FlashGNN
: An In-SSD Accelerator for GNN Training

Fuping Niu, Jianhui Yue, Jiangqiu Shen… , Huazhong University of Science and Technology

HPCA' 24

2024. 08. 14

Presentation by Suhwan Shin

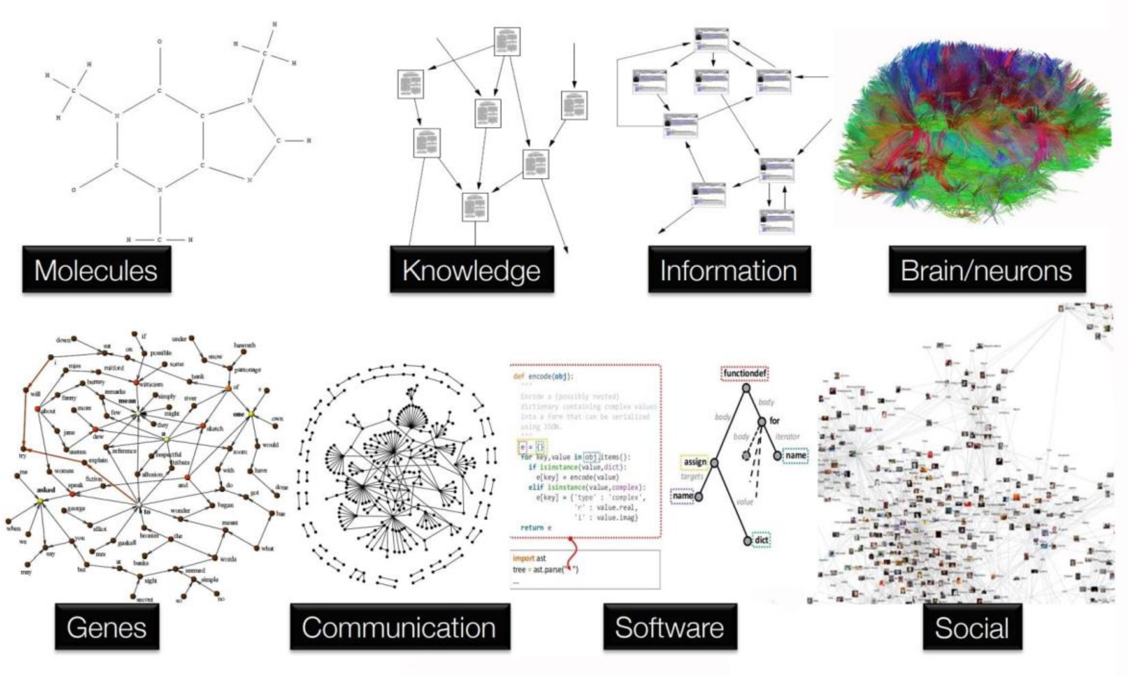
sshshshin@dankook.ac.kr

Contents

Introduction

- Recently, Graph Neural Network(GNN) have emerged as a powerful tool in various applications
- But, existing GNN training systems have reached their performance limits in large-scale datasets

  → The main challenges of GNN training are inefficient access to large amounts of data
- Existing SSD-based systems suffer from network bottlenecks and low hardware utilization



https://blogs.nvidia.com/blog/what-are-graph-neural-networks/

Background

- Graph Neural Network (GNN)
  - Machine learning models designed for processing graph-structured data
  - Node: Entities
  - Edge: Relationships between entities
  - Each layer consists of two sequential operations: Aggregation and Combination
  - h_v^k: Feature of node v at layer-k
  - $\mathrm{N(v)}$: Set of neighbors of node v

$$a_v^k = Aggregate(\{h_v^{k-1}\} \cup \{h_u^{k-1} | u \in N(v)\})$$
$$h_v^k = Combine(a_v^k)$$

$$a_v^k = Mean(\{h_v^{k-1}\} \cup \{h_u^{k-1} | u \in S(v)\})$$
$$h_v^k = ReLU(W^k a_v^k + b^k)$$

- GraphSage (GNN Model, NIPS' 17)
  - Employs a **Neighbor Sampling** to effectively address the neighborhood explosion

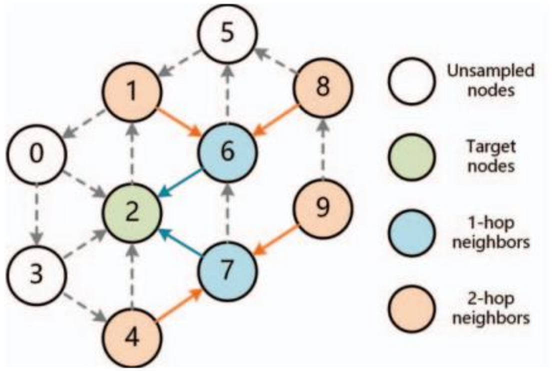


Fig. 1. An example of neighbor sampling

Motivation

- Previously, data transfer is slowed down due to PCIe bandwidth, and CPU & GPU utilization is low

