BeaconGNN: Large-Scale GNN Acceleration with Out-of-Order Streaming In-Storage Computing

Yuyue Wang[1], Xiurui Pan[2], Yuda An[2], Jie Zhang[2†], Glenn Reinman[1†]
[1]*UCLA*, [2]*Peking University*
yuyue@cs.ucla.edu, jiez@pku.edu.cn, reinman@cs.ucla.edu

Abstract—**Prior in-storage computing (ISC) solutions show fundamental drawbacks when applied to GNN acceleration. First, they obey a strict ordering of GNN neighbor sampling. Such serialization fails to utilize flash internal parallelism. Second, the I/O sizes generated by GNN are much smaller than the minimum flash access granularity. The limited channel bandwidth is wasted when serving the requests. Third, the prior solutions rely on firmware-based request processing, making the backend I/O throughput constrained by the embedded core processing power.**

To address these challenges, we propose BeaconGNN, an in-storage computing (ISC) design for GNN that supports both large-scale graph structures and feature tables. First, it utilizes a novel graph format to enable out-of-order GNN neighbor sampling, improving flash resource utilization. Second, it deploys near-data processing engines across multiple levels of the flash hierarchy (i.e., controller, channel, and die). Specifically, flash-die-level samplers perform neighbor samplings while reducing channel transfer simultaneously. Flash-channel-level command routers communicate with backend dies without the involvement of flash firmware. Lastly, a spatial accelerator is attached to the device bus to accelerate GNN computation. With our software and hardware co-design, BeaconGNN achieves up to $11.6\times$ higher throughput and $4\times$ better energy efficiency than the state-of-the-art ISC design.

I. Introduction

Recently, Graph Neural Networks (GNNs) have become increasingly prevalent in multiple computing domains such as social networks, recommendation systems, and pandemic prediction [14], [16], [57], [64], [72], [76]. This is because GNNs excel at capturing complex relationships and dependencies inherent in large-scale graph-structured data.

The GNN task consists of two main stages: *data preparation* and *GNN computation*. During the data preparation stage, nodes are sampled from the graph and their corresponding vectors are retrieved from the feature table, both of which are passed to the subsequent computation stage. In real-world industrial scenarios, the sizes of these graphs and feature tables scale to over hundreds of GBs or even reach TBs [11], [43], [82], surpassing the memory capacity of a single machine. This motivates system designers to accommodate the huge amount of data in high-volume and cost-effective storage such as solid-state drives (SSDs) [41], [42], [54], [58], [70]. Furthermore, the GNN computation stage performs highly parallelizable aggregation and update operations on graph embeddings. This has led to the widespread adoption of hardware acceleration, such as GPUs and ASICs.

† Jie Zhang and Glenn Reinman are corresponding authors.

Unfortunately, as GNN data initially resides in the storage, the data preparation stage incurs intensive data transfer between storage and processors (i.e., CPU and GPU), which imposes huge burdens on the traditional computing system [79]–[81]. In particular, the thin PCIe bus narrows down the I/O bandwidth while the software stack of storage and GPUs introduces redundant data copies and multiple address translations [24]. Moreover, the I/O access latency in SSDs is still considerably higher than that of the main memory, resulting in a significant portion of the total latency being attributed to the data preparation.

To address these challenges, recent advances in storage technologies have provided promising solutions. One such advancement is in-storage computing (ISC), which leverages SSD-embedded processors (i.e., low-end general-purpose processors) to process simple tasks [20], [38], [62] or incorporates FPGAs/ASICs to accelerate heavy computations [27], [52], [60], [68]. This technology eliminates expensive data transfer, which breaks the I/O bottleneck imposed by the thin PCIe bus. Another emerging technique, called ultra-low latency (ULL) flash, significantly reduces the SSD read (sense) latency to 3 μs [36], [37]. These two techniques present great opportunities to revive architectural designs for higher GNN performance.

Various prior studies have explored the feasibility of leveraging ISC to accelerate GNN tasks [39], [40], [44]. Their common objective is to identify the I/O-intensive part of the GNN task and offload it to in-storage compute engines. Specifically, GList [44] optimizes operations associated with the GNN feature table. It fetches the feature vectors from flash and sends them to an SSD-internal FPGA for computation. As vectors are processed within the SSD, PCIe traffic is eliminated. In a separate study, SmartSage [40] targets the in-storage graph structure by using the SSD-embedded processor to conduct neighbor sampling. This filters out unused nodes, thereby reducing the number of nodes transmitted back to the host. Despite the success in reducing I/O traffic, prior work has fundamental design drawbacks that prevent them from fully utilizing the potential benefits of ISC. To make matters worse, these weaknesses also become obstacles to optimizing the GNN performance by adapting to the emerging ULL flash.

The first drawback is the inefficiency in sampling multiple hops of neighbors. SmartSage manages part of the graph metadata on the host and requires inter-hop communication to be carried out between the host and storage. This communication acts as a barrier that enforces a serialized order of execution between earlier and later hops, ultimately resulting in the under-utilization of flash bandwidth due to a lack of internal parallelism. Second, the GNN data preparation stage exhibits a random and small I/O pattern that does not align well with the page-granular flash channel transfer. This mismatch causes significant read amplification [13] and wasted bandwidth on flash channels, resulting in I/O throughput reduction (in section III, we conduct two experiments to measure the negative impact of the two drawbacks, as shown in Figure 7). Lastly, in conventional SSDs, the backend flash I/O is processed by the flash firmware. However, firmware-based processing is restricted by the capability of the embedded processor and internal DRAM in SSD, and cannot keep up with the accumulated throughput of ULL flash [80].

Based on the above analysis, we gain three insights into designing a better ISC system for GNN. First, the hop-by-hop ordering of data preparation in prior work compromises the internal parallelism of the SSD. One feasible way to address this is to remove the host from the control path and run the entire procedure as a non-intermittent ISC task. Second, conventional page-level channel transfer performs poorly with the small random I/O pattern in the GNN task. We identify that flash dies have abundant area budgets in the control layer [10], where we can place additional logic to filter unused data to improve transfer efficiency. Third, the flash firmware cannot process a substantial volume of flash I/O efficiently, slowing down the overall progress of data preparation. A potential solution lies in implementing customized hardware-based flash I/O control mechanisms to accelerate the processing.

To address the aforementioned challenges, we propose BeaconGNN, an ISC design for GNN that supports both large-scale graph structures and feature tables. In BeaconGNN, we co-design the software (i.e., GNN engine and flash firmware) and hardware (i.e., SSD internal architecture) to accelerate the entire GNN task flow in the SSD. To eliminate the inter-hop host-SSD communication and the ordering constraint between hops of neighbor sampling, we propose DirectGraph, a novel GNN graph format directly indexed with flash physical addresses, with the co-designed host interface and SSD flash firmware to construct the new graph. The SSD controller can now directly locate GNN data in flash without the involvement of the host-side filesystem and NVMe storage stack. To better suit the small random I/O of GNN and exploit the ultra-low latency feature of ULL flash, we deploy near-data processing engines across multiple levels of the flash hierarchy (i.e., controller, channel, and die). Specifically, to eliminate unnecessary channel transfer, we customize the flash-die-level control logic to sample neighbors, retrieve feature vectors, and return only useful data through the channel. To improve the backend-I/O processing efficiency, we implement the control path in hardware. At the flash interface controller, requests to sample more neighbors are forwarded to destination channels and dies without the involvement of the embedded processor. This fully unleashes the high throughput of ULL flash. Finally, we integrate a spatial accelerator into the SSD internal bus to perform vectorized embedding aggregation and GEMM-based

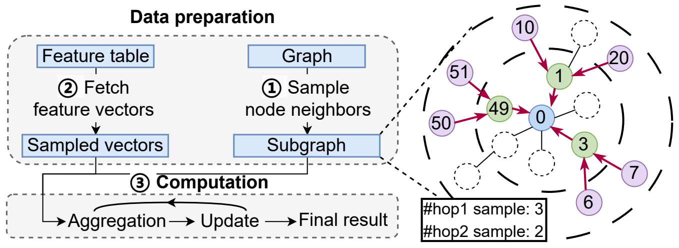


Fig. 1: Typical GNN task flow.

embedding update of the GNN computation stage. With our software and hardware co-design, we achieve up to $27.3\times$, $11.6\times$ overall throughput, and $9.86\times$, $4\times$ energy efficiency compared to the baseline CPU-centric GNN system and state-of-the-art ISC-for-GNN design, respectively.

The main **contributions** of this work are as follows:

1) We observe that the coarse-grain hop-by-hop in-order neighbor sampling and page-granular flash channel transfer are performance bottlenecks of ISC GNN acceleration. And we further observe that firmware-based flash I/O processing is inefficient when adapting ISC-for-GNN to ULL-flash.
2) We propose DirectGraph, a new GNN data format directly indexed with the flash physical address, and develop the algorithm and system support (customized flash request and flash firmware function).
3) We propose die-level processing units to offload neighbor sampling and feature lookup operations onto flash chips.
4) We redesign the flash backend control interface, enabling hardware-based flash I/O processing and streaming-like GNN data preparation.

II. Background

A. Graph Neural Networks

GNNs have been developed in multiple variations to fit various domains [22], [35], [69], [77]. In this paper, we focus on large-scale GNNs with high computation and communication demands. As a representative work of large-scale GNNs, GraphSage [22] effectively reduces both computation and communication burdens via two optimizations: *mini-batch processing* and *neighbor samplings*. Mini-batch processing selects a small batch of nodes as targets and generates corresponding *k-hop subgraphs*. Each k-hop subgraph consists of the target along with neighboring nodes within a maximum distance of k hops. Subsequent computations are confined to these small-scale subgraphs. On the other hand, neighbor sampling selects only a small subset of nodes at each hop when generating subgraphs. This approach controls subgraph scales when graph nodes have a very large number of neighbors.

