To precisely capture the structural characteristics of the given graph, GIN uses a two-layer MLP structure, making the combination more expressively powerful. GCN and GIN suppose that all the feature vectors of a given graph have the same level of weight, which are widely used for node and graph classifications [54, 94, 101]. Instead of using a simple average or summation for the aggregation, NGCF takes the similarity among the given graph's embeddings into account by applying an element-wise product to neighbors' embeddings.

Even though there are several variants of GNNs, they all require the graph's geometric information to analyze embeddings during the aggregation and transformation phases. Thus, it is necessary for GNNs to have an easy-to-access, efficient graph and embedding data structures.

2.2 Graph Dataset Preprocessing

The graph data offered by a de-facto graph library such as SNAP [48] in practice deal with edge information in the form of a text file. The raw graph file includes an (unsorted) edge array, each being represented by a pair of destination and source *vertex identifiers* (VIDs). Most GNN frameworks such as deep graph library (DGL) [74] and pytorch geometric (PyG) [22] preprocess the graph dataset to secure such easy-to-access graph and embeddings as a VID-indexed table or tensor. In this work, we classify these graph dataset preprocessing tasks into two: i) *graph preprocessing* and ii) *batch preprocessing*. While graph preprocessing is required only for the geometrical data (including the initialization), batch preprocessing should be performed for each inference service.

Graph preprocessing. Since the majority of emerging GNN algorithms are developed based on spatial or spectral networks encoding *"undirected"* geometric characteristics [15, 42], the main goal of this graph preprocessing is to obtain a sorted, undirected graph dataset. As shown in the top of Figure 2, it first loads the edge array (raw graph) from the underlying storage to the working memory [G-1]. To convert the array to an undirected graph, the GNN frameworks (e.g., DGL) allocate a new array and copy the data from the edge array to the new array by swapping the destination and source VIDs for each entry ($\{dst, src\} \rightarrow \{src, dst\}$) [G-2]. The frameworks merge and sort the undirected graph, which turns it into a VID-indexed graph structure [G-3]. As the target node to infer is also included in the 1-hop neighborhood, the frameworks inject self-loop information (an edge connecting a vertex to itself) to the undirected graph as well ($\{0,0\}, \{1,1\}, \cdots \{4,4\}$) [G-4]. If there is no self-loop information, the aggregation of GNNs cannot reflect a visiting node's features, which in turn reduces the inference accuracy significantly.

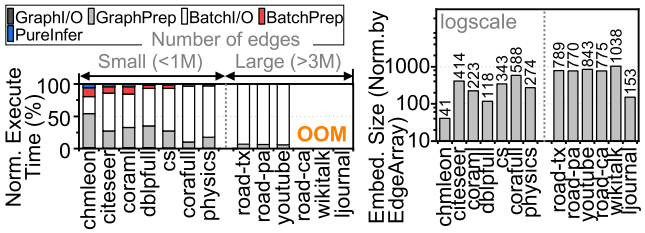
Batch preprocessing. Large-scale real-world graphs consist of a hundred billion edges, and each node of the edges is associated with its own embedding containing thousands of DL features. The number of nodes and the embedding size that the current GNN models process are typically an order of magnitude greater than heavy featured DL applications, such as natural language processing [18, 76]. Thus, for a given batch, the frameworks in practice perform *node sampling* such as random walk [92] and unique neighbor sampling [27]. The node sampling specifically extracts a set of subgraphs

and the corresponding embeddings from the original (undirected) graph datasets before aggregating and transforming the feature vectors, which can significantly reduce data processing pressures and decrease the computing complexity without an accuracy loss [27, 33]. Since the sampled graph should also be self-contained, the subgraphs and embeddings should be reindexed and restructured. We refer to this series of operations as *batch preprocessing*.

The bottom of Figure 2 shows an example of batch preprocessing. For the sake of brevity, this example assumes that the batch includes a just single target, V_4 (VID=4), the given sampling size is 2, and GNN is modeled with two layers (two hops). This example first reads all V_4's neighbors and extracts two nodes from the undirected graph (in a random manner) [B-1]. This generates a subgraph including the 1-hop neighbors, which is used for GNN's layer 2 computation (L2). For the sampled nodes (V_4 and V_3), it reads their neighbor nodes (2-hop) and samples the neighborhood again for GNN's layer 1 (L1). Since the number of nodes has been significantly reduced, the GNN frameworks allocate new VIDs in the order of sampled nodes (4→ 0*, 3→ 1*, and 0→ 2*) and create L1 and L2 subgraphs for 2-hop and 1-hop neighbors, respectively [B-2]. It then composes an embedding table whose index is the VID of each sampled node. To this end, the frameworks first need to load the embeddings from the underlying storage to working memory [B-3], called global embeddings, and lookup the embeddings of L1's subgraph (V_4, V_0, and V_3) [B-4]. Lastly, the subgraphs and sampled embedding table are required to transfer from the working memory to the target GPU's internal memory [B-5].

2.3 Challenge Analysis

While there is less system-level attention on the management of graph and batch preprocessing, their tasks introduce heavy storage accesses and frequent memory operations across the boundary of user space and storage stack. To be precise, we decompose the *"end-to-end GCN inference"* times across 14 real-world graph workloads (coming from [48, 61, 66, 93]) into the latency of graph preprocessing (`GraphPrep`), batch preprocessing (`BatchPrep`), GCN inference processing (`PureInfer`), and storage accesses for graph (`GraphI/O`) and embeddings (`BatchI/O`). Since the storage access latency being overlapped with the latency of preprocessing computation is invisible to users, this breakdown analysis excludes such latency, and the results are shown in Figure 3a. The detailed information of the evaluation environment is provided by Section 5. One can observe from this breakdown analysis that `PureInfer` only takes 2% of the end-to-end inference latency, on average. Specifically, `BatchI/O` accounts for 61% of the most end-to-end latency for the small graphs having less than 1 million edges. Before the sorted and undirected graph is ready for batch preprocessing, `BatchI/O` cannot be processed. Since `GraphPrep` includes a set of heavy (general) computing processes such as



(a) Latency breakdown. (b) Embed vs. Edge array.

Figure 3: End-to-End GNN execution.

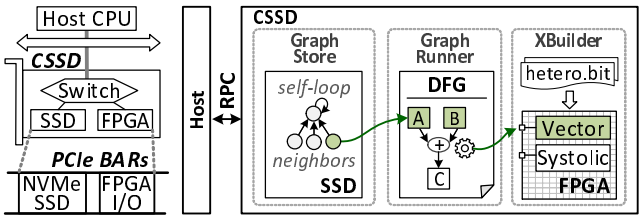
a radix sort, `GraphPrep` also consumes 28% of the end-to-end latency for these small graphs. As the graph size increases (> 3 million edges), `BatchI/O` becomes the dominant contributor of the end-to-end GNN inference time (94%, on average). Note that the inference system has unfortunately stopped the service during the preprocessing due to out-of-memory (OOM) when it handles large-scale graphs (>3 million edges) such as `road-ca`, `wikitalk`, and `ljournal`. This OOM issue can be simply addressed if one services the GNN inference from where the data exist. In addition, the heavy storage accesses and relatively lightweight computing associated with inference (`PureInfer`) make adoption of the in-storage processing concept [39] reasonable to shorten the end-to-end inference latency.

Figure 3b normalizes the size of the embedding table to that of the edge array (graph) across all the graphs that we tested. As shown in this comparison, embedding tables of the small and large graphs are greater than the edge arrays for those graphs by 285.7× and 728.1×, on average, respectively. This is because an embedding has thousands of DL features, each represented using floating-point values with high precision [61, 95]. In contrast, an entry of the edge arrays contains only a simple integer value (VID). This characteristic makes batching preprocessing I/O intensive while inducing graph preprocessing to be computation-intensive.

3 Storage as a GNN Accelerator

In-storage processing (ISP) is well-studied in the research literature [2, 25, 35, 41, 43, 46, 58, 62, 65], but it has been applied to accelerate limited applications such as compression and key-value management in real-world systems. There are several reasons, but the greatest weakness of ISP ironically is that it needs to process data where data is stored, i.e., at the flash device. Flash cannot be directly used as block storage because of its low-level characteristics, such as I/O operation asymmetry and low reliability [10, 37, 38, 55, 56, 78]. Thus, flash requires tight integration with multiple firmware and controller modules [98, 99], which renders ISP difficult to be implemented within an SSD.