- Minimize data movement and optimize perf. by performing GNN training directly within the SSD

Problem

- GNN training on large graph datasets
- I/O bottleneck
- Inefficiency in data access

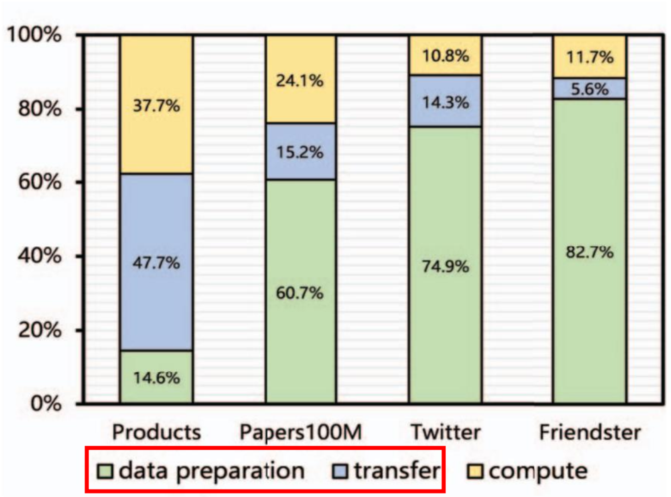
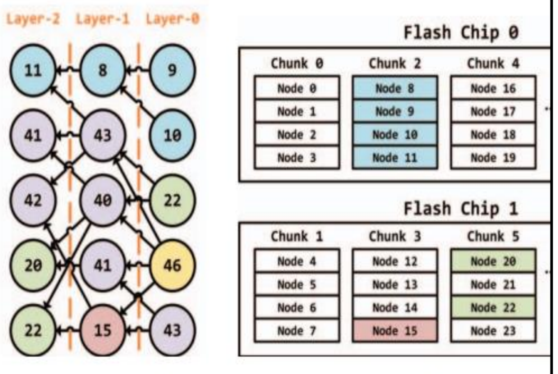


Fig. 3. Ginex runtime breakdown

FlashGNN

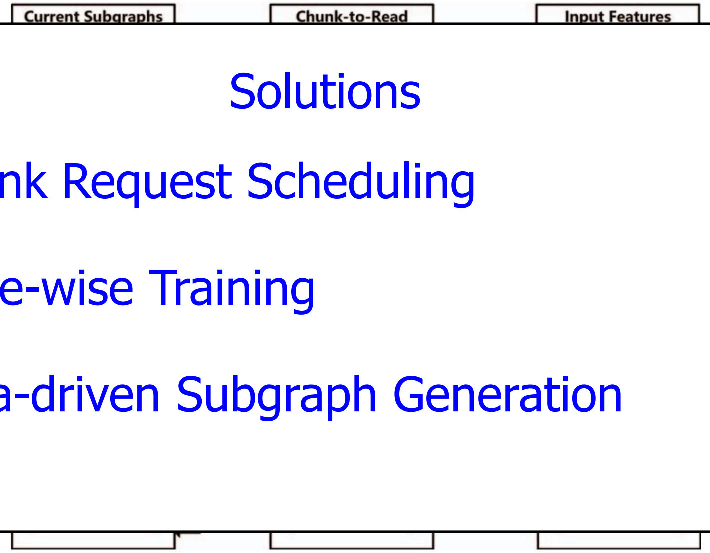
- FlashGNN is an accelerator that performs GNN training inside the SSD
  - Solving PCIe bottlenecks and maximizing I/O parallelism
  - Efficiently reuses data from flash memory chunks in the SSD to optimize performance

➢ The main goal of FlashGNN is to minimize data movement and maximize performance



(a) Subgraph (b) Datalayout

Fig. 6. An example of efficient chunk request sc



Solutions

- Chunk Request Scheduling
- Node-wise Training
- Data-driven Subgraph Generation

Fig. 8. Process flow for node-wise training

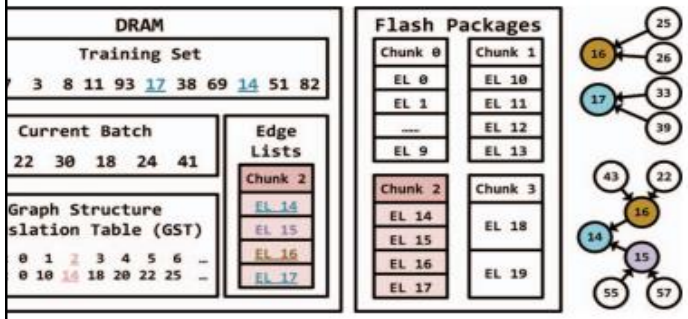


Fig. 9. An example of data-driven subgraph generation

Design - FlashGNN

- FlashGNN Architecture
  - FlashGNN's architecture shows the components of the system
  - Neighbor Sampling
    - Create a subgraph by taking the required edge list
    - Sampling the neighbors of each node
  - Message Passing
    - Chunk Request Scheduling
    - Node-wise training
  - Aggregator
    - Performs the role of aggregating features of node neighbors (Utilizing multiple SIMD cores)
  - Combiner
    - Combine aggregated features using systolic arrays
    - Compute final features