GNN computation stage is performed through multiple iterations of *message passing*, where each graph node collects the embedding information from its neighboring nodes and updates its own embedding with the gathered information. Suppose that we have a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ along with a set of node features $\mathbf{X} \in \mathbb{R}^{d\times|\mathcal{V}|}$. For each node $u \in \mathcal{V}$, its feature

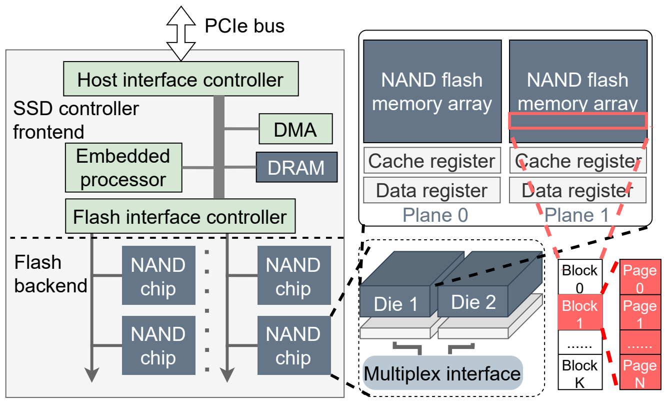


Fig. 2: Overview of the SSD internal architecture.

vector $\mathbf{X}_u$ is used as the initial embedding vector $h_u^{(0)}$. Then at iteration k, we update its embedding vector $h_u^{(k)}$ with the following message-passing rule:

$$h_u^{(k+1)} = \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}) \quad (1)$$

Here $\mathcal{N}_{(u)}$ denotes the neighborhood of node u, that is, the set of nodes that have a distance of 1 hop from node u. AGGREGATE$^{(k)}$is a neural network that aggregates embedding vectors of $\mathcal{N}_{(u)}$ and combines them with its own $h_u^{(k)}$ to generate a new one, $h_u^{(k+1)}$. At the first iteration, every node aggregates structural and feature information from the local neighborhood. As these iterations continue, the aggregation process gradually extends to nodes that are farther away from the starting node. For a GNN with K layers, the final output embedding $h_u^{(K)}$, $u \in \mathcal{V}$ has been updated for K times and encodes the information of the K hop neighbors.

Figure 1 illustrates the decomposition of GNN taskflow. In the first step, we construct a subgraph by sampling multiple hops of neighbors from the target node. In the second step, we fetch node feature vectors from the feature table X. In the last step, we process K iterations of the message passing to aggregate information in the subgraph. After the last iteration, the embedding vector of the target node is the final result.

B. Solid-State Drives

1) Architecture: The left half of Figure 2 shows the typical architecture of commodity solid-state drives (SSDs) [8], [12], [30]. It consists of the SSD controller frontend and the flash backend. At the frontend, an embedded processor runs the flash firmware with a low-end DRAM acting as its main memory. It communicates with the host and flash backend through the host and flash interface controllers, respectively. It also manages the DMA transfer among the flash, DRAM, and host. All of these components are interconnected through an internal I/O bus. At the backend, the flash media is connected to the flash interface controller via multiple flash channels, which allow for parallel data transfer between the frontend and the flash media. Each channel connects multiple flash chips, which can also be accessed simultaneously.

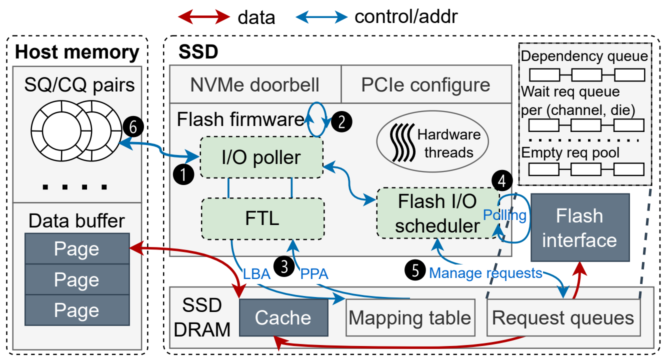


Fig. 3: IO processing in flash firmware.

The flash chip is organized into multiple hierarchies, as shown in the right half of Figure 2. A chip typically contains two flash dies, which multiplex the same I/O path. A die consists of a 3D-stacked NAND flash memory array and the underlying peripheral circuitry. The flash memory array is further organized as multiple blocks, each block containing hundreds of 4KB pages. The erase operation is conducted at the granularity of flash blocks while the minimum unit of read and program operations is a page. Due to flash intrinsics, dirty pages cannot be overwritten before invoking the erase operation [50]. The underlying layer controls these flash operations and manages data and cache registers (i.e., page-size SRAM) to buffer flash pages; it also has a redundant area budget, which provides the opportunity to customize some compute logic for *in-flash* computing [10].

2) I/O processing: Figure 3 shows how the SSD processes host I/O requests [7], [23], [29], [34]. On the data plane, flash firmware manages a cache in the SSD DRAM, which buffers page transfers from the host memory to the flash backend, and vice versa. On the control plane, flash firmware runs hardware threads for three functions to serve host I/O requests: I/O poller, flash translation layer (FTL), and flash I/O scheduler.

A single host I/O request is performed in four steps. First, the I/O poller acquires a new request from the host (❶, ❷). Second, FTL maps the logical page address (LPA) in the request to the physical page address (PPA) (❸). Third, the poller asks the flash I/O scheduler to complete the I/O operation on the backend flash media (❹, ❺). Finally, the I/O poller signals the host that the request is complete (❻).

In the first and last steps, the I/O poller repeatedly checks and updates submission/completion queue entries in the host memory (❶, ❻) [45], [56]. The heads and tails of queues are tracked by NVMe doorbell registers on the host interface (❷). In the second step, the mapping table is stored in DRAM. In the third step, the scheduler processes flash I/O by communicating with the flash interface controller. The scheduler continuously polls flash channel/chip status through the channel control interface, and initiates I/O operations once the necessary hardware resources are ready (❹). To manage ongoing/waiting requests and resolve dependencies among them, the scheduler also maintains several request queues in DRAM for tracking purposes (❺).

Design	In-SSD data	Offload function	Benefits
GList	Feature table	Table lookup; GNN computation	Shorten the transfer of feature vectors
SmartSage	Graph structure	Neighbor sampling	Avoid the transfer of full neighbor lists
Our target	Both	Full-stage (entire GNN task)	Scale well for graph and feature table; adapt to ULL flash

TABLE I: Comparison of prior work and our design target.

C. *ISC for GNN*

There exist multiple ISC approaches that offload different GNN operations to storage [40], [44]. While GList [44] offloads operations related to the feature table, SmartSage [40] offloads operations associated with the graph structure. These approaches are promising to reduce I/O transfers for their specific operations. However, they suffer from certain limitations. First, neither GList nor SmartSage is a full-stage offloading system. Consequently, they are inefficient when dealing with large-scale GNNs where both graph structures and feature tables reside in storage. Second, GList and SmartSage were originally designed for traditional SSDs that have high read latencies. It remains unclear whether they can be effectively adapted to ULL flash.

Table I summarizes the key features of these two prior works and outline our design goal, that is, a full-stage ISC offloading system that efficiently supports large-scale GNNs while leveraging the advantage of ULL flash.

III. Challenge and Motivation

To meet our design target, a straightforward approach is to integrate the offloading solutions from both GList and SmartSage into a unified architecture and utilize ULL flash as the backend media. This combined solution shall be referred to as *BeaconGNN-1.0*.

Figure 4 shows the BeaconGNN-1.0 architecture. The host application controls the execution of the entire GNN task while all data-intensive operations, including neighbor sampling, feature table lookup, and GNN computation, are offloaded to the SSD. Communication between the host and SSD is done via the host-side NVMe driver, in which offloading operations are performed as customized NVMe commands.

For each operation, the associated in-storage data (neighbor lists and embedding vectors) are indexed by multiple layers of address translation. In the host user space, the GNN application is responsible for managing metadata that maps the graph node index to sections in the graph data file. It then calls the file system to get LPAs that point to section locations in the SSD device. Next, LPAs are sent through the PCIe bus to the SSD, where the flash firmware performs the LPA-PPA translation and reads flash pages into the cache in the SSD DRAM. Subsequently, according to the data type, neighbor lists are filtered by the firmware sampler and sent back to the application, while feature vectors are used as inputs to the FPGA kernel for the GNN computation.

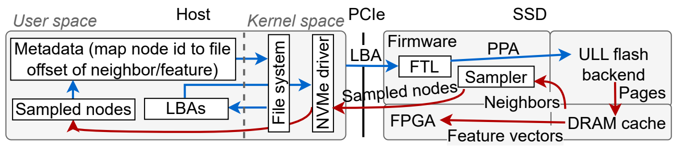
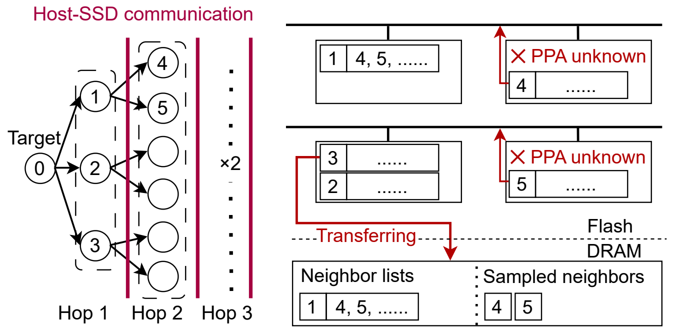


Fig. 4: The architecture of BeaconGNN-1.0.



(a) Generate subgraph for v_0. (b) A moment in the second hop sampling.

Fig. 5: Example of challenge 1.

Challenge 1: Inter-hop sampling barrier

In BeaconGNN-1.0, node neighbors are sampled hop by hop. After completing all samplings in a particular hop, the SSD sends these sampled nodes to the host, which performs the necessary host-side address translation (from node index to LPA) and commands the SSD to sample for the next hop.