In contrast to ISP, as shown in Figure 4a, the new concept of computational SSDs (CSSDs) decouples the computing unit from the storage resources by locating *reconfigurable hardware* (FPGA) near SSD in the same PCIe subsystem (card) [20]. CSSD allows the hardware logic fabricated in FPGA to access the internal SSD via the internal PCIe switch.



(a) CSSD. (b) Overview of HolisticGNN.

Figure 4: Enabling CSSD for near storage GNN processing.

To this end, the host is responsible for writing/reading data on the SSD using the I/O region of NVMe protocol while giving the data's block address to the FPGA through its own I/O region, whose address is designated by PCIe's base address register [84]. While CSSD is promising to realize near-data processing [44, 68], it is non-trivial to automate all end-to-end procedures of GNN inference over hardware-only logic because of the variety of GNN model executions. For example, the aggregation and/or combination of GNNs can be accelerated with parallel hardware architecture, but GNN's graph traversing, dataset preprocessing, and embedding handling are impractical to be programmed into hardware because of their graph-natured computing irregularities.

3.1 Overview of HolisticGNN

HolisticGNN is a hardware and software co-programmable framework that leverages CSSD to accelerate the end-to-end GNN inference services near storage efficiently. The software part of our framework offers easy-to-use programming/management interfaces and performs GNN preprocessing directly from where the data is stored, thereby minimizing the aforementioned storage access overhead. HolisticGNN can also eliminate the out-of-memory issue for deep learning on large-scale graphs. On the other hand, our framework's hardware logic and administration module provide a low-overhead bare-metal computing environment and reconfigurable hardware to accelerate GNN model executions.

Figure 4b illustrates a high-level view of HolisticGNN, which is composed of three major modules: i) graph-centric archiving system (*GraphStore*), ii) programmable inference model (*GraphRunner*), and iii) accelerator builder (*XBuilder*). Generally speaking, GraphStore prepares the target graph data and embeddings in a ready-to-access structure that the tasks of batch preprocessing can immediately use without preprocessing the datasets. On the other hand, GraphRunner executes a series of GNN inference tasks from the beginning to the end, and it processes the graph datasets by directly accessing SSD through GraphStore. GraphRunner also provides a *dataflow graph* (DFG) based program and execution model to support easy-to-use and flexible implementation of a wide spectrum of GNN algorithms. This enables the users to simply generate a DFG and deliver it to HolisticGNN, which can dynamically change the end-to-end GNN inference services without cross-compilation and/or understanding underlying hardware configurations. Lastly, XBuilder makes CSSD simply reconfigurable and has heterogeneous hardware components to satisfy the diverse needs of GNN inference acceleration services. XBuilder also provides several kernel building blocks, which abstract the heterogeneous hardware components. This can decouple a specific hardware acceleration from the GNN algorithm implementation.

Service type	RPC function	Service type	RPC function
GraphStore (Bulk)	`UpdateGraph (EdgeArray, Embeddings)`	GraphStore (Unit, Get)	`GetEmbed(VID)`
GraphStore (Unit, Update)	`AddVertex(VID,Embed)`		`GetNeighbors(VID)`
	`DeleteVertex(VID)`	Graph Runner	`Run(DFG, batch)`
	`AddEdge(dstVID,srcVID)`		`Plugin(shared_lib)`
	`DeleteEdge(dstVID,srcVID)`	XBuilder	`Program(bitfile)`
	`UpdateEmbed(VID,Embed)`		

Table 1: RPC services of HolisticGNN.

Each module of our framework exposes a set of APIs through remote procedure calls (RPCs) to users. These APIs are not related to GNN programming or inference services, but to framework management such as updating graphs, inferring features, and reprogramming hardware logic. Since CSSD has no network interface for the RPC-based communication, we also provide an *RPC-over-PCIe* (RoP) interface that overrides the conventional PCIe to enable RPC between a host and CSSD without an interface modification.

3.2 Module Decomposition

Graph-centric archiving system. The main goal of GraphStore is to bridge the semantic gap between graph and storage data without having a storage stack. As shown in Table 1, GraphStore offers two-way methods for the graph management, *bulk operations* and *unit operations*. The bulk operations allow users to update the graph and embeddings with a text form of edge array and embedding list. For the bulk operations, GraphStore converts the incoming edge array to an adjacency list in parallel with transferring the embedding table, and it stores them to the internal SSD. This makes the conversion and computing latency overlapped with the heavy embedding table updates, which can deliver the maximum bandwidth of the internal storage. In contrast, the unit operations of GraphStore deal with individual insertions (`AddVertex()`/`AddEdge()`), deletions (`DeleteVertex()`/`DeleteEdge()`), and queries (`GetEmbed()`/`GetNeighbors()`) for the management of graph datasets. When GraphStore converts the graph to storage semantic, it uses VID to *logical page number* (LPN) mapping information by being aware of a long-tailed distribution of graph degree as well as flash page access granularity. The LPNs are used for accessing CSSD's internal storage through NVMe, which can minimize the write amplification caused by I/O access granularity differences when CSSD processes GNN services directly on the SSD. The design and implementation details are explained in Section 4.1.

Programmable inference model. GraphRunner decouples CSSD task definitions from their actual implementations, which are called *C-operation* and *C-kernel*, respectively. To program a GNN model and its end-to-end service, the users can write a DFG and download/execute to CSSD by call-

API type	Function	API type	Operation format
DFG Creation	createIn(name)	XBuilder	GEMM(inputs, output)
	createOp(name)		ElementWise(inputs, output)
	createOut(name)		Reduce(inputs, output)
	save(graph)		SpMM(inputs, output)
Plugin	RegisterDevice(newDevice)		SDDMM(inputs, output)
	RegisterOpDefinition(newOp)		

Table 2: Programming interface of HolisticGNN.

ing GraphRunner's RPC interface (`Run()`) with a request batch containing one or more target nodes. Figure 10b shows a simple example of GCN implementation. The DFG has a set of input nodes for the target sampled subgraphs, embeddings, and weights, which are connected to a series of C-operations such as averaging features (Mean), matrix multiplication (Matmul), a non-linear function (ReLU), and output feature vector (Out_embedding). This DFG is converted to a computational structure by sorting the node (C-operation) and edge (input node information) in topological order. Once the DFG is downloaded through HolisticGNN's RoP serialization, GraphRunner's engine deserializes it and executes each node with appropriate inputs by checking the registered C-operations and C-kernels in CSSD. The users may want to register more C-operations/kernels because of adoption of a new GNN model or hardware logic. To meet the requirement, GraphRunner offers a Plugin mechanism registering a pair of C-operation/C-kernel and a new device configuration as a shared object. We will explain the details of GraphRunner in Section 4.2.

Accelerator builder. To make the FPGA of CSSD easy to use, we configure CSSD's hardware logic die into two groups, *Shell* and *User* logic, by leveraging a dynamic function exchange (DFX) mechanism [82]. DFX allows hardware to be modified blocks of logic with separate *bitfiles* that contain the programming information for an FPGA. XBuilder secures Shell logic associated with irregular tasks of GNN management, including GraphStore and GraphRunner executions, while managing User logic for users to reprogram the hardware in accelerating GNN algorithms via XBuilder's RPC interface (`Program()`). `Program()` moves a partial bitfile into the internal memory and asks an XBuilder engine to reconfigure User logic hardware using the bitfile via FPGA internal configuration access port [85, 86].

XBuilder also abstracts the registered device (at User logic) by providing a set of basic building blocks to the users as shown in Table 2. The building blocks basically implement what DL and GNN algorithms mostly use, such as general matrix multiplication (GEMM) and sparse matrix multiplication (SpMM), across different legacy acceleration hardware such as multi-core, vector processor, systolic architecture. XBuilder's building blocks operate specific hardware based on the device priority designated by C-kernel that user defines. We will discuss this in details in Section 4.3.

3.3 Enabling RPC over PCIe

While the key method to program GNN models (and request their inference services) is associated with DFG, the underpinning of such a device-to-device communication method

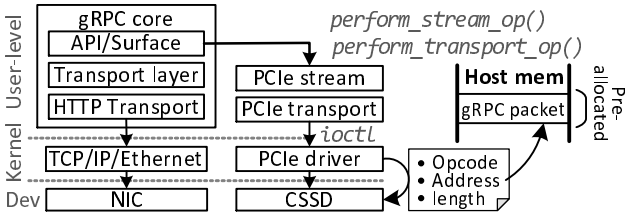


Figure 5: RPC over PCIe (RoP).