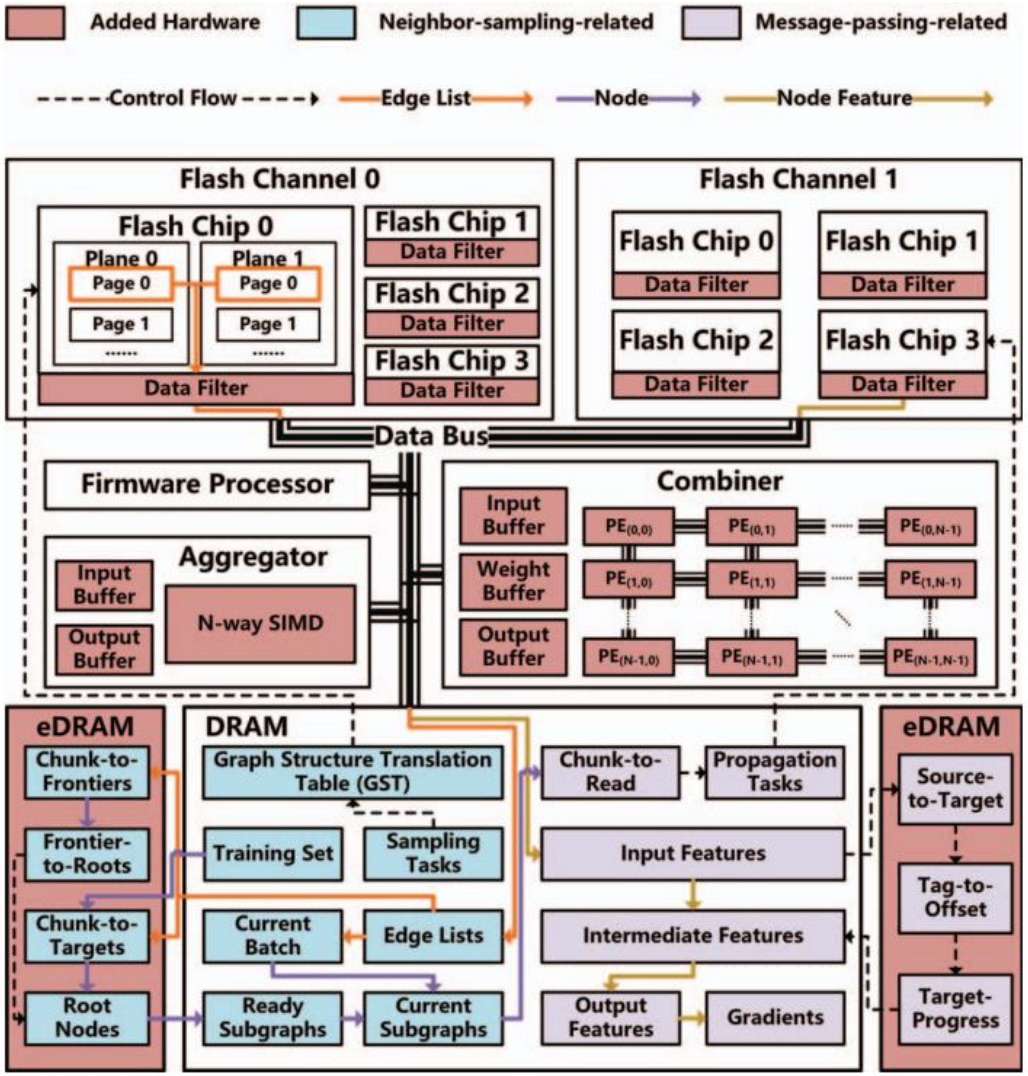


Fig. 4. FlashGNN architecture

Design - FlashGNN

- Chunk Request Scheduling
  - Problem: Sequentially accessing input features by layer
    → Inefficient because the same memory chunk is accessed repeatedly
  - Approach: Maximize the use of input features within a chunk by considering data dependencies between all subgraphs
  - Effect: Reduce the number of flash memory accesses