Figure 5 (a) illustrates the process of sampling three hops of neighbors. Operations in different hops are separated by barriers at each iteration of host-SSD communication. Figure 5 (b) zooms in the procedure of the second hop: The SSD has sampled v_4, v_5 from $\mathcal{N}(1)$ while still reading $\mathcal{N}(2)$, $\mathcal{N}(3)$. At this moment, flash dies that store $\mathcal{N}(4)$, $\mathcal{N}(5)$ are idle, but their samplings must wait until the current hop and subsequent host-SSD communication are completed. This highlights how the strict hop-by-hop order becomes a barrier that prevents the overlap of neighbor sampling from different hops and adversely affects the parallel utilization of flash dies.

Challenge 2: Page-granular channel transfer

The page-granular channel transfer wastes bandwidth for random I/Os smaller than the page size. This issue is evident in the GNN data-preparation stage, where both neighbor sampling and feature table lookup acquire useful data (sampled nodes, feature vectors) that occupies only a small portion of the page. As a result, the time spent on transferring entire pages significantly prolongs the delay of data preparation.

Figure 6 shows an example in which 4 ULL-flash dies on the same channel are read simultaneously. Initially, every die reads a page in parallel, but later pages are queued at the channel bus, awaiting sequential transfer. This negatively affects flash I/O throughput, especially for ULL flash, whose sequential transfer dominates the total latency. The experiment in Figure 7a gives a vivid demonstration: increasing active ULL-flash dies from 1 to 8 brings merely 49% throughput improvement but undertakes 7.7× average latency.

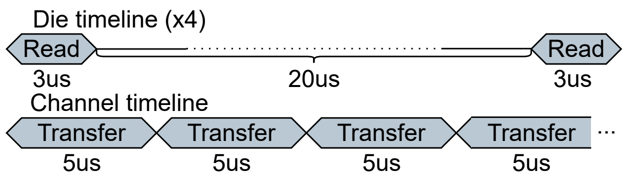
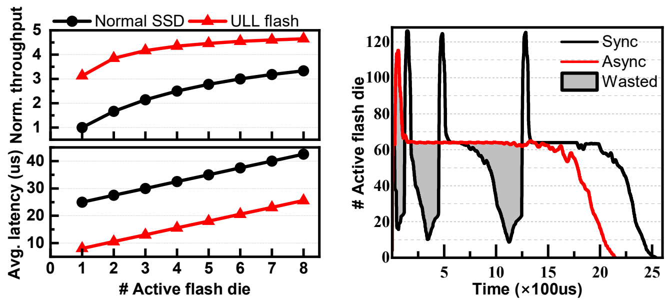


Fig. 6: Example of challenge 2.



(a) Performance of flash requests for normal SSD and ULL-flash.

(b) The overheads of hop-by-hop ordering in multihop data preparation.

Fig. 7: Motivation data.

Challenge 3: Firmware-scheduled flash I/O

Given that the small random I/O is already supported, the bottleneck of flash I/O shifts to the processing capability of the flash firmware owing to three reasons. First is request queue management. In GNN tasks, there is no read-write dependency among flash I/O requests, and thus unnecessary to manage them in DRAM. Second is DMA-configured data transfer between DRAM and NAND flash. In neighbor sampling, the sample address of *(k+1)*th hop depends on the sample result of the *k*th hop. Writing/reading data to/from DRAM delays the time for the *(k+1)*th hop to be ready. Last is polling-based checking for flash status. There are many more channels and dies than processor threads available to perform the polling. Therefore, the processing of a specific flash die will be delayed until its time slot. These factors all together severely affect the backend I/O throughput.

IV. DirectGraph and BeaconGNN-2.0

A. DirectGraph GNN format

Based on the first challenge in Section III, we infer that multi-level address translations enforce the hop-by-hop order of neighbor sampling. However, since most GNN data in real-world scenarios are static without any changes over a long period of time, such translations are unnecessary. To exploit this feature, we propose *DirectGraph*, a GNN format that embeds flash physical addresses directly into the graph to eliminate redundant translation.

Figure 8 illustrates the layout of DirectGraph. We organize the entire GNN graph into two types of pages: *primary* and *secondary pages*, both of which are aligned to the physical flash pages. Each of the primary and secondary pages contains one or more variable-length sections, named *primary* and *secondary sections*, respectively.

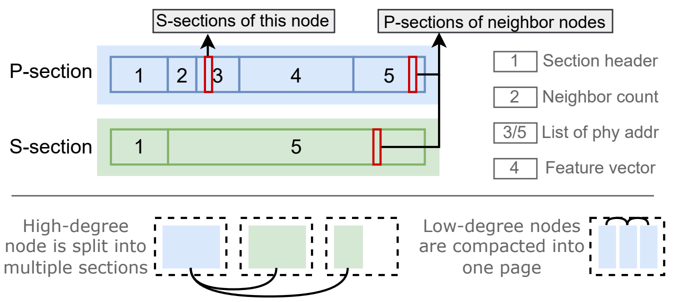

S-sections of this node
P-sections of neighbor nodes
P-section
1
2
3
4
5
S-section
1
5
1 Section header
2 Neighbor count
3/5 List of phy addr
4 Feature vector
High-degree node is split into multiple sections
Low-degree nodes are compacted into one page

Fig. 8: DirectGraph example pages.

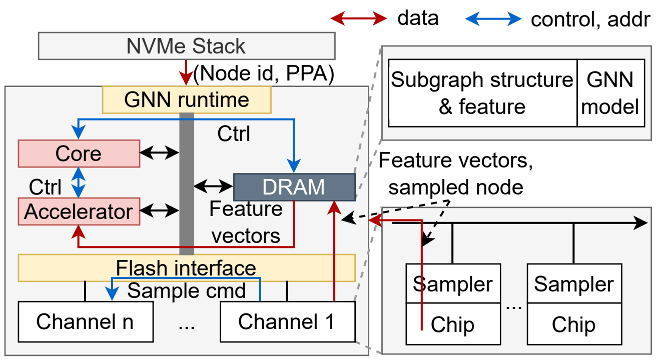


Fig. 9: The architecture of BeaconGNN-2.0.

Each node stores information (i.e., feature vector and neighbor list) in a primary section until the section fulfills an entire primary page. Afterward, excessive neighbors will be placed in secondary sections across one or more secondary pages. It is worth noting that primary sections that store low-degree nodes cannot occupy entire primary pages, resulting in severe space waste. To avoid this, multiple primary sections are organized into a linked array and compacted into a single primary page.

Each section begins with a header to maintain several metadata including its type, length, and node index. Moreover, a primary section records the neighbor count and addresses of associated secondary sections. To locate neighbors without indirection, each neighbor index is mapped to a 4-byte physical address. This address comprises 28 bits for flash page indexing and 4 bits for in-page section indexing in the context of a 1TB SSD with 4KB pages (i.e., $\log_2 \frac{1\text{TB}}{4\text{KB}} = 28$). Using larger pages means more bits can be used for section indexing, and thus more sections can be stored in a single page.

With DirectGraph, the host provides primary section addresses only for target nodes at the start of a mini-batch. Afterward, all subsequent data addressing is done inside the SSD without translation. To read the neighbor list of a node, the page storing its primary section is read first, followed by any pages storing secondary sections. Then sampling takes place, and the results are used to sample the next hop. Similarly, feature vector lookup is performed by reading the flash page containing the primary section of the node.

B. BeaconGNN-2.0

To tame the three challenges in Section III, we propose a new architecture: BeaconGNN-2.0, as illustrated in Figure

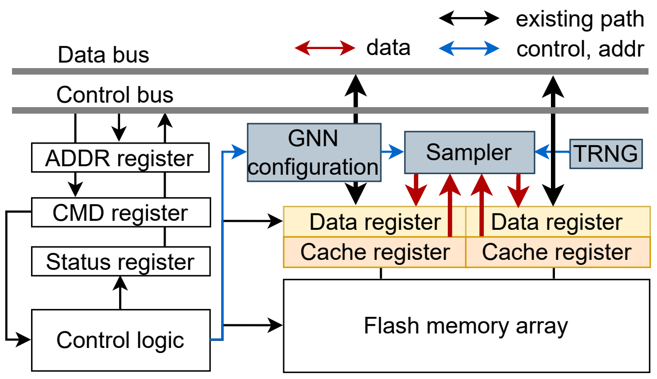


Fig. 10: Flash architecture with die-level hardware sampler.

9. BeaconGNN-2.0 has significant improvements over its predecessor BeaconGNN-1.0. Similarly to BeaconGNN-1.0, it offloads the entire GNN workflow to storage. Nevertheless, it requires minimal host intervention and has more optimization with respect to the I/O efficiency of GNN data preparation.

At the software level, BeaconGNN runs a GNN engine in the flash firmware, which schedules the data preparation and GNN computation stages. It utilizes DirectGraph as the GNN data format to simplify multi-level address translations. Details on how to construct DirectGraph are given in Section VI. DirectGraph also allows BeaconGNN-2.0 to break the inter-hop barriers and sample neighbors out of order.

At the hardware level, BeaconGNN optimizes SSD internal communication pathways for GNN and deploys near-data processing engines at different levels of storage organization, such as flash dies, flash channels, and the SSD controller. To be specific, first, at the flash-die level, a sampler performs neighbor sampling and feature vector retrieval. It then generates new flash commands for subsequent sampling operations near the SRAM buffer. Second, at the flash-channel level, an inter-channel router is added to the flash interface controller, which extracts sampling commands from the channel data stream and forwards them to destinations. These two designs ensure that the data preparation is handled entirely inside the flash backend. Third, at the SSD-controller level, a spatial accelerator is attached to the SSD internal bus, which performs the embedding aggregations and updates to compute the final result. The subgraphs and sampled feature vectors, together with model parameters, are stored inside the SSD DRAM.

V. Multi-Level Near-Data Processing Engines

A. Die-level sampler

Figure 10 shows the functional diagram of a two-plane flash die. Our processing logic is implemented in the control circuitry associated with the flash memory array.