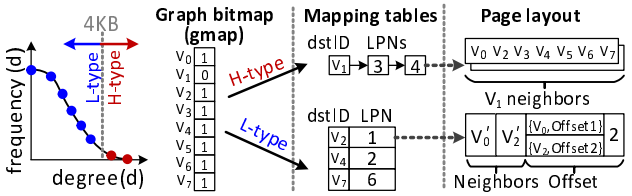
is RPC. As the investigation of efficient RPC is not the purpose of this work, we use Google's gRPC [24] and implement our RPC-based interfaces themselves (e.g., `UpdateGraph()`, `Run()`, etc.) using interface definition language (IDL) [16]. We also modify the gRPC stack to enable RPC services without changing hardware and storage interfaces.

Figure 5 explains the gRPC stack and how HolisticGNN enables gRPC over PCIe. The host-side gPRC interfaces are served by a user-level gRPC core, which manages transport and HTTP connection. We place two gRPC plugin interfaces (`perform_stream_op()` and `perform_transport_op()`), each forwarding the requests of gRPC core's transport layer and HTTP transport to our PCIe stream and PCIe transport modules. Specifically, the PCIe stream is responsible for managing `stream` data structures, which are used for gPRC packet handling. Similarly, the PCIe transport deals with the host and CSSD connection by allocating/releasing `transport` structures. While the original gPRC core is built upon a kernel-level network stack including TCP/IP and Ethernet drivers, we place a PCIe kernel driver connected to the PCIe transport. It supports gRPC's send/receive packet services and other channel establishment operations to the PCIe transport module via `ioctl`. The PCIe kernel driver also provides preallocated buffer memory to the PCIe stream through a memory-mapped I/O (`mmap`). This buffer memory contains gPRC packet's metadata and message such that the PCIe driver lets the underlying CSSD know the buffer's location and offset. Specifically, the PCIe drive prepares a PCIe command that includes an opcode (send/receive), address (of memory-mapped buffer), and length (of the buffer). When the driver writes the command to FPGA's designated PCIe memory address, CSSD parses the command and copies the data from the memory-mapped buffer into FPGA-side internal memory for gRPC services.

4 Design Details and Implementation

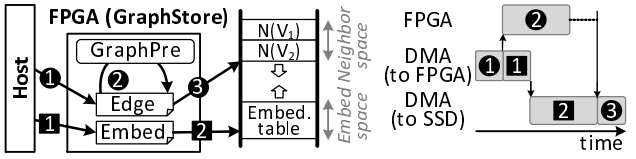
4.1 Efficient Storage Accesses for Graphs

GraphStore maintains the graph datasets as an adjacency list and an embedding table to handle the geometric information and feature vectors. While the embedding table is stored in sequential order (and thus it does not require page-level mapping), the adjacency list is maintained in two different ways by considering the efficiency of graph searches/updates: i) high-degree graph mapping (*H-type*) and ii) low-degree graph mapping (*L-type*). As shown in Figure 6a, the power-law graph's natures make a few nodes have severely heavy



(a) Distribution. (b) Two types of mapping.

Figure 6: GraphStore's mapping structure.



(a) Procedure. (b) Overlapped execution.

Figure 7: Bulk operations.

neighbor nodes (high-degree) [59]. These high-degree nodes account for a small fraction of the entire graph, but they have a high potential to be frequently accessed and updated (because of their many neighbors). H-type mapping is therefore designed towards handling the graph's long-tailed distribution well, while L-type mapping is structured to achieve high efficiency of flash page management.

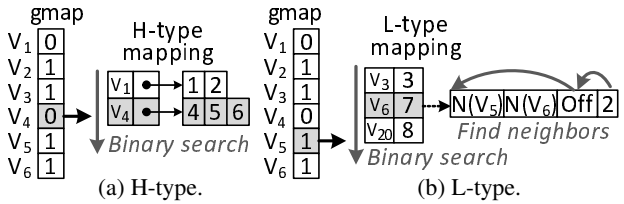
Mapping structure. As shown in Figure 6b, GraphStore has a graph bitmap (*gmap*), which explains what kind of tables are used for mapping (per VID). Basically, the mapping entry for both types of mapping tables pairs a VID and an LPN (VID-to-LPN), but the corresponding page stores different data with its own page layout. The H-type page maintains many neighbors' VID in a page, and its mapping table entry indicates a linked list in cases where the neighbors of the target (source) VID cannot be stored in a flash page (4KB). The L-type page also contains many VIDs, but their source VIDs vary. To this end, the end of page has meta-information that indicates how many nodes are stored and where each node exists on the target page (offset). Thus, L-type mapping table's VID is the biggest VID among VIDs stored in the corresponding page.

Bulk operation. As shown in Figure 7, while the embedding table is stored from the end of LPN (embedding space), the graph pages are recorded from the beginning of storage (neighbor space), similar to what the conventional memory system stack does. Note that, the actual size of graph(s) is small enough, but it is involved in heavy graph preprocessing, and the majority of graph datasets are related to their node embeddings (cf. Section 2.3). Thus, when an edge array (graph) arrives, GraphStore performs graph preprocessing and flushes pages for the graph, but it does not immediately update them to the target storage. Instead, GraphStore begins to write the embedding table into the embedding space in a sequential manner while preprocessing the graph. This can make heavy storage accesses (associated with the embeddings) entirely overlap with the computation burst of graph preprocessing (associated with adjacency list conversions). From the user's viewpoint, the latency of bulk operation is the same as that of data transfers and embedding table writes.

Unit operations. GraphStore's unit operations support mutable graph management corresponding to individual vertex/edge updates or queries. Figure 8 shows how to find out neighbors of V_4 and V_5, each being classified as high-degree and low-degree nodes. In this example, as the gmap indicates that V_4 is managed by the H-type mapping, the neighbors can be simply retrieved by searching where the target VID is. In contrast, the page managed by L-type contains many neighborhoods each being associated with different VID. Therefore, when GraphStore searches the mapping table, it considers the range of VIDs, stored in each entry. For example, V_5 is within the range of V_4 and V_6, GraphStore first needs to retrieve the page corresponding V_6. It finds out V_5's offset and the next VID's offset (V_6) by considering the number of node counts in the page's meta-information, which indicates the data chunk containing V_5's neighbors.

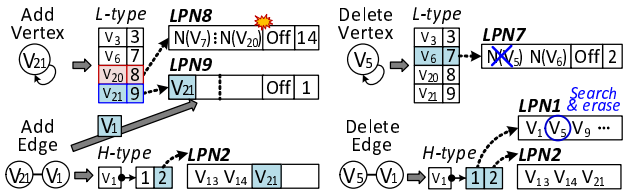
Figures 9a and 9b show add operations (`AddEdge()` / `AddVertex()`) and delete operations (`DeleteEdge()` / `DeleteVertex()`). Let us suppose that V_{21} is given by `AddVertex()` (Figure 9a). GraphStore checks the last entry's page (LPN8) of L-type and tries to insert V_{21} into the page. However, as there is no space in LPN8, GraphStore assigns a new entry ([V_{21},9]) to the L-type mapping table by allocating another page, LPN9, and simply appends the vertex information (V_{21}) to the page. Note that, when adding a vertex, it only has the self-loop edge, and thus, it starts from L-type. When $V_{21}\rightarrow V_1$ is given by `AddEdge()`, GraphStore makes it an undirected edge ($V_{21}\rightarrow V_1$ & $V_{21}\leftarrow V_1$). As V_1 is H-type, GraphStore checks V_1's linked list and places V_{21} to the last page (LPN2). If there is no space in LPN2, it allocates a new page and updates the linked list with the newly allocated page. In contrast, since V_{21} is L-type, GraphStore scans the meta-information of LPN9 and appends V_1 to the page. Note that, in cases where there is no space in an L-type page, GraphStore evicts a neighbor set (represented in the page) whose offset of the meta-information is the most significant value. This eviction allocates a new flash page, copies the neighbor set, and updates L-type mapping table. Since each L-type's destination node has a few source nodes, this eviction case is very rare in practice (lower than 3% of the total update requests for all graph workloads we tested).

On the other hand, delete operations (Figure 9b) consist of search and erase tasks. If `DeleteVertex()` is called with V_5, GraphStore finds out LPN7 and deletes all the neighbors of V_5, N(V_5). During this time, other neighbors having V_5 should also be updated together. For `DeleteEdge()` with the given $V_5\rightarrow V_1$, GraphStore checks all the LPNs indicated by the



(a) H-type. (b) L-type.

Figure 8: Unit operations (Get).



(a) AddEdge/AddVertex. (b) DeleteEdge/DeleteVertex.