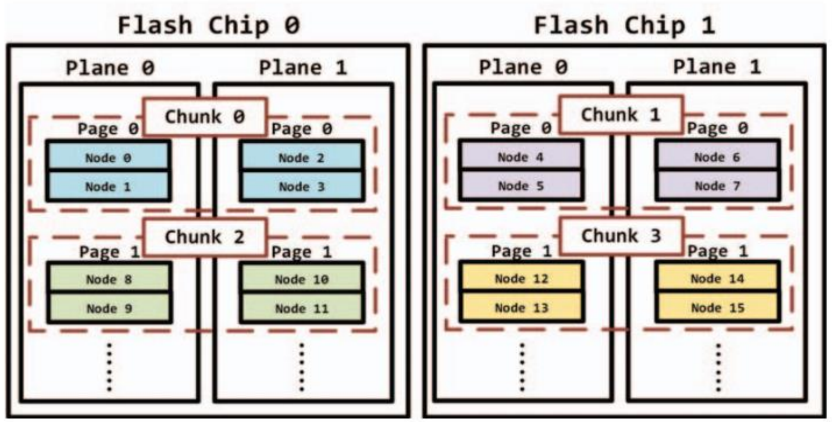
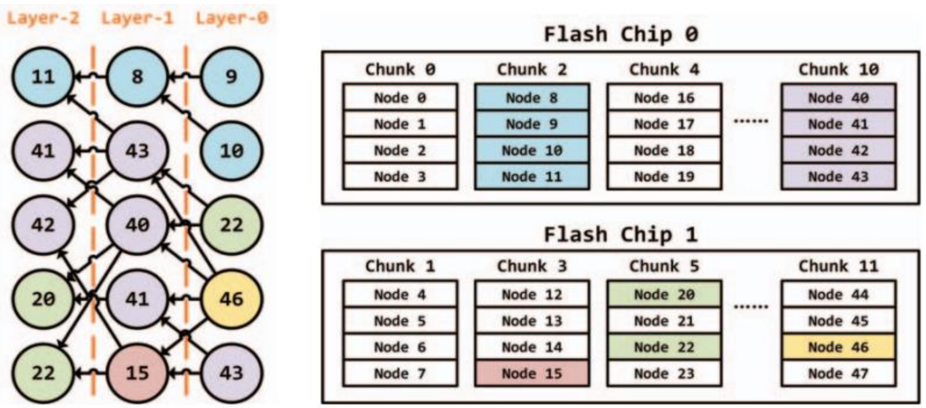
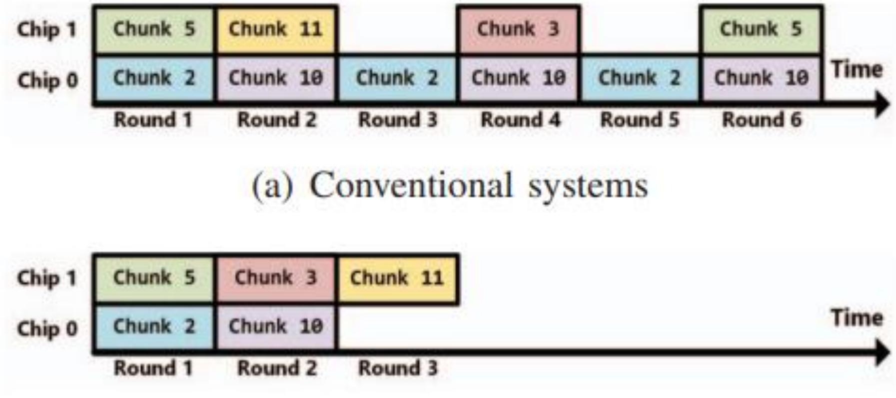


Fig. 5. The organization of flash chunks



(a) Subgraph (b) Datalayout

Fig. 6. An example of efficient chunk request scheduling



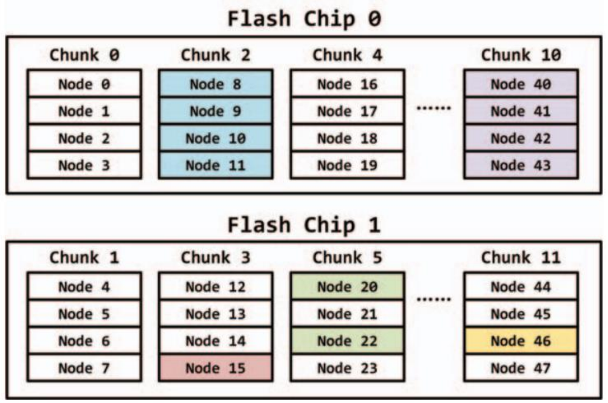
(a) Conventional systems

(b) FlashGNN

Fig. 7. The comparison of chunk scheduling

Design - FlashGNN

- Node-wise Training (NOD)
  - Problem: All nodes in the current layer must be processed before computing nodes in the next layer
    → Increased memory usage and extensive synchronization required
  - Approach: To starts calculations for each node as soon as the dependent data is ready
  - Effect: Reduce memory usage and increase utilization of computing components
  - Black process: Implemented by multi-thread firmware
  - Red process: Implmented by hardware
  - Q: Why is this process necessary?
  - A: NOD improves the utilization of DRAM bandwidth and increases the utilization
    of systolic arrays, improving the training throughput for in-SSD accelerators