The processing logic comprises a set of GNN configuration registers, a sampler, and a true random number generator (TRNG), all shared by two planes. The existing control logic is also modified to receive command parameters from the data bus and store them in configuration registers; it is also responsible for initiating the execution of the sampler. The sampler, configured by the GNN configuration registers, operates with data read from cache registers and written to data registers through two one-directional buses, respectively. Finally, the TRNG generates random numbers for random samplings.

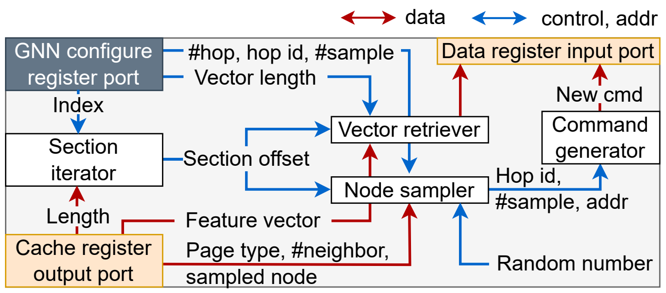


Fig. 11: Microarchitecture of the die-level sampler.

Our die supports two new commands, a global GNN configuration command, which sets basic GNN configurations for each die before the task begins; a sampling command, which reads a page and performs neighbor sampling on the page. The configuration command sets the number of hops, the sample count per hop, and the length of the feature vector; while the essential elements to reconstruct the subgraph (i.e., hop id, sampling count, and metadata) are offered by the sampling command at runtime. At sampling runtime, the specified page is read from the flash memory array to the cache register, from where the sampler samples neighbors, generates new sampling commands and writes them to the data register. In addition to this general procedure, there are two special cases. First, the sampler needs to retrieve feature vectors for primary pages; second, the sampler stops sampling more neighbors at the final hop. When the above process finishes, the data register content transfers off the die over the data bus.

As illustrated in 11, the microarchitecture of the sampler includes four components: a section iterator, a vector retriever, a node sampler, and a command generator. The section iterator traverses the page until it reaches the target section and then transmits the section offset to both the vector retriever and node sampler. The vector retriever transfers the feature vector from the cache register to the data register. The node sampler works differently for primary sections and secondary sections. For primary sections, neighbors are sampled from the entire neighbor range, including those stored in secondary sections. When the sampling result falls within the page, the sampler samples this neighbor, and generates a command to sample for this neighbor; otherwise, the sampler generates a command to sample from the secondary section that the sampling result falls into (all commands for the same secondary section will coalesce later to avoid redundant reads). For secondary sections, neighbors are sampled only from the section itself. Each sampling result is generated with a modulo operation using a TRNG-generated random number.

B. Channel-level command router

At the flash interface controller, we customize its logic to route sampling commands among all flash channels. Each

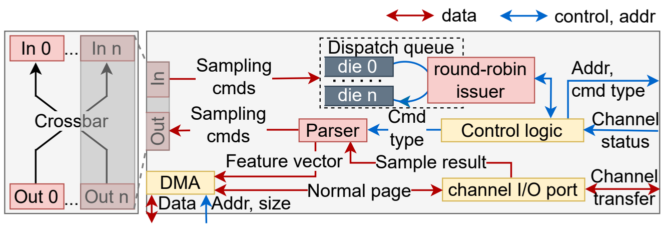


Fig. 12: Channel-level command router.

channel is customized with dispatch queues, a round-robin command issuer, a data stream parser, and I/O ports connected by a crossbar to other channels. The customized logic for one channel is shown in Figure 12.

The channel can be controlled in two ways: traditional firmware-based control and new hardware-automated control. In the firmware-based approach, the channel control logic and DMA are configured by the flash firmware, while the regular page-size I/O goes through the conventional flash-chip↔DRAM path. In the hardware-automated approach, when a sampling command completes, the parser is activated to read sampling results from the channel bus, classify them into new sampling commands and feature vectors, and forward them to the crossbar and the SSD DRAM. The modified DMA transfers feature vectors without configuring the DRAM address and transfer size every time. To buffer sampling commands from other channels, dispatch queues are employed for each channel, with each flash die coupled with a separate queue. The queues are polled by a command issuer in a round-robin manner, which signals the channel control logic to issue commands when detecting the die is idle.

At a higher level, in&out ports for sampling commands of each channel are connected by a crossbar. The crossbar routes the sampling commands to destination channels according to their addressing information.

C. Bus-attached spatial accelerator

At the SSD level, a spatial accelerator is attached to the internal bus, providing a 1D vector array for feature aggregation, a 2D systolic array for GEMM-based feature update, and an SRAM buffer to cache data and provide low latency and high bandwidth access for these compute units. The SRAM buffer is shared by both vector and systolic arrays and can be configured to flexibly support different input/weight/output data partitions. Data from the flash backend are first written into SSD DRAM and then sent to the accelerator.

VI. System support for BeaconGNN

A. Flash firmware support for DirectGraph

We customize the flash firmware to reserve a list of physical blocks for the host to perform direct manipulation, which bypasses the regular FTL. This manipulation interface is exposed to the host as customized NVMe commands via the *ioctl* system call.

Before the GNN task, the host fetches block addresses from this list. It then processes the original GNN dataset and

Algorithm 1: DirectGraph construction

```
Input:
ppa_list, a list of empty PPA
$\mathcal{G}(\mathcal{V}, \mathcal{E})$, graph structure
$\mathbf{X} \in \mathbb{R}^{d \times |\mathcal{V}|}$, node feature table
/* Initialize metadata for nodes and allocated pages.
   */
node_infos ← map()
p_pages, s_pages ← 2 × ordered_map()
/* Allocate space node by node.                                 */
for v in $\mathcal{V}$ do
    info ← node_metadata()
    info.neighbor_n ← len($\mathcal{N}(v)$)
    calculate the number and sizes of the primary and
      secondary sections of v and record them in info.
    for each section of v do
        /* *_pages refers to either p_pages or s_pages
           according to the section type.                        */
        find a page with sufficient available space from
          *_pages. If it fails, get a page from ppa_list and
          add it to the map.
        record the address of this section in info, and
          (v, sectionindex) in *_pages[v].
    node_infos[v] ← info
allocate a page-size buffer buf in memory
for each page in p_pages ∪ s_pages do
    /* Write sections to buf.                                     */
    for each v, section in page do
        write the section header to buf according to
          node_info[v]
        if section is primary then
            write all v's secondary page addresses, and $\mathbf{X}_v$
              (v's feature vector) to buf
        for each neighbor in section do
            write neighbor's primary section address to
              buf
    flush buf to page.PPA in the SSD
```

flushes the converted DirectGraph directly into these designated blocks. The allocation of physical flash to DirectGraph is done at the block granularity to minimize metadata (block-level bitmap, length=N_{block}) management and modifications to the regular FTL. Subsequently, these blocks are marked in firmware and exempted from the regular allocation and garbage collection processes within the FTL.

B. DirectGraph construction

The conversion to DirectGraph involves two steps: mapping-based metadata collection and subsequent serialization to the SSD. In Step 1, we determine the required space for each section and establish mappings between these sections and the physical pages. This calculation is based solely on the lengths of the neighbor list and the feature vector. In Step 2, we construct each DirectGraph page in the host buffer. For each page section, we fill its entries using information from its corresponding neighbor list fragment and metadata prepared in Step 1. Additionally, primary sections are filled with features. Once all sections are written to the buffer, the page is then flushed to the SSD. Further details are in Algorithm 1.

C. ONFI commands for sampling

ONFI is the most common standard interface for communicating with flash chips. We customize two ONFI commands to configure global GNN parameters and perform the sampling operation, respectively. Data associated with the new commands are transferred over the existing data bus, with communication handled by the extended control logic described in Sections V-A and V-B. Figure 13 shows the formats of the global configurations, the sampling parameters, and the sampling results.

D. Flash-firmware based GNN engine

During the GNN workflow, the host's involvement occurs at two points. First, before the task begins, the host sends converted graph data and essential task information (e.g., GNN model parameters, sampling configuration, and mini-batch size) to the SSD. Second, at the start of a mini-batch, the host informs the SSD of target nodes and their primary-section addresses via customized NVMe commands, similar to what Section VI-A describes. Throughout the rest of the mini-batch, the flash firmware acts as the GNN engine to schedule the workflow. In each mini-batch, the feature vectors and information to reconstruct subgraphs (i.e., batch id, last node id, and current node id in the sampling result) for the last mini-batch are already stored inside DRAM. Therefore, the firmware pipelines the data preparation of the current mini-batch with the computation of the last mini-batch, making the spatial accelerator and flash backend work simultaneously. This overlapped execution scheme achieves higher resource utilization and task throughput.

E. Security and privacy support

In BeaconGNN, although DirectGraph bypasses the host file system and SSD FTL, the block interface of host OS and the standard functionality of FTL remain intact. Therefore, host-side applications can continue their regular storage operations on the SSD and avail themselves of the security and privacy features inherent to the host file system.

Moreover, BeaconGNN enforces data isolation between regular storage I/O accesses and DirectGraph manipulations. On the one hand, DirectGraph blocks are designated as unusable within the FTL, rendering them invisible to the host storage stack and preventing any inadvertent access or modification from regular storage I/O requests. On the other hand, to thwart potential malicious intent, wherein customized BeaconGNN commands might be leveraged to circumvent the storage stack and tamper with normal storage data, the firmware imposes stringent verification procedures. Initially, during the flushing of DirectGraph to the flash, the firmware rigorously checks that writing destination addresses, and section addresses embedded in page contents, are all restricted to blocks allocated for this DirectGraph. Subsequently, the firmware does the same check for primary section addresses of received target nodes in every mini-batch. Finally, section headers are checked by on-die samplers at runtime. In case the section is not found or the type is incorrect, the sampler stops immediately and returns the control to SSD firmware.