Figure 9: Unit operations (Update).

linked list of V_1 and updates the corresponding page (LPN1 in this example) by removing V_5. Note that, GraphStore does not have explicit page compaction for the node/edge deletions in an L-type page. This is because, when there is a deletion, GraphStore keeps the deleted VID and reuses it (and the corresponding neighbor set space) for a new node allocation.

4.2 Reconfiguring Software Framework

HolisticGNN provides a CSSD library package, which includes the interfaces for C-kernel registration and DFG management as explained previously (cf. Table 2).

C-kernel registration and management. GraphRunner manages C-kernels by employing a registration mechanism and an execution engine. GraphRunner has two metadata structures, *Device table* and *Operation table*. As shown in Table 3, the device table includes currently registered device names and the corresponding priority. On the other hand, the operation table maintains C-operation names and the address pointers of their C-kernel implementation. When users implement a C-kernel, it should invoke two registration interface methods of the Plugin library, `RegisterDevice()` and `RegisterOpDefinition()`, at its initial time. `RegisterDevice()` configures the priority value of the device that users want to execute for any of C-kernels (e.g., "Vector processor", 150). On the other hand, `RegisterOpDefinition()` registers the device that this C-kernel. When GraphRunner registers the C-kernel, it places the registration information as a pair of the device name and such C-kernel's pointer. If there are multiple calls of `RegisterOpDefinition()` with the same name of a C-operation (but a different name of device), GraphRunner places it in addition to the previously registered C-kernels as a list. In this example, GraphRunner can recognize that GEMM C-operation defines three C-kernels each using "CPU", "Vector processor", and "Systolic array" by referring to the operation table. Since the device table indicates "Systolic array" has the highest priority, GraphRunner takes the C-kernel associated with "Systolic array" for the execution of GEMM C-operation.

Device table

Name	Priority
"CPU"	50
"Vector processor"	150
"Systolic array"	300
...	...

Operation table

Name	C-kernel
"GEMM"	<"CPU", ptr>
	<"Vector processor", ptr>
	<"Systolic array", ptr>
...	...

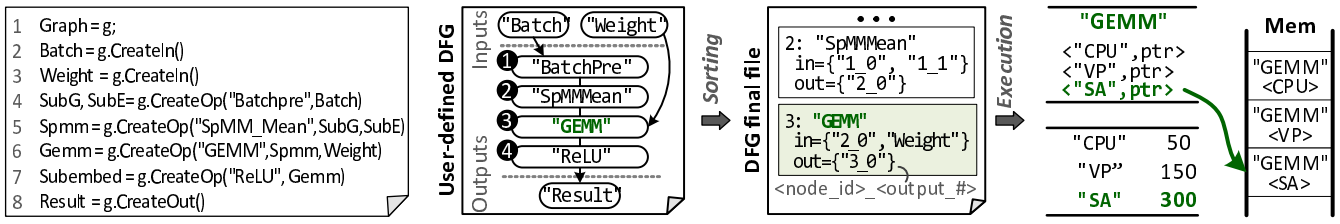
Table 3: GraphRunner's metadata structure.

Handling computational graphs. DFG management interfaces of the CSSD library (`CreateIn()`, `CreateOut()` and `CreateOp()`) are used for explaining how C-operations are mapped to DFG's nodes and how their input and output parameters are connected together (Table 2).

Figure 10a shows how the users can create a DFG to implement a GCN inference service as an example. The input and output of this DFG is `Batch`, `Weight`, and `Result`. `BatchPre` is the first C-operation that takes Batch as its input (❶), and the result is forwarded to `SpMM_Mean` C-operation (❷), which performs GCN's average-base aggregation. Then, the result of `SpMM_Mean` is fed to GCN's transformation consisting of GEMM (having Weight) and ReLU C-operations (❸/❹). The final output should be `Result` in this DFG. Note that, ReLU is a function of MLPs, which prevents the exponential growth in the computation and vanishing gradient issue [40]. The user can write this DFG using our computation graph library as shown in Figure 10b. It declares `Batch` and `Weight` by calling `CreateIn()` (lines 2∼3). `BatchPre` (❶), `SpMM_Mean` (❷), `GEMM` (❸), and `ReLU` (❹) are defined through `CreateOp()`, which are listed in lines 4∼7.

GraphRunner then sorts the calling sequence of CSSD library interfaces and generates a markup file as shown in Figure 10c. This DFG final file includes a list of nodes, each defining its node sequence number, C-operation name, where the input(s) come from, and what the output(s) are. For example, the third node is `GEMM(3: "GEMM")`, and its inputs come from the second node's first output (`2_0`) as well as input node, `Weight` (`in={"2_0", "Weight"}`). This node generates one output only (`out={"3_0"}`).

Execution of DFG. The host can run CSSD with the programmed GNN by downloading the corresponding DFG and a given batch through `Run()` RPC. As shown in Figure 10d, GraphRunner's engine visits each node of the DFG and checks the node's C-operation name. For each node, the engine finds the set of C-kernels (matched with the C-operation name) by checking the operation table. It then refers to the device



(a) Example of DFG programming. (b) Example of DFG. (c) DFG file generation. (d) Execution.

Figure 10: Overview of reconfigurable software framework (GraphRunner).

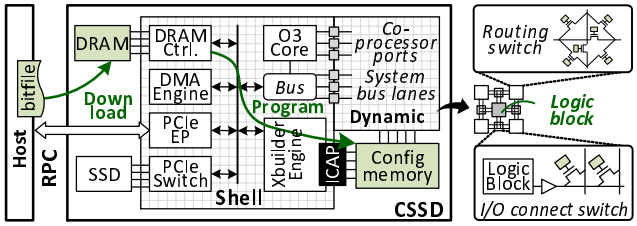


Figure 11: Reconfigurable hardware.

table and selects the appropriate implementation among the retrieved C-kernels based on the device priority, assigned by `RegisterDevice()`. The engine de-refers the C-kernel's address pointer and calls it by passing the C-kernel's parameters, which can also be checked up with offloaded DFG's node information (e.g., `in={...}`). Note that, GraphRunner's engine performs these dynamic binding and kernel execution for all the nodes of DFG per GNN inference.

4.3 Managing Reconfigurable Hardware

As shown in Figure 11, XBuilder provides static logic at Shell logic that includes an out-of-order core, a DRAM controller, DMA engines, and a PCIe switch. This static logic is connected to User (dynamic) logic through a co-processor port such as RoCC [4] and system bus (e.g., TileLink [67]).

XBuilder exposes the boundary position of the FPGA logic die in the form of a design checkpoint file [81, 89]. The boundary is wire circuits that separate Shell and User logic, called *partition pin*. Since the static logic is fixed at the design time, we place the maximum number of co-processor ports and system bus lanes to the partition pin, which can be united with the hardware components fabricated in Shell logic. In addition, we locate an XBuilder hardware engine in Shell, which includes the internal configuration access port (ICAP [85, 86]). Note that, FPGA logic blocks are connected by many wires of routing switches and input/output connection switches. Since the switches maintain the status of connection in built-in FPGA memory, called *configuration memory*, we can reconfigure the FPGA hardware by reprogramming the connection states on the configuration memory. As the configuration memory should be protected from anonymous accesses, FPGA only allows the users to reprogram the configuration memory only through the primitive hardware port, ICAP. The users can simply call HolisticGNN's RPC interface, `Program` with their own hardware (partial) bitfile to reconfigure User logic. XBuilder copies the bitfile into CSSD's FPGA internal DRAM first, and then, it reconfigures User logic by programming the logic using the bitfile via ICAP. While User logic is being reconfigured, it would unfortunately be possible to make the static logic of Shell unable to operate appropriately. Thus, XBuilder ties the partition pin's wires (including a system bus) by using DFX decoupler IP [83, 89] and makes User logic programming separate from the working logic of Shell. In default, XBuilder implements Shell by locating an out-of-core processor and PCIe/memory subsystems that run GraphRunner and GraphStore. Figure 12 shows three example implementation views of our Shell and User logic. Shell logic locates an out-of-core processor and PCIe/memory subsystems that run GraphRunner and GraphStore. In this example, we program an open-source RISC-V CPU, vector processor, and systolic array. We will explain details of example implementations in Section 5.