(b) Datalayout

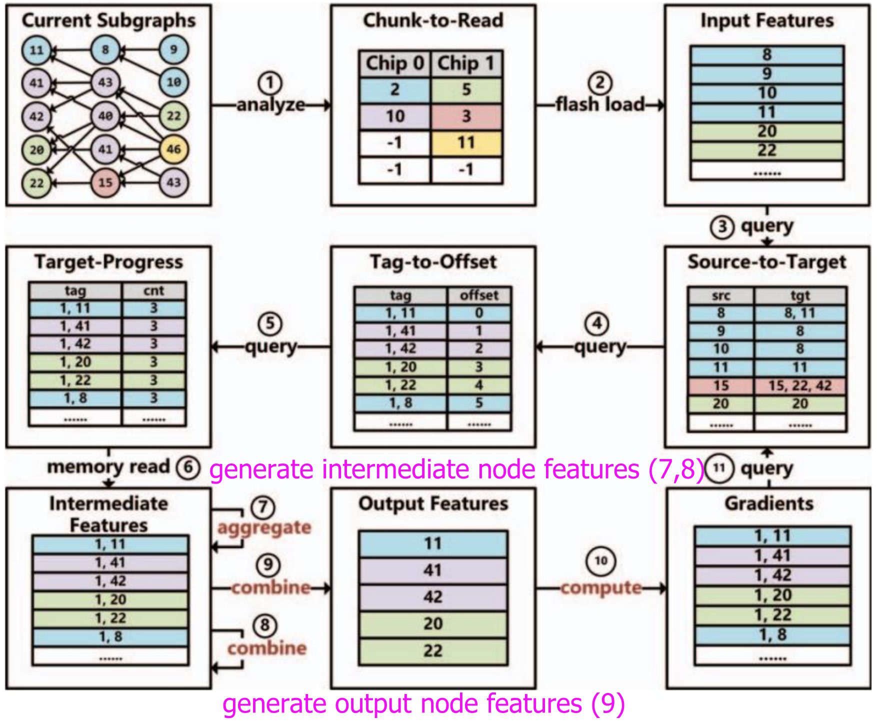


Fig. 8. Process flow for node-wise training

Design - FlashGNN

- Data-driven subgraph generation
  - Problem: Inefficient use of edge lists within chunks
    → Causes frequent and inefficient access
  - Approach: Maximize the use of edge lists within chunks to pre-generate subgraphs needed for future training batches
  - Effect: Hide latency and reduce the need for frequent chunk access by preparing subgraph creation in advance

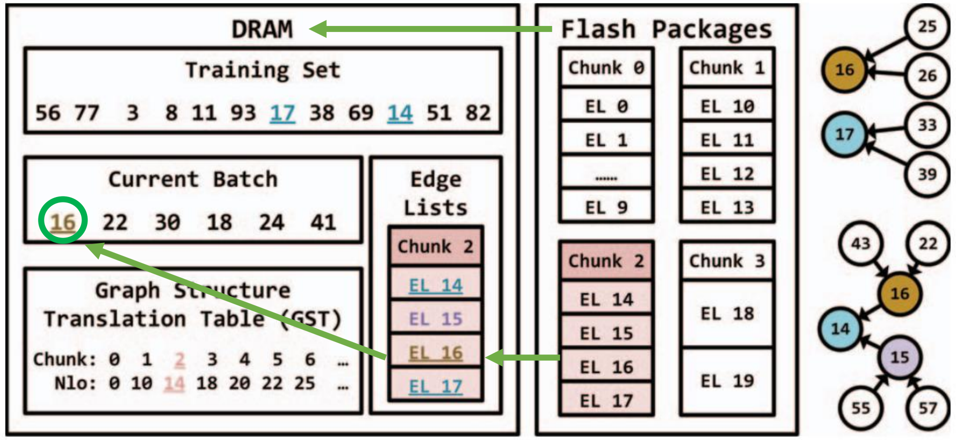


Fig. 9. An example of data-driven subgraph generation

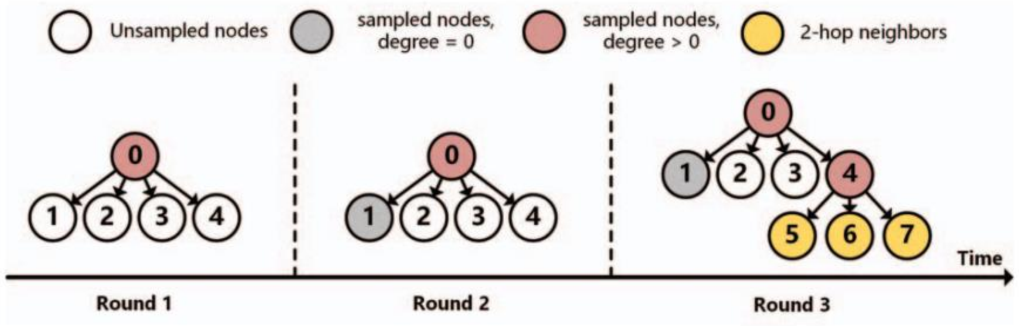


Fig. 10. An example of sampling progress tracking

Progress[N_n]: the number of nodesin SG[N_n] that have been sampled
Goal[N_n]: the number of nodes in SG[N_n]
Progress[N_n] = Goal[N_n]: Complete subgraph generation