F. Reliability and error resilience

Flash-based SSDs are susceptible to two common reliability challenges. Primarily, data stored in flash pages are prone to errors over time due to various factors, including charge leakage from memory cells [31]. BeaconGNN incorporates two optimizations to prevent corruption within DirectGraph. First, during idle time, BeaconGNN firmware periodically performs data scrubbing [31] for DirectGraph blocks. It first reads a flash block and checks every page with the Error Correction Code (ECC) module of the SSD controller. As pages in the same block have similar retention characteristics, once an error is found, the entire physical block is erased and re-programmed with corrected content. Second, BeaconGNN uses SLC Z-NAND flash, which has an extremely low raw bit error rate (RBER) (less than 10^{-7}) [19]. Consequently, corruption instances are drastically rare, and any dangerous runtime error will be caught by on-die checking (cf. Section VI-E), imposing negligible impact on the overall reliability and performance.

Additionally, flash blocks have a limited number of program/erase (P/E) cycles. As P/E operations accumulate, a block becomes increasingly prone to error and ultimately wears out. To extend SSD lifespan, FTL employs a mechanism called *wear leveling* to evenly distribute P/E operations across all flash blocks. However, in the case of BeaconGNN, where flash blocks accommodating DirectGraph are pinned and isolated from conventional FTL operations, other regular blocks end up bearing the brunt of most P/E cycles. To mitigate this, when the discrepancy in P/E counts between DirectGraph blocks and regular blocks reaches a certain threshold, the firmware initiates a reclamation process. Specifically, it migrates DirectGraph to clean regular blocks while updating the embedded physical addresses to these new locations. Then old DirectGraph blocks rejoin the regular FTL management.

G. End-to-end processing

BeaconGNN operates in two modes, namely *regular-I/O* and *acceleration* modes. In regular-I/O mode, it handles regular storage requests, DirectGraph construction (cf. Section VI-A), and preparation of GNN task (cf. Section VI-D). In acceleration mode, it receives mini-batched jobs from the host and starts execution. During this period, any incoming regular storage requests are deferred to the end of the current mini-batch. Note that, due to the small size of DirectGraph metadata and GNN task data, the page table remains in SSD-DRAM without swapping to flash, allowing BeaconGNN to serve regular storage requests quickly even in acceleration mode.

VII. Evaluation

A. Experiment setup

Evaluation platforms. We construct six GNN acceleration systems and evaluate their performance behaviors.

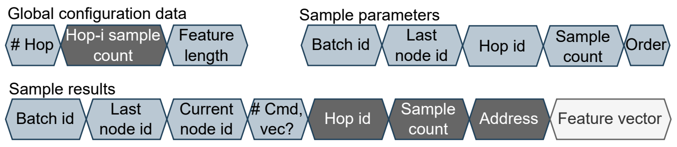


Fig. 13: Data transfer for customized ONFI commands. Data in deep grey are repeated; data in light grey is optional.

• *CPU-centric (CC)*: The host controls the execution of the entire GNN task. It samples neighbors on the host CPU and performs GNN computation on a discrete DNN accelerator. Neighbor lists and feature vectors are retrieved via the block device interface and transferred through the PCIe bus.
• *BeaconGNN-1.0 (BG-1)*: It combines state-of-the-art ISC designs to offload the entire GNN task (cf. Section III). It samples neighbors with SSD processor cores and utilizes an accelerator on SSD-bus for GNN computation.
• *BG-DG*: This design extends `BG-1` with DirectGraph to eliminate multi-level address translations. It offloads neighbor sampling as a single NVMe command and relaxes the hop-by-hop ordering constraint.
• *BG-SP*: This design extends `BG-1` with die-level samplers. Only retrieved features and sampled neighbors are transferred over flash channels to improve the throughput of flash backend.
• *BG-DGSP*: It combines features of `BG-DG` and `BG-SP`.
• *BeaconGNN-2.0 (BG-2)*: This design extends `BG-DGSP` with command routing at the channel level and thus enables hardware-automated backend I/O processing.

Performance modeling. To flexibly model the various systems above, we implement an event-driven cycle-accurate simulator in Python. Our simulator is inspired by existing works such as SimpleSSD [18] and MQSim [67]. It accurately models the latency of traditional SSD components (i.e., embedded processor cores, SSD DRAM, and multi-level flash backend) and the customized multi-level processing engines introduced by our design. Specifically, the SSD-level and PCIe discrete DNN accelerators are modeled using the systolic array simulator ScaleSim-2.0 [61]. The SSD-level accelerator is configured to meet SSD resource budgets [51], while the discrete accelerator is configured as a server-scale TPU [26], which has comparable performance and higher energy efficiency than GPUs for GEMM workloads, and represents the state-of-the-art solution in the traditional Von Neumann architecture. The detailed configurations are listed in Table II.

Area and Power estimation. We model the power of traditional SSD components with McPAT [46] and DRAMPower [4], similar to what SimpleSSD does. To estimate the area and power of die-level and channel-level processing engines, we implement them in Verilog HDL and synthesize them by using the Synopsys Design Compiler [66] at openPDK 40nm technology node [1]. The router logic for all channels brings in total 1.26% additional area to the SSD controller [2] while the sampler (including configuration registers and the TRNG) takes less than 0.1% area of the ULL flash die [9]. For DNN accelerators, we estimate the energy consumption of SRAM with CACTI-7.0 [3] in the 32nm technology node while scaling the energy of arithmetic units also to 32nm [63].

Simulation Configuration	
Controller & Host I/F	4 ARM Cortex-A9 Cores NVMe, PCIe 4.0 ×4
DRAM	DDR4-3200, 25.6 GB/s, 1GB
Flash Memory	16 Channels, 4 Packages, 2 Dies, 2 Planes 1024 Blocks, 1024 Pages, Page size = 4KB NV-DDR3 800MT/s, 8bit Channel Bus
SSD-level Accelerator	64×64 systolic array, 64-width vector unit 6MB shared scratchpad, precision FP16 area 27.2 mm^2, frequency 800MHz
Discrete Accelerator	128×128 systolic array, 128-width vector unit 32MB shared scratchpad, precision FP16 frequency 1GHz
Timing Parameters	page read latency=3 μs host-side software stack latency=10 μs
Energy Parameters	Operation Voltage = 3.3 V $I_{READ}, I_{PROG}, I_{ERASE}$ = 25 mA $I_{BUSIDLE} = 5mA, I_{STDBY}$ = 10 μA PCIe PHY=7.5 pJ/bit [47] host memory read/write=40 pJ/bit sampler=5.23 mW

TABLE II: System simulation configurations.

Dataset	Graph structure			Features
	Nodes	Edges	Avg degree	
Reddit	37.3M	53.9B	1445	602
Amazon	265.9M	79.8B	300	200
Movielens	22.2M	59.2B	2666	30
OGBN	179.1M	5.0B	28	32
PPI	9.1M	8.8B	965	256

TABLE III: GNN dataset information.

Workloads. To evaluate our approach, we adopted the same large-scale GNN datasets as [40], taken from Pytorch Geometric [15]. We also follow the same methodology as [40] to synthesize benchmarks by scaling up real datasets. The details of these benchmarks are listed in Table III.

To prepare the datasets for testing, we convert each of them into the DirectGraph format and store them entirely in the SSD. Nodes in the graph are represented as INT-32 scalar, while features are represented in the form of FP-16 vectors, with the dimension specified by the respective dataset.

We adopt a GNN model that operates on 3-hop subgraphs with 3 neighbors sampled for each node (i.e., a total of 40 nodes in the generated subgraph for each target node). We use vector_sum as the aggregation function and a perception layer for embedding updates. All intermediate embeddings are represented by 128-dimensional FP-16 vectors.

B. Performance improvement

Throughput. Figure 14 compares the throughput of different platforms on five workloads. The results are normalized to the baseline `CC`. We first evaluate two prior designs (i.e., SmartSage and GList) individually, which offload data sampling and GNN computation respectively. Compared to `CC`, they achieve 2.11×, 1.42× throughput on average. `BG-1`, which combines their solutions without further optimizations, achieves 2.35× throughput on average. `BG-DG` has a marginal improvement over `BG-1`, because the high latency of transferring whole pages suppresses the flash chip utilization at a low level.

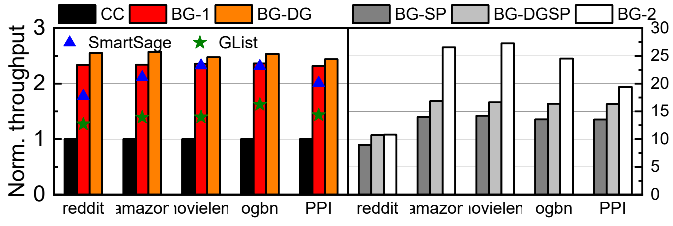


Fig. 14: Throughput on five large-scale GNN datasets.

BG-SP samples nodes and features near flash dies and thus breaks this bottleneck, thereby achieving a giant speedup of 5.47× over BG-1. Furthermore, in BG-SPDG, the out-of-order sampling takes effect and brings an additional 20% improvement. Finally, BG-2 removes the flash firmware from the data preparation path, outperforming BG-DGSP by 41% and obtaining an overall 21.70× speedup.

Flash resource utilization. To understand the root causes for the performance difference among BG-SP, BG-DGSP, and BG-2, we first delve into their flash resource utilization. Figure 15(a-e) depict the number of active flash channels and dies over time. BG-SP contains three remarkable low-utilization valleys. They correspond to the sampling barriers, where one hop of sampling is about to finish, and thus many idle flash resources are wasted. In contrast, BG-DGSP has exhibited more consistent high utilization ever since the beginning. This is because resources that should have been idle in BG-SP are utilized by samplings of later hops prior to the end of the current hop. However, both BG-SP and BG-DGSP have low utilization compared to the total available resources (16 channels, 128 dies). Compared to BG-SP, BG-2 significantly increases the utilization for both flash channels and dies (by 76%) and effectively reduces the total sampling latency (by 78%) for most datasets. This massive leap is achieved by removing flash firmware from the sampling path.