5 Evaluation

Prototypes. While CSSD is officially released in storage communities [20, 63, 70], there is no commercially available device yet. We thus prototype a customized CSSD that employs a 14*nm* 730MHz FPGA chip [87, 88], 16GB DDR4-2400 DRAM [71], and a 4TB high-performance SSD [12] together within the same PCIe 3.0×4 subsystem [57] as shown in Figure 13. We prepare three sets of hardware accelerators for XBuilder's User logic; a multi-core processor (*Octa-HGNN*), large systolic array processors (*Lsap-HGNN*), and a heterogeneous accelerator having a vector processor and a systolic array (*Hetero-HGNN*), as shown in Figure 12. Octa-HGNN employs eight out-of-order (O3) cores and performs all GNN processing using multi-threaded software. Each O3 core is implemented based on open-source RISC-V [3, 100] having 160KB L1 and 1MB L2 caches. For Lsap-HGNN and Hetero-HGNN, we modify an open-source SIMD (Hwacha [47]) and systolic architecture (Gemmini [23]). In our evaluation, SIMD employs four vector units, and the systolic architecture employs 64 floating-point PEs with 128KB scratchpad memory. Note that, all these prototypes use the same software part of HolisticGNN (GraphStore, GraphRunner, and XBuilder) as it can handle the end-to-end GNN services over DFG.

GPU-acceleration and testbed. For a fair performance com-

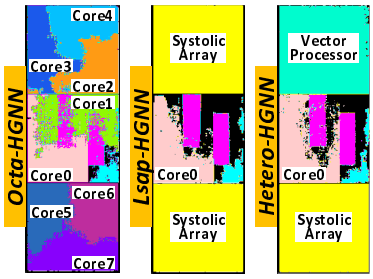


Figure 12: Shell/User prototypes.

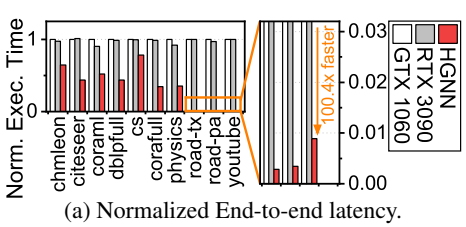
Figure 13: HolisticGNN prototype.

Host Setup
AMD Ryzen 3900X 2.2GHz, 12 cores DDR4-2666 16GB x4
GTX 1060 6GB [13] 1.8GHz, 1024 cores (10 SMs)
RTX 3090 24GB [14] 1.74GHz, 10496 cores (82 SMs)
FPGA Setup
Xilinx Virtex UltraScale+ [87] DDR4-2400 16GB x2
Storage
Intel SSD DC P4600 [12] 3D TLC NAND, 4TB

Table 4: Host and FPGA setup.

	Legend	Original Graph			Sampled Graph		
		Vertices	Edges	Feature Size	Vertices	Edges	Feature Length
Small (< 1M Edges)	chmleon [61]	2.3K	65K	20 MB	1,537	7,100	2326
	citeseer [93]	2.1K	9K	29 MB	667	1,590	3704
	coraml [93]	3.0K	19K	32 MB	1,133	2,722	2880
	dblpfull [93]	17.7K	123K	110 MB	2,208	3,784	1639
	cs [66]	18.3K	182K	475 MB	3,388	6,236	6805
	corafull [93]	19.8K	147K	657 MB	2,357	4,149	8710
	physics [66]	34.5K	530K	1,107 MB	4,926	8,662	8415
Large (>3M Edges)	road-tx [48]	1.39M	3.84M	23.1 GB	517	904	4353
	road-pa [48]	1.09M	3.08M	18.1 GB	580	1,010	4353
	youtube [48]	1.16M	2.99M	19.2 GB	1,936	2,193	4353
	road-ca [48]	1.97M	5.53M	32.7 GB	575	999	4353
	wikitalk [48]	2.39M	5.02M	39.8 GB	1,768	1,826	4353
	ljournal [48]	4.85M	68.99M	80.5 GB	5,756	7,423	4353

Table 5: Graph dataset characteristics.



(a) Normalized End-to-end latency.

Legend	GTX 1060
chmleon	140 ms
citeseer	162 ms
coraml	166 ms
dblpfull	323 ms
cs	618 ms
corafull	1233 ms
physics	2335 ms
road-tx	426732 ms
road-pa	332391 ms
youtube	341035 ms

(b) Latency.

Figure 14: End-to-end latency comparison.

parison, we also prepare two high-performance GPUs, *GTX 1060* and *RTX 3090*. While GTX 1060's 10 streaming multiprocessors (SMs) operate at 1.8GHz with 6GB DRAM, RTX 3090 employs 82 SMs working at 1.7 GHz with 24GB DRAM. To enable GNN services, we use deep graph library (DGL) 0.6.1 [74] and TensorFlow 2.4.0 [1], which use CUDA 11.2 and cuDNN 8.2 for GPU acceleration. DGL accesses the underlying SSD via the XFS file system to pre-/processing graphs. The testbed uses a 2.2GHz 12-core processor with DDR4-2666 64GB DRAM and a 4TB SSD (same with the device that we used for CSSD prototype), and connect all GPUs and our CSSD prototype. The detailed information of our real evaluation system is shown in Table 4.

GNN models and graph datasets. We implement three popular GNN models, GCN [42], GIN [90], and NGCF [75], for both GPUs and CSSD. We also select 14 real-graph datasets (workloads) from LBC [45], MUSAE [61], and SNAP [48]. Since the workloads coming from SNAP [48] do not provide the features, we generate the features based on the feature length that the prior work (pinSAGE [95]) uses (4K). The important characteristics of our graph datasets and workloads are described in Table 5. Note that, the workloads that we listed in Table 4 is sorted in ascending order of their graph size. For better understanding, we summarize the characteristics for graph before batch preprocessing (*Original Graph*) and after batch preprocessing (*Sampled Graph*).

5.1 End-to-end Performance Comparisons

Overall latency. Figure 14a compares the end-to-end inference latency of GTX 1060, RTX 3090, and our HolisticGNN (*HGNN*) using the heterogeneous hardware acceleration. For better understanding, the end-to-end latency of RTX 3090 and HGNN is normalized to that of GTX 1060. The actual latency value of GTX 1060 is also listed in Table 14b. We use GCN as representative of GNN models for the end-to-end performance analysis; since we observed that the pure inference computing latency only accounts for 1.8% of total latency, the performance difference among the GNN models that we tested are negligible in this analysis (<1.1%). We will show the detailed inference latency analysis on the different GNN models in Section 5.2.

One can observe from the figure that HGNN shows 7.1× and 7.0× shorter end-to-end latency compared to GTX 1060 and RTX 3090 across all the graph datasets except for `road-ca`, `wikitalk`, and `ljournal`. Note that both GTX 1060 and RTX 3090 cannot execute such large-scale graphs

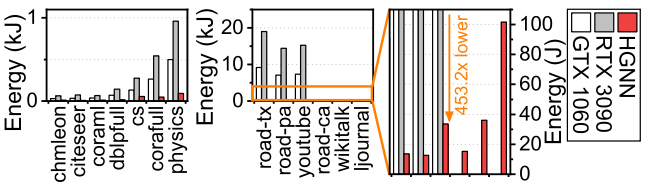


Figure 15: Estimated energy consumption comparison.

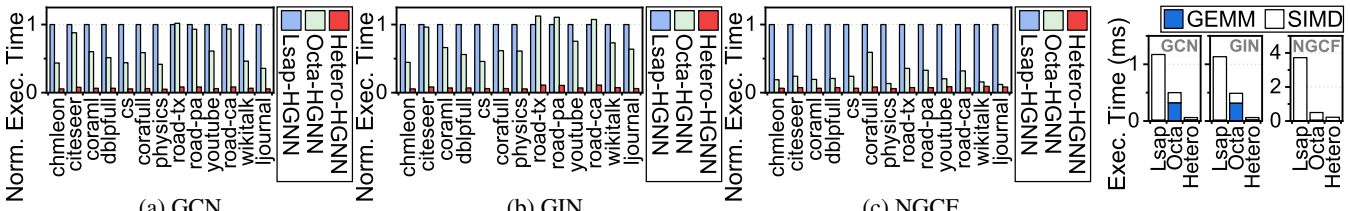
due to the out-of-memory issue, and thus, we exclude them in this comparison. Specifically, for the small graphs (<1M edges), HGNN outperforms GPUs by 1.69×, on average. This performance superiority of HGNN becomes higher when we infer features on large-scale graphs (>3M edges), which makes HGNN 201.4× faster than GTX 1060 and RTX 3090, on average. Even though the operating frequency and computing power of GTX 1060 and RTX 3090 are much better than HGNN, most of data preprocessing for both graphs and batches are performed by the host, and its computation is involved in storage accesses. This in turn makes the end-to-end inference latency longer. In contrast, HGNN can preprocess graphs in parallel with the graph updates and prepare sampled graphs/embeddings directly from the internal SSD, which can successfully reduce the overhead imposed by preprocessing and storage accesses. We will dig deeper into the performance impact of preprocessing/storage (GraphStore) and hardware accelerations (XBuilder) shortly.