Evaluation

- Evaluation Setup

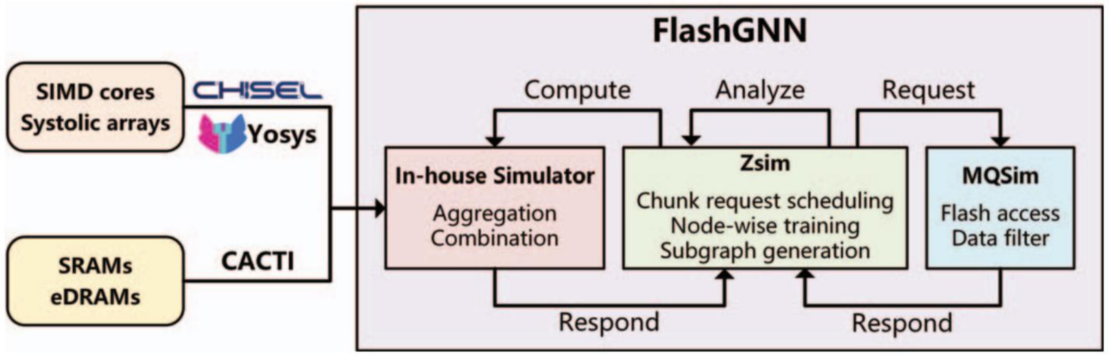


Fig. 11. Evaluation setup

|N|: Node included in the dataset

|E|: Edge included in the dataset

CSC Size: Size stored in the CSC format of the graph (Compressed Sparse Column)

Feature Size: Total size of node features included in each dataset

TABLE I
FLASHGNN CONFIGURATIONS

Flash Organization	32 channels, 4 chips per channel, 4 planes per chip, 2048 blocks per plane, 64 pages per block, 4KB page capacity
Flash Communication	333MT/s transfer rate per channel, 35µs read latency, 350µs program latency
DRAM	2 channels, 4GB per channel, 25.6GB/s per channel
Aggregator	64-way SIMD, 480MHz, 64KB input buffer, 16KB output buffer
Combiner	128 × 128 × 2 subarrays, 310MHz, 64KB weight/input/output buffer
Scratchpad Memory	32MB eDRAM
Firmware Processor	4 quad-core OOO CPU, 2GHz, 250mW/core, 64KB L1 private cache, 4MB L2 shared cache

TABLE II
DATASETS USED FOR EVALUATION

Dataset	$\|N\|$	$\|E\|$	CSC Size	Feature Size
Products	2.5M	61.9M	307MB	2.4GB
Papers100M	111.1M	1.6B	8.0GB	105.9GB
Twitter	41.7M	1.5B	7.4GB	39.7GB
Friendster	65.6M	3.6B	8.0GB	62.6GB

Evaluation

- Performance (Compare with Ginex)
  - Comparison with FlashGNN for Ginex (Using PCIe 3.0 SSDs and 8GB, 16GB, and 32GB DRAMs and for PCIe 4.0 SSD RAID0)
  - FlashGNN achieves an average speedup of 7.88x when using PCIe 3.0 SSDs and 8GB DRAMs
  - FlashGNN maintains a speedup of more than 5x when using larger memory and SSD RAID0 configurations
  - FlashGNN's speedup varies across datasets
    - Because the node graphs of Papers100M and Twitter datasets have a power law, which allows Ginex to reduce SSD accesses through caching strategies