However, die utilization when running reddit and PPI is low even with BG-2. This is because these workloads have high-dimensional features whose accumulated channel-transfer time outweighs the die-read time. Similarly, channel utilization in movielens and OGBN is low because their features are short and transfer quickly, making reading flash dies the bottleneck. amazon exhibits the highest utilization of both flash channels and dies, and its average degree and feature length are representative in common large-scale GNNs. Therefore, we use amazon in later experiments.

Overall latency breakdown. Figure 15f shows the latency breakdown with amazon workload. For baseline CC, the PCIe transfer among the host, the SSD, and the accelerator takes a large part of the entire latency because the PCIe link bandwidth (even high-end Gen4×4) is lower compared to the internal bandwidth in SSD. BG-1 eliminates most PCIe transfer and spends the most time on the flash; BG-DG supports the DirectGraph feature, which nevertheless helps little with this high latency. From BG-SP to BG-2, flash I/O latency gradually decreases. We also observe that the host-side delay is just a minor part of total latency, and most latency reductions come from optimizing the flash I/O.

Hop timeline. Figure 16 shows the timeline of different hops in the data preparation stage. A GNN with k hops of neighbors performs $k+1$ steps: k samplings and the kth-hop feature retrieval. For BG-1 and BG-SP, these hops are performed in strict order with distinct gaps between, and each hop is longer than its prior one. BG-DG, BG-DGSP, and BG-2 overlap these hops and thus achieve lower latency. Among them, BG-2 has the shortest overall time by creating the largest overlap.

command latency breakdown. Figure 17 shows the latency breakdown of a flash command. The lifetime start/end point is the time when its address information/result is available at the frontend controller. On all the examined platforms, the command consumes only a small portion of the lifetime on its own flash processing. However, a large portion of its lifetime is wasted by waiting (wait_before_flash and wait_after_flash) due to the high contention of flash I/O resources. BG-SP drastically reduces the waiting time of both types by cutting down most flash transfers. BG-DG and BG-DGSP have 41%, 42% longer wait_before_flash than BG-1 and BG-DG, respectively. This is because DirectGraph allows more commands to be ready to issue at each moment, and as the number of ready commands increases, the waiting latency turns out to accumulate. Finally, BG-2 implements specialized hardware rather than embedded processor cores to process sampling commands, resulting in a 68% wait time reduction compared to BG-DGSP.

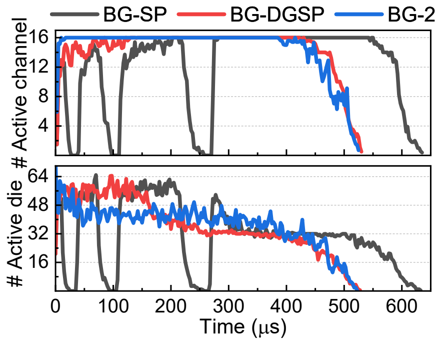
C. *Sensitivity test*

In this subsection, we examine the sensitivity of our design. For each test in Figure 18, we keep the same configuration on all BG-X platforms and vary one system/architecture configuration to observe the effect. The results of each test are normalized to the lowest point.

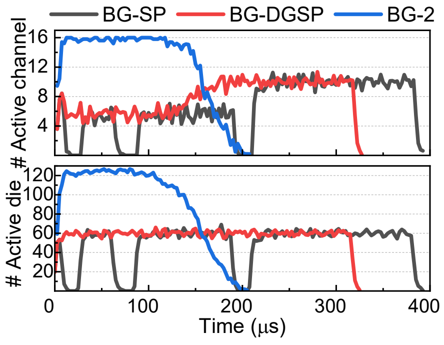
Batch size. We sweep the mini-batch size in the (32, 64, 128, 256) range. BG-1 and BG-DG keep low regardless of the batch size; BG-SP gradually gets close to BG-DGSP with the increasing batch size, indicating that a large batch alleviates the underutilized sampling valley; BG-DGSP itself converges to a limit imposed by the processing capability of flash firmware; BG-2 scales best and shows no obvious convergence in our test range.

Channel bandwidth. We configure the channel bandwidth to other common values: 333 MB/s for low-end SSDs, 1600 MB/s for high-end SSDs, and 2400 MB/s for state-of-the-art models. BG-1 and BG-DG improves significantly with the increasing bandwidth because page-granular channel transfer is their biggest bottleneck; BG-SP and BG-DGSP are constraint by the flash firmware processing; BG-2 improves little with bandwidths higher than 800 MB/s, because the total effective throughput of flash dies already saturates.

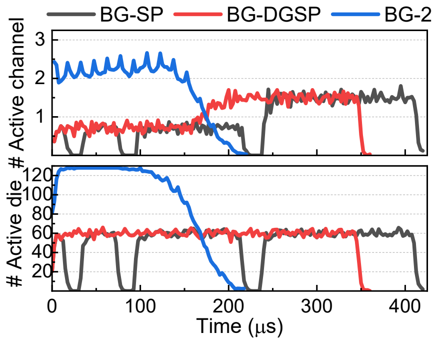
Controller core number. We vary the number of SSD embedded-processor cores from 1 to 8. BG-SP and BG-DGSP are initially close to the baseline, but gradually widen the gap with more cores added; BG-2 is not affected by the number



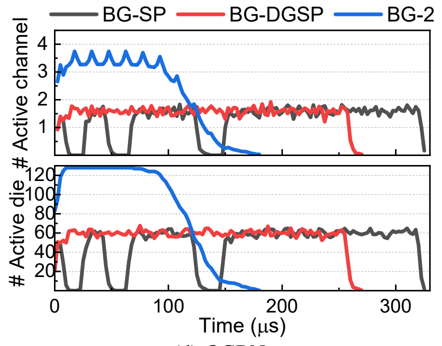
(a) Reddit.



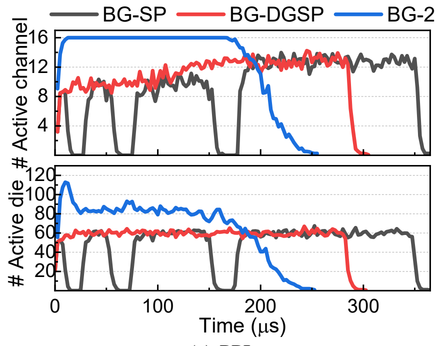
(b) Amazon.



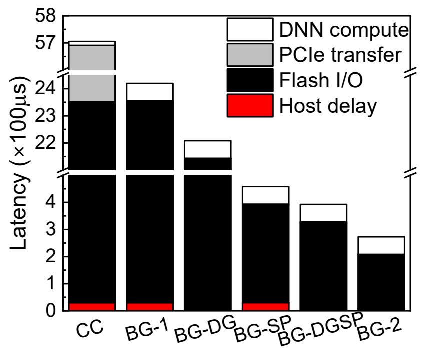
(c) Movielens.



(d) OGBN.



(e) PPI.



(f) Amazon overall latency breakdown.

Fig. 15: (a-e) shows the number of active flash channels/chips during the stage of GNN data preparation for five datasets, respectively. (f) shows the breakdown of the overall latency to complete a batch of processing for `amazon` workload.

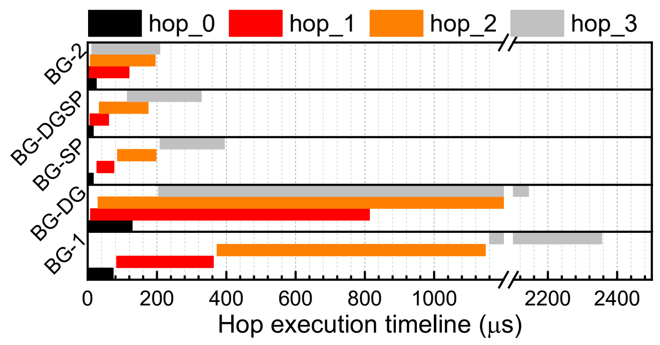


Fig. 16: Hop execution timeline in GNN data preparation.

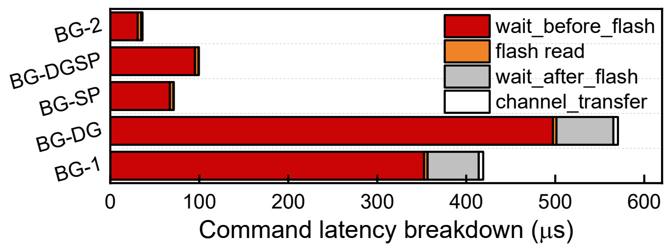


Fig. 17: Sample command latency breakdown.

of cores, but the gap between it and BG-DGSP narrows with the increasing number of cores.

Flash channel number. We keep the die number per channel constant and vary the channel number. BG-1 and BG-DG constantly improve as the number of channels increases, as more channels provide higher bandwidth for both flash read and transfer; BG-SP and BG-DGSP show improvement only with low channel numbers, because firmware processing quickly becomes the primary constraint after 8 channels; BG-2's linear scaling stops at 16 channels because at this point the total channel bandwidth approaches the read/write bandwidth of SSD DRAM; as the channel bandwidth continues growing, the DRAM transfer becomes the bottleneck.

Flash die number. We keep the number of channels constant and vary the number of dies per channel. BG-1 and BG-DG keep low, showing that page-granular transfer is inefficient even for two dies. BG-SP and BG-DGSP increase first and finally converge to the firmware-processing limit, while BG-2 scales linearly until 16 dies/channel when the channel cannot provide enough bandwidth to transfer for all dies.

Flash page size. We vary the size of flash pages from 2 KB to 16 KB. BG-1 and BG-DG perform better with small pages due to less read amplification; BG-SP and BG-DGSP slightly improve with larger pages because more neighbors are stored in one page, and fewer pages need to be read; BG-2 shows no significant variance for different page sizes.