Energy consumption. Figure 15 analyzes the energy consumption behaviors of all three devices we tested. Even though GTX 1060 and RTX 3090 show similar end-to-end latency behaviors in the previous analysis, RTX 3090 consumes energy 2.04× more than what GTX 1060 needs because it has 8.2× and 4× more SMs and DRAM, respectively. In contrast, HGNN exhibits 33.2× and 16.3× better energy consumption behaviors compared to RTX 3090 and GTX 1060, on average, respectively. Note that, HGNN processes large-scale graphs by consuming as high as 453.2× less energy than the GPUs we tested. This is because, in addition to the latency reduction of HGNN, our CSSD consumes only 111 Watts at the system-level thanks to the low-power computing of FPGA (16.3 Watts). This makes HGNN much more promising on GNN computing compared to GPU-based acceleration approaches. Note that, RTX 3090 and GTX 1060 consume 214 and 447 Watts at the system-level, respectively.

5.2 Pure Inference Acceleration Comparison

Figure 16 shows the pure inference performance of Hetero-HGNN and Octa-HGNN, normalized to Lsap-HGNN; before analyzing the end-to-end service performance, we first compare HolisticGNN itself different User logic here.

One can observe from this figure that, even though systolic arrays are well optimized for conventional DL such as CNN and RNN, Lsap-HGNN exhibits much worse performance than software-only approach. For all the graph datasets that we tested, Octa-HGNN exhibits shorter GNN inference latency compared to Lsap-HGNN by 2.17×, on average. This is crystal clear evidence that the conventional DL hardware



(a) GCN. (b) GIN. (c) NGCF.

Figure 16: Inference latency comparison.

Figure 17: Breakdown.

acceleration is not well harmonized with GNN inference services. Since the computation of aggregation is involved in traversing the graph data, the systolic arrays (bigger than any hardware logic that we tested) cannot unfortunately accelerate the inference latency. In contrast, Octa-HGNN processes the aggregation (including transformation) over multi-processing with many cores in User logic. As shown in Figure 16c, this phenomenon is more notable on the inference services with NGCF (4.35× faster than Lsap-HGNN) because NGCF has more heavier aggregation (similarity-aware and element-wise product explained in Section 2.1).

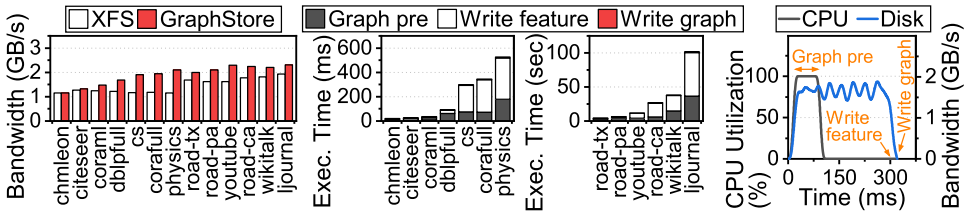
However, the performance of Octa-HGNN is also limited because matrix computation on dense embeddings (GEMM) is not well accelerated by its general cores. In contrast, Hetero-HGNN has both SIMD and systolic array units, which are selectively executed considering the input C-kernel, such that Hetero-HGNN shortens the inference latency of Octa-HGNN and Lsap-HGNN by 6.52× and 14.2×, on average, respectively. Figure 17 decomposes the inference latency of three HGNN that we tested into SIMD and GEMM for a representative workload, physics. As shown in figure, Lsap-HGNN mostly exhibits GEMM as its systolic arrays accelerate the transformation well, but its performance slows down due to a large portion of SIMD. The latency of Octa-HGNN suffers from GEMM computation, which accounts for 34.8% of its inference latency, on average. As Hetero-HGNN can accelerate both SIMD and GEMM, it successfully shortens the aggregation and transformation for all GNN models that we tested. This is the reason why we evaluate the end-to-end GNN latency using Hetero-HGNN as a default hardware acceleration engine in the previous section.

5.3 Performance Analysis on GraphStore

Bulk operations. Figures 18a and 18b show the bandwidth and latency of GraphStore's bulk operations. While the GPU-enabled host system writes the edge array and corresponding embeddings to the underlying SSD through its storage stack, GraphStore directly writes the data to internal storage without any storage stack involvement. This does not even exhibit data copies between page caches and user-level buffers, which in turn makes GraphStore exposes performance closer to what the target SSD actually provides. As a result, GraphStore shows 1.3× better bandwidth on graph updates compared to conventional storage stack (Figure 18a). More importantly, GraphStore hides the graph preprocessing overhead imposed by converting the input dataset to the corresponding adjacency list with the update times of heavy embeddings. We also show how much the embedding update (`Write feature`) can hide the latency of graph preprocessing (`Graph pre`) in Figure 18b. Since `Write feature` in the figure only shows the times longer than `Graph pre`, we can observe that GraphStore can make `Graph pre` completely invisible to users. For better understanding, we also perform a time series analysis of `cs` as an example of other workloads, and the results are shown Figure 18c. The figure shows the dynamic bandwidth in addition to the per-task utilization of Shell's simple core. As shown in the figure, GraphStore starts the preprocessing as soon as it begins to write the embeddings to the internal SSD. `Graph pre` finishes at 100*ms* while `Write feature` ends at 300*ms*. Thus, `Write feature` is performed with the best performance of the internal SSD (around 2GB/s). Note that, even though writing the adjacency list `Write graph` is performed right after `Write feature` (Figure 18b), it is almost invisible to users (Figure 18c) as the size of graph is much smaller than the corresponding embeddings (357.1×, on average).

Batch preprocessing (Get). Figure 19 shows batch preprocessing, which is the only task to read (sub)graphs from the storage in the end-to-end viewpoint; node sampling and embedding lookup use `GetNeighbor()` and `GetEmbed()`, respectively. In this evaluation, we compare batch preprocessing performance of GPU-enabled host and CSSD using `chmleon` and `youtube` each being representative of small and large graphs. For the earliest batch preprocessing, GraphStore performs batch preprocessing 1.7× (`chmleon`) and 114.5× (`youtube`) faster than that of the GPU-enabled host, respectively. Even though GraphStore is working at a lower frequency (3× than the host CPU), `GetNeighbor()` and `GetEmbed()` are much faster because the graph data has been already converted into an adjacency list at the graph update phase. In contrast, the host needs to process the graph data at the first batch, such that node sampling and embedding lookup can find out the appropriate targets. After the first batch, both cases, mostly accessing the neighbors and the corresponding embeddings are processed in memory thereby showing sustainable performance. Note that, even though we showed the batch preprocessing performance for only `chmleon` and `youtube` (due to the page limit), this performance trend is observed across all the workloads that we tested.

Mutable graph support (Unit operations). Since there is no publicly available dataset for mutable graph support, we evaluate the unit operations (requested by the host to CSSD) by processing historical DBLP datasets [30]. The top of Figure 20 shows the number of per-day add and delete operations



(a) Peak write bandwidth. (b) Latency breakdown. (c) Timeline of cs.

Figure 18: Performance analysis of GraphStore bulk operations.

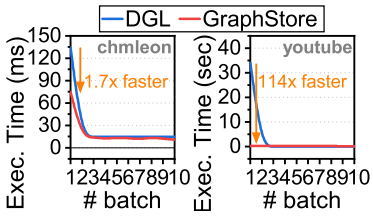


Figure 19: Multiple batches.

for the past 23 years (1995∼2018), and its bottom shows the corresponding per-day (accumulated) latency of GraphStore. The workload adds 365 new nodes and 8.8K new edges into GraphStore, and deletes 16 nodes and 713 edges per day, on average. As shown in Figure, GraphStore exhibits 970*ms* for per-day updates, on average, and the accumulated latency in the worst case of GraphStore is just 8.4 sec, which takes reasonably short in the workload execution time (0.01%).

6 Related Work and Discussion

There are many studies for in-storage processing (ISP) [25, 34, 43, 46, 58, 65], including DL accelerating approaches such as [51, 53, 76]. All these studies successfully brought significant performance benefits by removing data transferring overhead. However, these in-storage, smart storage approaches require fully integrating their computations into an SSD, which unfortunately makes the data processing deeply coupled with flash firmware and limited to a specific computing environment that the storage vendor/device provides. These approaches also use a thin storage interface to communicate with the host and the underlying SSD, which require a significant modification of application interface management. More importantly, all they are infeasible to accelerate GNN computing, which contains both graph-natured processing and DL-like dense computing operations.