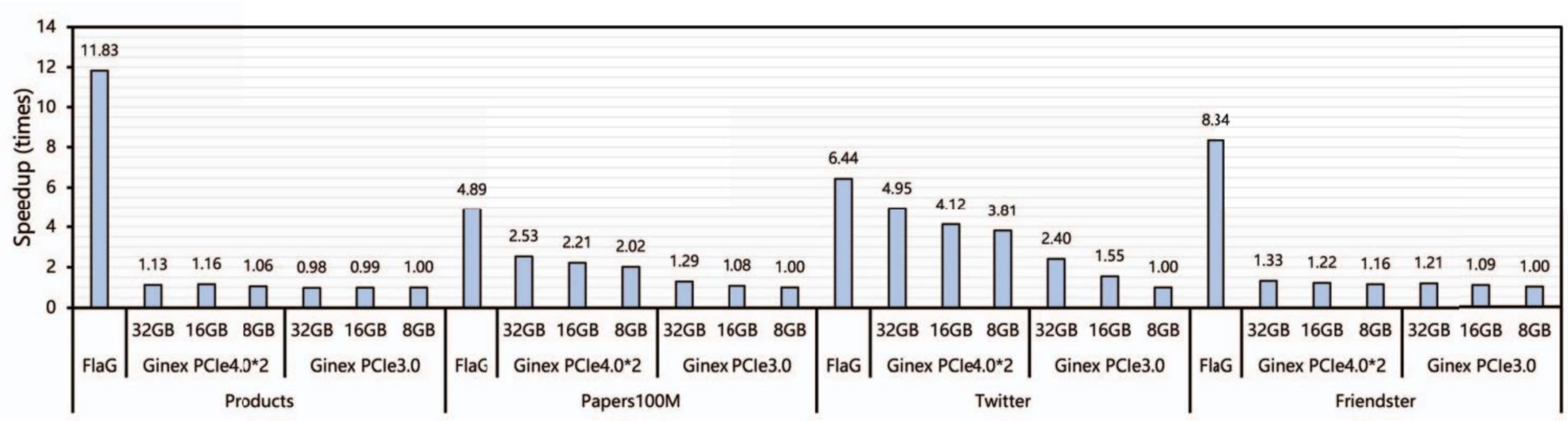


Fig. 12. FlashGNN speedup over Ginex

Evaluation

- Power & Area Overheads
  - Achieves energy savings of 192.66×, 60.35×, 64.25×, and 57.14× (average of 93.60×)
  - Power consumption is mainly for flash and DRAM access
  - SIMD cores and eDRAM consume negligible power

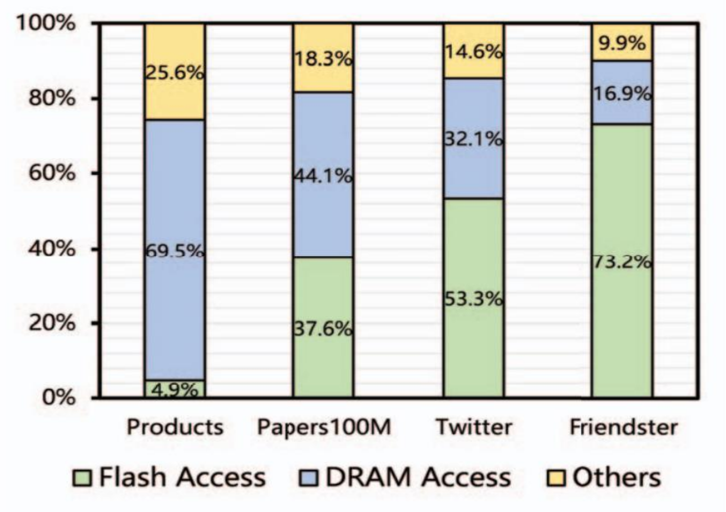


Fig. 13. FlashGNN power consumption breakdown

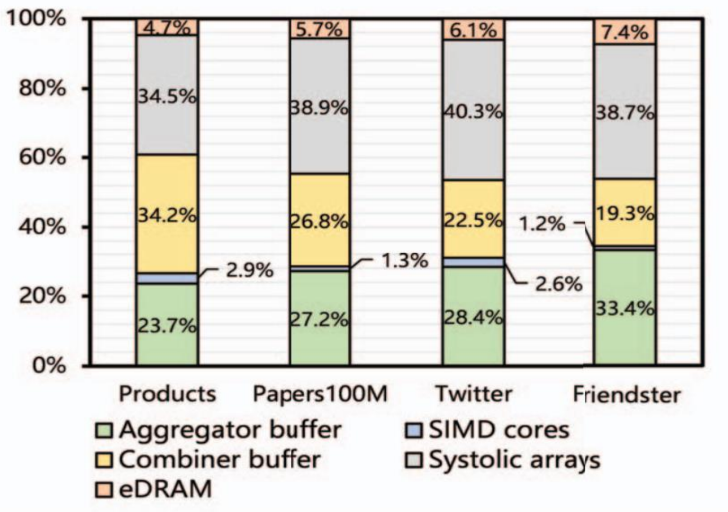
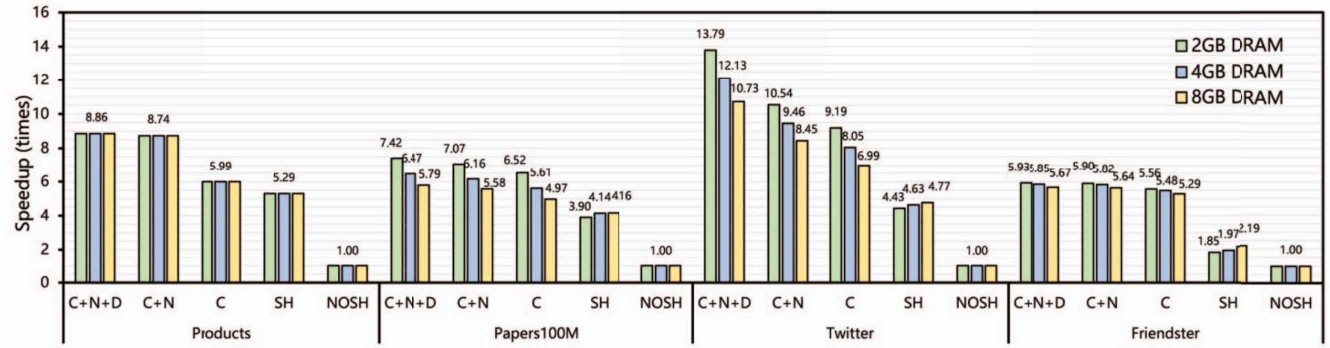