D. *Energy improvement*

Figure 19 shows the energy breakdown/efficiency for `amazon` on all simulated platforms. CC spends 57% of total energy to transfer data outside storage. BG-1 and BG-DG, although limiting data transfer within the storage, still transfer entire pages to the SSD internal DRAM, which consumes 75% of total energy. BG-SP, BG-DGSP, and BG-2 eliminate

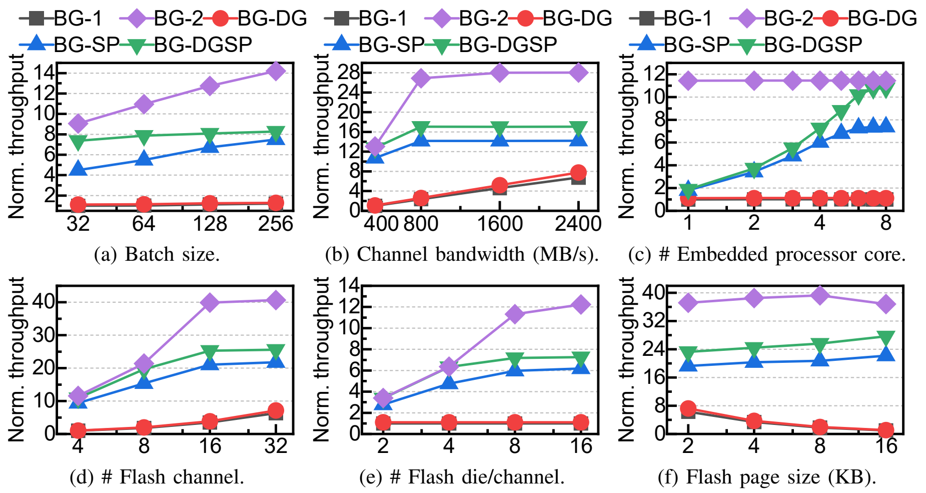


Fig. 18: Sensitivity analysis.

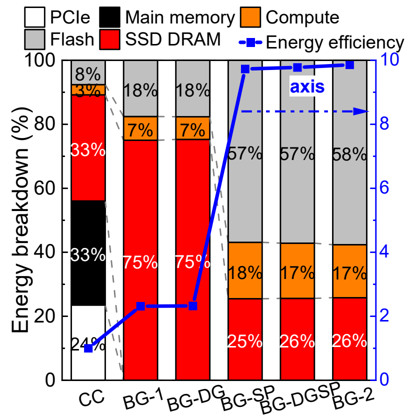


Fig. 19: Energy breakdown and efficiency for `amazon` dataset.

TABLE IV: Raw data volume and inflation with DirectGraph.

Dataset	reddit	amazon	movielens	OGBN	PPI
Raw size (GB)	242.6	397.2	221.8	30.02	37.1
Inflate ratio	2.8%	4.1%	3.5%	32.3%	3.5%

aforementioned energy consumption, spending 57% energy on the flash backend, and 43% on the frontend DRAM buffer and accelerator computation. For energy efficiency, `BG-2` outperforms `CC` and `BG-1` by 9.86× and 4.25×, respectively. Finally, our calculation shows that BG-2 consumes 13.4W power, on average, which is far below the 75W limitation of PCIe power supply [59].

E. Performance on traditional SSD

We also evaluate benchmark performance using a simulated 20μs read-latency SSD. `BG-1`, `BG-DG`, `BG-SP`, `BG-DGSP`, and `BG-2` achieve on average 2.20×, 2.50×, 3.19×, 4.19×, and 4.19× throughput improvement respectively against baseline `CC`. Such results show that DirectGraph and die-level sampling are effective not just for ULL flash, and these two techniques together provide a significant advantage over the `BG-1` approach. Nevertheless, the negligible difference between `BG-DGSP` and `BG-2` shows that for such high read latency, the firmware is sufficient for I/O processing and channel-level routing is unnecessary.

F. DirectGraph storage efficiency

We calculate the storage inflation ratio for converting raw datasets into DirectGraphs by using SSD configurations in Table II. As illustrated in Table IV, all workloads except `OGBN` incur minimal overhead when using DirectGraph formats. In the case of `OGBN`, characterized by a low average degree of 28, its DirectGraph consists mainly of short sections, remaining unused space in flash pages even after section compaction. However, given that most large-scale graphs follow the *Densification law* [11], which means that the average degree increases with the number of nodes, such space wastage exists only in medium-scale graphs with small absolute volumes.

VIII. Discussion

Limitation of SSD DRAM. In our design we use SSD DRAM to buffer data between the flash backend and the accelerator, but we notice that with high flash throughput, the DRAM bandwidth becomes the bottleneck to GNN throughput. One solution is to increase the memory bandwidth, using higher-frequency DRAM or HBM; another is to enable direct I/O between flash and accelerator SRAM, bypassing the DRAM.
Support for GNN query. Although in experiments we only focus on GNN training, our design can also benefit real-time GNN queries. GNN queries are small-batch inference requests, for which the delay is critical. Our design reduces host-SSD communication to one round and avoids the long congestion time on flash channels, greatly improving the overall latency.
Practicality and future proof. BeaconGNN is practical regarding TDP and system-level TCO. For TDP, BeaconGNN satisfies the PCIe power limitation and exhibits higher performance per watt (cf. Section VII-D). For TCO, BeaconGNN is more economical in terms of both capital and operational expenditures. Specifically, its full-stage-ISC design saves PCIe slots and expenses of deploying GPUs, and its system-level energy reduction leads to lower running costs.

Moreover, as storage or compute requirements of GNN workloads continue increasing, we expect that BeaconGNN would scale out, forming computational storage arrays to cope with such demands. Specifically, within a storage array, multiple BeaconGNN SSDs communicate via direct P2P links and work collaboratively. With such an architecture, both the storage capacity and the computation power would increase linearly with the number of SSDs, and the optimizations in BeaconGNN continue to take effect.

IX. Related work

GNN acceleration. There are copious prior works on GNN acceleration (both inference and training). Some works [17], [65], [74], [78] design specialized compute engines and dataflow to suit irregular GNN data pattern; some [5], [6], [55] exploit the high feature sparsity and biased degree distribution of GNN, to co-design the algorithm and hardware, and reduce data access and compute redundancy; some [25], [75] utilize emerging hardware such as ReRAM for their high parallelism and energy efficiency. Specific to ISC for GNN, SmartSage [40] and GList [44] are not designed to offload the entire GNN taskflow to storage, whose limitations have been well discussed in prior sections; GraphStore [39] stores both the graph structure and the feature table in the SSD, but it computes on a separate FPGA, with performance severely constrained by the bandwidth of the FPGA-SSD P2P link. Lastly, none of them removes host control or optimizes I/O efficiency in the flash backend.

In-storage computing. In-storage computing (ISC) has been proposed for many applications such as query processing [38], [62], data analytics [20], [27], [68], [83], deep learning inference/training [21], [32], [33], [48], [49], [51], [71], graph processing [28], [53], and bioinformatics [52], [73]. While existing research mainly focuses on processing data using embedded cores or additional logic near the controller, there remains a gap in exploring the potential of processing near flash channels [51], [83] and dies [10], [21], [32]. To the best of our knowledge, BeaconGNN is the first architecture to exploit processing at multiple levels of SSD hierarchy, with channel transfer, I/O communication, and computation optimized altogether in a single design.

X. Conclusion

In this paper, we explore the design of offloading the complete taskflow of large-scale GNN to ULL flash. We identify that the strict GNN neighbor sampling order, the page-granular flash channel transfer, and the firmware-based flash request processing can become obstacles to high performance. To address these issues, we propose BeaconGNN, an out-of-order streaming in-storage computing system. It lifts the in-order graph sampling restriction with a novel GNN graph format, improves the flash channel transfer efficiency by sampling neighbors at the flash die level, and accelerates the flash request processing by routing commands directly among channels. BeaconGNN improves the throughput and energy efficiency by up to $11.6\times$ and $4\times$, respectively, compared to the state-of-the-art ISC design.

Acknowledgement

Yuyue Wang is supported by Samsung under the Center for Domain-Specific Computing (CDSC). Dr. Jie Zhang is supported in part by the National Key Research and Development Program of China under Grant No. 2023YFB4502702, the Natural Science Foundation of China under Grant No. 62332021, and the Fundamental Research Funds for the Central Universities, Peking University.

References

[1] "Openpdks - open circuit design," http://opencircuitdesign.com/open_pdks/.

[2] ARM, "Cortex-A9," https://developer.arm.com/Processors/Cortex-A9.

[3] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.

[4] K. Chandrasekar, C. Weis, Y. Li, B. Akesson, N. Wehn, and K. Goossens, "Drampower: Open-source dram power & energy estimation tool," *URL: http://www.drampower.info*, vol. 22, 2012.

[5] C. Chen, K. Li, Y. Li, and X. Zou, "Regnn: A redundancy-eliminated graph neural networks accelerator," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 429–443.

[6] C. Chen, K. Li, X. Zou, and Y. Li, "Dygnn: Algorithm and architecture support of dynamic pruning for graph neural networks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 1201–1206.

[7] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1, pp. 181–192, 2009.

[8] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE, 2011, pp. 266–277.

[9] Choe, Jeongdong, "Comparison of Current 3D NAND Chip Cell Architecture," https://www.flashmemorysummit.com/Proceedings2019/08-07-Wednesday/20190807_FTEC-202-1_Choe.pdf, 2019.

[10] M. Chun, J. Lee, S. Lee, M. Kim, and J. Kim, "Pif: In-flash acceleration for data-intensive applications," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 106–112.

[11] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[12] C. Dirik and B. Jacob, "The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 279–289, 2009.

[13] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb." in *CIDR*, vol. 3, 2017, p. 3.

[14] W. Fan, Y. Ma, Q. Li, Y. He, E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *The world wide web conference*, 2019, pp. 417–426.

[15] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[16] C. Gao, X. Wang, X. He, and Y. Li, "Graph neural networks for recommender system," in *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, 2022, pp. 1623–1625.

[17] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[18] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung, "Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 469–481.

[19] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing flash memory: Anomalies, observations, and applications," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 24–33.