On the other hand, architectural research [5, 50, 91] focuses on accelerating GNN core over a fixed hardware design such as vector units and systolic processors. While this simulation-based achieves the great performance benefit on GNN inference, they are ignorant of performance-critical components such as graph preprocessing and node sampling. These simulation-based studies also assume that their accelerator can have tens of hundreds of preprocessing elements (PEs), which may not be feasible to integrate into CSSD because of the hardware area limit. In contrast, HolisticGNN accelerates GNN-related tasks from the beginning to the end near storage, and its real system implementation only contains 64 PEs for the GNN inference acceleration.

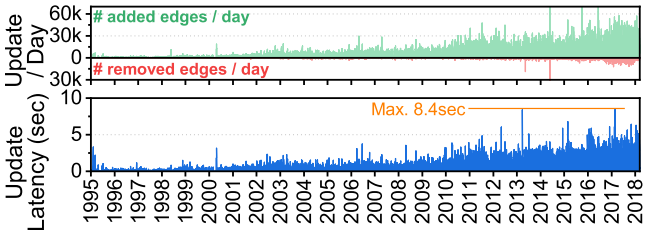


Figure 20: GraphStore update performance.

Lastly, there are FPGA approaches to deep learning accelerations [26, 77, 97]. Angel-Eye [26] quantizes data to compress the original network to a fixed-point form and decrease the bit width of computational parts. A frequency-domain hybrid accelerator [97] applies discrete Fourier transformation methods to reduce the number of multiplications of convolutions. On the other hand, a reconfigurable processing array design [77] tries to increase the operating frequency of any target FPGA in order to build a high throughput reconfigurable processing array. These studies are unfortunately not feasible to capture the GNN acceleration, and cannot eliminate the preprocessing overhead imposed by graph-natured complex computing near storage. Note that, it would be possible to use cross-platform abstraction platforms, such as OpenCL [69] or SYCL [60], rather than using RPC. OpenCL/SYCL is excellent for managing all hardware details at a very low-level, but they can bump up the complexity of what users need to control. For example, users should know all heterogeneities of reconfigurable hardware for the end-to-end GNN acceleration and handle CSSD's memory space over OpenCL/SYCL.

7 Conclusion

We propose HolisticGNN that provides an easy-to-use, near-storage inference infrastructure for fast, energy-efficient GNN processing. To achieve the best end-to-end latency and high energy efficiency, HolisticGNN allows users to implement various GNN algorithms close to the data source and execute them directly near storage in a holistic manner. Our empirical evaluations show that the inference time of HolisticGNN outperforms GNN inference services using high-performance modern GPUs by 7.1× while reducing energy consumption by 33.2×, on average.

Acknowledgements

This research is supported by Samsung Research Funding & Incubation Center of Samsung Electronics (SRFC-IT2101-04). This work is protected by one or more patents, and Myoungsoo Jung is the corresponding author. The authors would like to thank the anonymous reviewers for their comments and suggestions. The authors also thank Raju Rangaswami for shepherding this paper.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th Operating Systems Design and Implementation (OSDI)*, 2016.

[2] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[3] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[4] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley, 2016.

[5] Adam Auten, Matthew Tomei, and Rakesh Kumar. Hardware Acceleration of Graph Neural Networks. In *57th Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[6] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, 34(4), 2014.

[7] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. Relational Inductive Biases, Deep Learning, and Graph Networks. *arXiv preprint arXiv:1806.01261*, 2018.

[8] Chi Chen, Weike Ye, Yunxing Zuo, Chen Zheng, and Shyue Ping Ong. Graph Networks as a Universal Machine Learning Framework for Molecules and Crystals. *Chemistry of Materials*, 31(9):3564–3572, 2019.

[9] Jie Chen, Tengfei Ma, and Cao Xiao. FastGCN: Fast Learning with Graph Convolutional Networks via Importance Sampling. *arXiv preprint arXiv:1801.10247*, 2018.

[10] Wonil Choi, Mohammad Arjomand, Myoungsoo Jung, and Mahmut Kandemir. Exploiting Data Longevity for Enhancing the Lifetime of Flash-based Storage Class Memory. In *2017 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2017.

[11] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[12] Intel Corporation. Intel SSD DC P4600 Series. https://ark.intel.com/content/www/us/en/ark/products/96998/intel-ssd-dc-p4600-series-4-0tb-12-height-pcie-3-1-x4-3d1-tlc.html.

[13] NVIDIA Corporation. GeForce GTX 1060. https://www.nvidia.com/en-in/geforce/products/10series/geforce-gtx-1060/.

[14] NVIDIA Corporation. GeForce RTX 3090. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3090/.

[15] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Advances in Neural Information Processing Systems 29 (NIPS)*, 2016.

[16] Google Developers. Protocol Buffers. https://developers.google.com/protocol-buffers.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using Non-volatile Memory for Storing Deep Learning Models. *arXiv preprint arXiv:1811.05922*, 2018.

[19] Chantat Eksombatchai, Pranav Jindal, Jerry Zitao Liu, Yuchen Liu, Rahul Sharma, Charles Sugnet, Mark Ulrich, and Jure Leskovec. Pixie: A system for recommending 3+ billion items to 200+ million users in

real-time. In *Proceedings of the 2018 world wide web conference*, 2018.

[20] Samsung Electronics. Samsung SmartSSD. https://samsungsemiconductor-us.com/smartssd-archive/pdf/SmartSSD_ProductBrief_13.pdf.

[21] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference*, pages 417–426, 2019.

[22] Matthias Fey and Jan E. Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[23] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *58th Design Automation Conference (DAC)*, 2021.

[24] gRPC Authors. gRPC. https://grpc.io/.

[25] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *43rd International Symposium on Computer Architecture (ISCA)*, 2016.

[26] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2017.

[27] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30 (NIPS)*, pages 1025–1035, 2017.

[28] William L Hamilton, Rex Ying, and Jure Leskovec. Representation Learning on Graphs: Methods and Applications. *arXiv preprint arXiv:1709.05584*, 2017.

[29] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation. In *43rd International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 639–648, 2020.

[30] Oliver Hoffmann and Florian Reitz. hdblp: Historical Data of the DBLP Collection. https://doi.org/10.5281/zenodo.3051910, May 2019.

[31] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 1991.

[32] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 1989.

[33] Abhinav Jangda, Sandeep Polisetty, Arjun Guha, and Marco Serafini. Accelerating Graph Sampling for Graph Machine Learning using GPUs. In *16th European Conference on Computer Systems (EuroSys)*, pages 311–326, 2021.

[34] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. YourSQL: A High-performance Database System Leveraging In-storage Computing. *2016 Proceedings of the VLDB Endowment (PVLDB)*, 9(12):924–935, 2016.

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

[36] Hoeseung Jung, Sanghyuk Jung, and Yong Ho Song. Architecture Exploration of Flash Memory Storage Controller through a Cycle Accurate Profiling. *IEEE Transactions on Consumer Electronics*, 57(4):1756–1764, 2011.

[37] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Taylan Kandemir. Design of a host interface logic for gc-free ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[38] Myoungsoo Jung, Ramya Prabhakar, and Mahmut Taylan Kandemir. Taking garbage collection overheads off the critical path in SSDs. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (USENIX Middleware)*, 2012.

[39] Kang, Yangwook and Kee, Yang-suk and Miller, Ethan L. and Park, Chanik. Enabling cost-effective data processing with smart ssd. In *29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, 2013.

[40] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 2011.

[41] Kimberly Keeton, David A Patterson, and Joseph M Hellerstein. A case for intelligent disks (IDISKs). *ACM SIGMOD Record*, 27(3):42–52, 1998.

[42] Thomas N Kipf and Max Welling. Semi-supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907*, 2016.

[43] Gunjae Koo, Kiran Kumar Matam, I Te, HV Krishna Giri Narra, Jing Li, Hung-Wei Tseng, Steven Swanson, and Murali Annavaram. Summarizer: Trading Communication with Computing Near Storage. In *50th International Symposium on Microarchitecture (MICRO)*, pages 219–231. IEEE, 2017.

[44] Dongup Kwon, Dongryeong Kim, Junehyuk Boo, Wonsik Lee, and Jangwoo Kim. A Fast and Flexible Hardware-based Virtualization Mechanism for Computational Storage Devices. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 729–743, 2021.

[45] Machine learning research group from the University of Maryland. Link-based classification project.