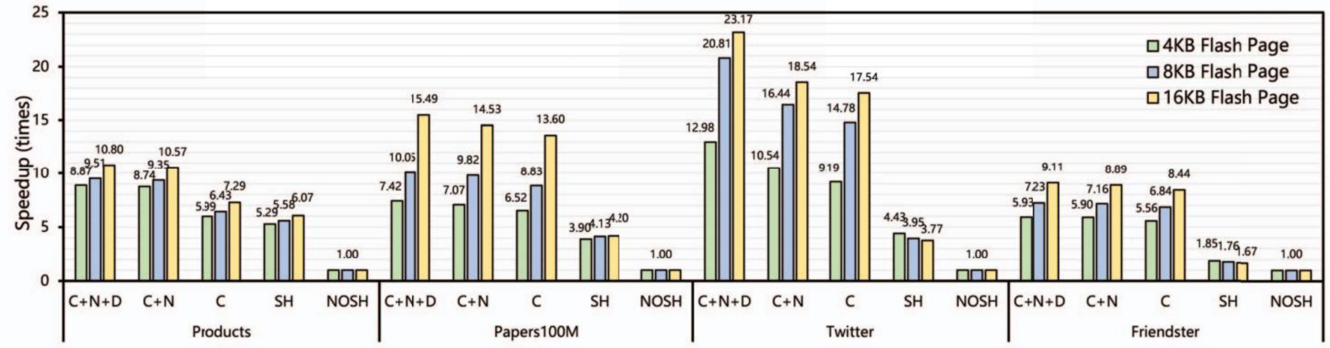
Fig. 14. Hardware power consumption breakdown

Evaluation

- Compare with State-of-the-art (In-Storage Accelerator: SmartSAGE+)
  - C: Chunk request scheduling
  - N: Node-wise training
  - D: Data-driven subgraph generations
  - SH: Data-sharing SmartSAGE
  - NOSH: No data-sharing SmartSAGE (baseline)



(a) Fixing flash page capacity to 4KB, varying DRAM capacities



(b) Fixing DRAM capacity to 2GB, varying flash page capacities

Conclusion

- Existing GNN training systems have not been able to achieve efficient performance
  - The performance limitations of CPU/GPU-based systems
  - The slow data transfer speed of SSDs when processing large-scale graph datasets

- FlashGNN solved the problem by optimizing SSD firmware
  - To overcome PCIe bottlenecks and maximizing the I/O parallelism within SSDs

- FlashGNN achieved up to 23.17 times <u>better performance</u> than the latest SSD-based systems and SmartSAGE+, and also significantly <u>improved energy efficiency</u>

Thank you

Q&A

Background

- Supervised GNN Training
  - Initializing the model with random weights and biases
  - Dividing the training set into batches
  - Processing the features of each node (Aggregate and Combine functions)

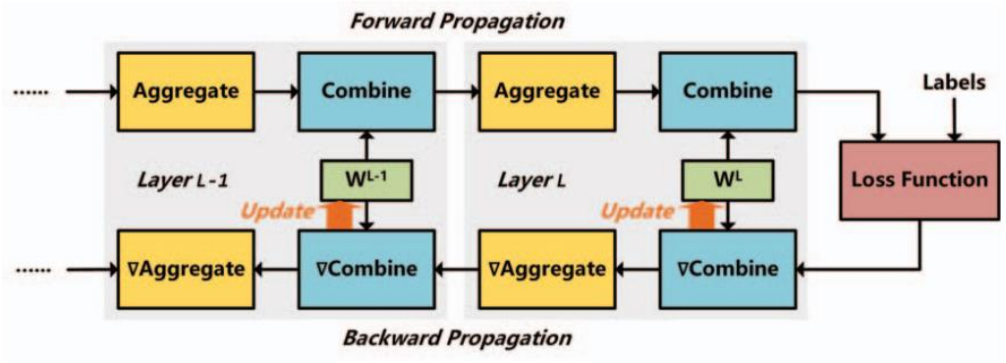


Fig. 2. Supervised GNN training process

  - After passing through all layers
    - Model generates a predicted label for each node
    - Backpropagates the gradient calculated from the loss function to update the weights and biases
  - This process is repeated until the loss function converges
  - When training is complete, the node labels of the new graph can be predicted