[20] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho *et al.*, "Biscuit: A framework for near-data processing of big data workloads," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 153–165, 2016.

[21] S. Gupta, J. Morris, M. Imani, R. Ramkumar, J. Yu, A. Tiwari, B. Aksanli, and T. Š. Rosing, "Thrifty: Training with hyperdimensional computing across flash hierarchy," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[22] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[23] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the international conference on Supercomputing*, 2011, pp. 96–107.

[24] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped ssds with flashmap," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, 2015, pp. 580–591.

[25] Y. Huang, L. Zheng, P. Yao, Q. Wang, X. Liao, H. Jin, and J. Xue, "Accelerating graph convolutional networks using crossbar-based processing-in-memory architectures," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 1029–1042.

[26] N. P. Jouppi, D. H. Yoon, G. Kurian, S. Li, N. Patil, J. Laudon, C. Young, and D. Patterson, "A domain-specific supercomputer for training deep neural networks," *Communications of the ACM*, vol. 63, no. 7, pp. 67–78, 2020.

[27] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind, "Bluedbm: An appliance for big data analytics," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 1–13, 2015.

[28] S.-W. Jun, A. Wright, S. Zhang, S. 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, pp. 411–424.

[29] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2014, pp. 524–535.

[30] M. Jung, E. H. Wilson III, and M. Kandemir, "Physically addressed queueing (paq) improving parallelism in solid state disks," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 3, pp. 404–415, 2012.

[31] B. S. Kim, J. Choi, and S. L. Min, "Design tradeoffs for {SSD} reliability," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 281–294.

[32] J. Kim, M. Kang, Y. Han, Y.-G. Kim, and L.-S. Kim, "Optimstore: In-storage optimization of large scale dnns with on-die processing," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 611–623.

[33] S. Kim, Y. Jin, G. Sohn, J. Bae, T. J. Ham, and J. W. Lee, "Behemoth: a flash-centric training accelerator for extreme-scale {DNNs}," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 371–385.

[34] T. Y. Kim, D. H. Kang, D. Lee, and Y. I. Eom, "Improving performance by bridging the semantic gap between multi-queue ssd and i/o virtualization framework," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–11.

[35] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[36] S. Koh, J. Jang, C. Lee, M. Kwon, J. Zhang, and M. Jung, "Faster than flash: An in-depth study of system challenges for emerging ultra-low latency ssds," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 216–227.

[37] S. Koh, C. Lee, M. Kwon, and M. Jung, "Exploring system challenges of ultra-low latency solid state drives," in *10th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[38] G. Koo, K. K. Matam, T. I, H. K. G. Narra, J. Li, H.-W. Tseng, S. Swanson, and M. Annavaram, "Summarizer: trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 219–231.

[39] M. Kwon, D. Gouk, S. Lee, and M. Jung, "{Hardware/Software}{Co-Programmable} framework for computational {SSDs} to accelerate deep learning service on {Large-Scale} graphs," in *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022, pp. 147–164.

[40] Y. Lee, J. Chung, and M. Rhu, "Smartsage: training large-scale graph neural networks using in-storage processing architectures," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 932–945.

[41] Y. Lee, Y. Kwon, and M. Rhu, "Understanding the implication of non-volatile memory for large-scale graph neural network training," *IEEE Computer Architecture Letters*, vol. 20, no. 2, pp. 118–121, 2021.

[42] A. Lerer, L. Wu, J. Shen, T. Lacroix, L. Wehrstedt, A. Bose, and A. Peysakhovich, "Pytorch-biggraph: A large scale graph embedding system," *Proceedings of Machine Learning and Systems*, vol. 1, pp. 120–131, 2019.

[43] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 177–187.

[44] C. Li, Y. Wang, C. Liu, S. Liang, H. Li, and X. Li, "Glist: Towards in-storage graph learning." in *USENIX Annual Technical Conference*, 2021, pp. 225–238.

[45] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The {CASE} of {FEMU}: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 83–90.

[46] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, 2009, pp. 469–480.

[47] S. Li, F. Spagna, J. Chen, X. Wang, L. Tong, S. Gowder, W. Jia, R. Nicholson, S. Iyer, R. Song *et al.*, "A power and area efficient 2.5-16 gbps gen4 pcie phy in 10nm finfet cmos," in *2018 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2018, pp. 5–8.

[48] S. Li, F. Tu, L. Liu, J. Lin, Z. Wang, Y. Kang, Y. Ding, and Y. Xie, "Ecssd: Hardware/data layout co-designed in-storage-computing architecture for extreme classification," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, 2023, pp. 1–14.

[49] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li, "Cognitive {SSD}: A deep learning engine for {In-Storage} data retrieval," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 395–410.

[50] C.-Y. Liu, J. Kotra, M. Jung, and M. Kandemir, "{PEN}: Design and evaluation of {Partial-Erase} for 3d {NAND-Based} high density {SSDs}," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 67–82.

[51] V. S. Mailthody, Z. Qureshi, W. Liang, Z. Feng, S. G. De Gonzalo, Y. Li, H. Franke, J. Xiong, J. Huang, and W.-m. Hwu, "Deepstore: In-storage acceleration for intelligent queries," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 224–238.

[52] N. Mansouri Ghiasi, J. Park, H. Mustafa, J. Kim, A. Olgun, A. Gollwitzer, D. Senol Cali, C. Firtina, H. Mao, N. Almadhoun Alserr *et al.*, "Genstore: a high-performance in-storage processing system for genome sequence analysis," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 635–654.

[53] K. K. Matam, G. Koo, H. Zha, H.-W. Tseng, and M. Annavaram, "Graphssd: graph semantics aware ssd," in *Proceedings of the 46th international symposium on computer architecture*, 2019, pp. 116–128.

[54] J. Mohoney, R. Waleffe, H. Xu, T. Rekatsinas, and S. Venkataraman, "Marius: Learning massive graph embeddings on a single machine," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 21*, 2021.

[55] S. Mondal, S. D. Manasi, K. Kunal, and S. S. Sapatnekar, "Gnnie: Gnn inference engine with load-balancing and graph-specific caching," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 565–570.

[56] NVM Express, Inc., "Nvme 2.0 specification," https://nvmexpress.org/specifications/, 2021.

[57] G. Panagopoulos, G. Nikolentzos, and M. Vazirgiannis, "Transfer graph neural networks for pandemic forecasting," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 6, 2021, pp. 4838–4845.

[58] Y. Park, S. Min, and J. W. Lee, "Ginex: Ssd-enabled billion-scale graph neural network training on a single machine via provably optimal in-memory caching," in *Proceedings of the VLDB Endowment*, vol. 15, no. 11, 2022.

[59] PCI-SIG. (2017) Pci express base specification 4.0. [Online]. Available: https://pcisig.com/specifications

[60] Z. Ruan, T. He, and J. Cong, "{INSIDER}: Designing {In-Storage} computing system for emerging {High-Performance} drive," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 379–394.

[61] A. Samajdar, J. M. Joseph, Y. Zhu, P. Whatmough, M. Mattina, and T. Krishna, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 58–68.

[62] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson, "Willow: A {User-Programmable}{SSD}," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 67–80.

[63] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 97–108, 2014.

[64] C. Song, B. Wang, Q. Jiang, Y. Zhang, R. He, and Y. Hou, "Social recommendation with implicit social influence," in *proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*, 2021, pp. 1788–1792.

[65] J. R. Stevens, D. Das, S. Avancha, B. Kaul, and A. Raghunathan, "Gnnerator: A hardware/software framework for accelerating graph neural networks," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 955–960.

[66] Synopsys, "Design compiler," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-graphical.html.

[67] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "{MQSim}: A framework for enabling realistic studies of modern {Multi-Queue}{SSD} devices," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 49–66.

[68] M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, H. Bobarshad, V. Alves, and N. Bagherzadeh, "Catalina: in-storage processing acceleration for scalable big data analytics," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 430–437.

[69] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[70] R. Waleffe, J. Mohoney, T. Rekatsinas, and S. Venkataraman, "Mariusgnn: Resource-efficient out-of-core training of graph neural networks," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 144–161.

[71] M. Wilkening, U. Gupta, S. Hsia, C. Trippel, C.-J. Wu, D. Brooks, and G.-Y. Wei, "Recssd: near data processing for solid state drive based recommendation inference," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 717–729.

[72] S. Wu, F. Sun, W. Zhang, X. Xie, and B. Cui, "Graph neural networks in recommender systems: a survey," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–37, 2022.

[73] W. Xu, J. Kang, and T. Rosing, "A near-storage framework for boosted data preprocessing of mass spectrum clustering," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 313–318.

[74] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[75] T. Yang, D. Li, Y. Han, Y. Zhao, F. Liu, X. Liang, Z. He, and L. Jiang, "Pimgcn: a reram-based pim design for graph convolutional network acceleration," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2021, pp. 583–588.

[76] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 974–983.

[77] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, and J. Leskovec, "Graph convolutional neural networks for web-scale recommender systems," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 974–983.

[78] H. You, T. Geng, Y. Zhang, A. Li, and Y. Lin, "Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 460–474.

[79] J. Zhang, D. Donofrio, J. Shalf, M. T. Kandemir, and M. Jung, "Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 13–24.

[80] J. Zhang and M. Jung, "Zng: Architecting gpu multi-processors with new flash for scalable data analysis," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 1064–1075.

[81] J. Zhang, M. Kwon, H. Kim, H. Kim, and M. Jung, "Flashgpu: Placing new flash next to gpu cores," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–6.

[82] Q. Zhao, J. Chen, M. Chen, S. Jain, A. Beutel, F. Belletti, and E. H. Chi, "Categorical-attributes-based item classification for recommender systems," in *Proceedings of the 12th ACM conference on recommender systems*, 2018, pp. 320–328.

[83] C. Zou and A. A. Chien, "Assasin: Architecture support for stream computing to accelerate computational storage," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2022, pp. 354–368.