[46] Young-Sik Lee, Luis Cavazos Quero, Youngjae Lee, Jin-Soo Kim, and Seungryoul Maeng. Accelerating External Sorting via On-the-fly Data Merge in Active SSDs. In *6th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[47] Yunsup Lee, Albert Ou, Colin Schmidt, Sagar Karandikar, Howard Mao, and K Asanovic. The Hwacha Microarchitecture Manual, Version 3.8. Technical Report UCB/EECS-2015-263, EECS Department, University of California, Berkeley, 2015.

[48] Jure Leskovec and Rok Sosič. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1, 2016.

[49] Jia Li, Yu Rong, Hong Cheng, Helen Meng, Wenbing Huang, and Junzhou Huang. Semi-supervised Graph Classification: A Hierarchical Graph Perspective. In *The World Wide Web Conference*, pages 972–982, 2019.

[50] Shengwen Liang, Ying Wang, Cheng Liu, Lei He, LI Huawei, Dawen Xu, and Xiaowei Li. EnGN: A High-Throughput and Energy-Efficient Accelerator for Large Graph Neural Networks. *IEEE Transactions on Computers (TC)*, 2020.

[51] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 395–410, 2019.

[52] Xuan Lin, Zhe Quan, Zhi-Jie Wang, Tengfei Ma, and Xiangxiang Zeng. KGNN: Knowledge Graph Neural Network for Drug-Drug Interaction Prediction. In *29th International Joint Conference on Artificial Intelligence (IJCAI)*, volume 380, pages 2739–2745, 2020.

[53] Vikram Sharma Mailthody, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia De Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen-mei Hwu. DeepStore: In-storage Acceleration for Intelligent Queries. In *52nd International Symposium on Microarchitecture (MICRO)*, pages 224–238, 2019.

[54] Diego Marcheggiani and Ivan Titov. Encoding Sentences with Graph Convolutional Networks for Semantic Role Labeling. *arXiv preprint arXiv:1703.04826*, 2017.

[55] Muthukumar Murugan and David HC Du. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. In *27th International Conference on Mass Storage Systems and Technologies (MSST)*, 2011.

[56] Yangyang Pan, Guiqiang Dong, and Tong Zhang. Exploiting Memory Device Wear-Out Dynamics to Improve NAND Flash Memory System Performance. In *9th Conference on File and Storage Technologies (FAST)*, volume 11, pages 18–18, 2011.

[57] PCI-SIG. PCI Express Base Specification Revision 3.1a. https://members.pcisig.com/wg/PCI-SIG/document/download/8257.

[58] Luis Cavazos Quero, Young-Sik Lee, and Jin-Soo Kim. Self-sorting SSD: Producing sorted data inside active SSDs. In *31st International Conference on Mass Storage Systems and Technologies (MSST)*, 2015.

[59] Hannu Reittu and Ilkka Norros. On the power-law random graph model of massive data networks. *Performance Evaluation*, 2004.

[60] Ruyman Reyes and Victor Lomüller. Sycl: Single-source c++ accelerator programming. In *Parallel Computing: On the Road to Exascale*. IOS Press, 2016.

[61] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale Attributed Node Embedding. *Journal of Complex Networks*, 9(2):cnab014, 2021.

[62] Zhenyuan Ruan, Tong He, and Jason Cong. Insider: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, 2019.

[63] ScaleFlux. ScaleFlux Computational Storage. https://www.scaleflux.com/.

[64] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2008.

[65] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A User-Programmable SSD. In *11st USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[66] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.

[67] SiFive. TileLink Specification. https://sifive.cdn.prismic.io/sifive/7bef6f5c-ed3a-4712-866a-1a2e0c6b7b13_tilelink_spec_1.8.1.pdf.

[68] SNIA. Computational Storage. https://www.snia.org/computational.

[69] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 2010.

[70] NGD Systems. NGD Systems Newport Platform. https://www.ngdsystems.com/technology/computational-storage.

[71] Micron Technology. MTA18ASF2G72PZ. https://www.micron.com/products/dram-modules/rdimm/part-catalog/mta18asf2g72pz-2g3.

[72] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph Attention Networks. *arXiv preprint arXiv:1710.10903*, 2017.

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

[74] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315*, 2019.

[75] Xiang Wang, Xiangnan He, Meng Wang, Fuli Feng, and Tat-Seng Chua. Neural Graph Collaborative Filtering. In *42nd International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 165–174, 2019.

[76] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. RecSSD: Near Data Processing for Solid State Drive based Recommendation Inference. In *26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 717–729, 2021.

[77] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. A High-Throughput Reconfigurable Processing Array for Neural Networks. In *27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

[78] Guanying Wu and Xubin He. Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality. In *7th Eeuropean Conference on Computer Systems (EuroSys)*, pages 253–266, 2012.

[79] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2020.

[80] Xilinx. Computational Storage. https://www.xilinx.com/applications/data-center/computational-storage.html.

[81] Xilinx. Design Checkpoints. https://www.rapidwright.io/docs/Design_Checkpoints.html.

[82] Xilinx. Dynamic Function eXchange. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_1/ug909-vivado-partial-reconfiguration.pdf.

[83] Xilinx. Dynamic Function eXchange Decoupler. https://www.xilinx.com/content/dam/xilinx/support/documentation/ip_documentation/dfx_decoupler/v1_0/pg375-dfx-decoupler.pdf.

[84] Xilinx. SmartSSD Computational Storage Drive. https://www.xilinx.com/content/dam/xilinx/support/documentation/boards_and_kits/accelerator-cards/1_2/ug1382-smartssd-csd.pdf.

[85] Xilinx. UltraScale Architecture Configuration. https://www.xilinx.com/support/documentation/user_guides/ug570-ultrascale-configuration.pdf.

[86] Xilinx. UltraScale Architecture Libraries Guide. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2021_1/ug974-vivado-ultrascale-libraries.pdf.

[87] Xilinx. Virtex UltraScale+ FPGA. https://www.xilinx.com/content/dam/xilinx/support/documentation/product-briefs/virtex-ultrascale-product-brief.pdf.

[88] Xilinx. Virtex UltraScale+ FPGA Data Sheet: DC and AC Switching Characteristics. https://www.xilinx.com/support/documentation/data_sheets/ds923-virtex-ultrascale-plus.pdf.

[89] Xilinx. Vivado Design Suite Tutorial: Dynamic Function eXchange. https://www.xilinx.com/content/dam/xilinx/support/documentation/sw_manuals/xilinx2020_2/ug947-vivado-partial-reconfiguration-tutorial.pdf.

[90] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How Powerful are Graph Neural Networks? *arXiv preprint arXiv:1810.00826*, 2018.

[91] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. HyGCN: A GCN Accelerator with Hybrid Architecture. In *26th International Symposium on High Performance Computer Architecture (HPCA)*, pages 15–29. IEEE, 2020.

[92] Ke Yang, MingXing Zhang, Kang Chen, Xiaosong Ma, Yang Bai, and Yong Jiang. Knightking: a fast distributed graph random walk engine. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[93] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. Revisiting Semi-supervised Learning with Graph Embeddings. In *33rd International Conference on Machine Learning (ICML)*, pages 40–48. PMLR, 2016.

[94] Liang Yao, Chengsheng Mao, and Yuan Luo. Graph Convolutional Networks for Text Classification. In *33rd AAAI Conference on Artificial Intelligence (AAAI)*, pages 7370–7377, 2019.

[95] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. Graph Convolutional Neural Networks for Web-scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 974–983, 2018.

[96] Jiaxuan You, Bowen Liu, Rex Ying, Vijay Pande, and Jure Leskovec. Graph Convolutional Policy Network for Goal-directed Molecular Graph Generation. *arXiv preprint arXiv:1806.02473*, 2018.

[97] Chi Zhang and Viktor Prasanna. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *2017 International Symposium on Field-Programmable Gate Arrays (FPGA)*, pages 35–44, 2017.

[98] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018.

[99] Jie Zhang, Miryeong Kwon, Michael Swift, and Myoungsoo Jung. Scalable Parallel Flash Firmware for Many-core Architectures. In *18th Conference on File and Storage Technologies (FAST)*, 2020.

[100] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*, May 2020.

[101] Ling Zhao, Yujiao Song, Chao Zhang, Yu Liu, Pu Wang, Tao Lin, Min Deng, and Haifeng Li. T-GCN: A Temporal Graph Convolutional Network for Traffic Prediction. *IEEE Transactions on Intelligent Transportation Systems*, 21(9):3848–3858, 2019.