GDR-HGNN: A Heterogeneous Graph Neural Networks Accelerator Frontend with Graph Decoupling and Recoupling

Runzhen Xue[1,2], Mingyu Yan[1,2,3,*], Dengke Han[1,2],
Yihan Teng[1,2], Zhimin Tang[1,2], Xiaochun Ye[1,2], Dongrui Fan[1,2]
[1]State Key Lab of Processors, Institute of Computing Technology, Chinese Academy of Sciences;
[2]University of Chinese Academy of Sciences; [3]Shanghai Innovation Center for Processor Technologies;
{xuerunzhen21s,yanmingyu,handengke21s,tengyihan21s,tang,yexiaochun,fandr}@ict.ac.cn

ABSTRACT

Heterogeneous Graph Neural Networks (HGNNs) have broadened the applicability of graph representation learning to heterogeneous graphs. However, the irregular memory access pattern of HGNNs leads to the buffer thrashing issue in HGNN accelerators.

In this work, we identify an opportunity to address buffer thrashing in HGNN acceleration through an analysis of the topology of heterogeneous graphs. To harvest this opportunity, we propose a graph restructuring method and map it into a hardware frontend named GDR-HGNN. GDR-HGNN dynamically restructures the graph on the fly to enhance data locality for HGNN accelerators. Experimental results demonstrate that, with the assistance of GDR-HGNN, a leading HGNN accelerator achieves an average speedup of 14.6× and 1.78× compared to the state-of-the-art software framework running on A100 GPU and itself, respectively.

KEYWORDS

Heterogeneous Graph Neural Network, Hardware Accelerator.

ACM Reference Format:
Runzhen Xue[1,2], Mingyu Yan[1,2,3,*], Dengke Han[1,2],, Yihan Teng[1,2], Zhimin Tang[1,2], Xiaochun Ye[1,2], Dongrui Fan[1,2]. 2024. GDR-HGNN: A Heterogeneous Graph Neural Networks Accelerator Frontend with Graph Decoupling and Recoupling. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3656540

1 INTRODUCTION

The earlier success of graph neural networks (GNNs) has predominantly focused on homogeneous graphs (HomoGs), namely graphs with a singular type of vertices and edges. However, many real-world datasets in complex systems are more aptly represented as heterogeneous graphs (HetGs) [14, 20]. HetGs encompass multiple categories of entities and relations, characterized by diverse types of vertices and edges, respectively. In contrast to HomoGs, HetGs not only encapsulate structural information but also feature-rich semantic information [13].

Heterogeneous graph neural networks (HGNNs) are proposed to capture information in HetGs. They have reportedly demonstrated state-of-the-art (SOTA) performance in various crucial applications, including recommendation systems [7], medical analysis [8]. The execution of the most prevalent HGNNs can be primarily divided into four stages, namely semantic graph build (SGB) that partitions the original HetG into semantic graphs, feature projection (FP) that transforms each vertex's feature vector in semantic graphs using a multi-layer perceptron (MLP), neighbor aggregation (NA) that aggregates features from neighbors for each vertex in semantic graphs, and semantic fusion (SF) that fuses NA results across different semantic graphs for each vertex [19].

Due to the unique workflow outlined above, current hardware, such as GPUs and GNN accelerators, faces challenges in efficiently executing HGNNs. GPUs, for instance, struggle with efficiently handling irregular memory accesses stemming from the graph-topology-dependent program behavior in the NA stage [17, 18]. On the other hand, GNN accelerators tailored their hardware to GNNs, such as HyGCN [18], lack the HGNN-oriented scheduling and executing units to process the unique workflow of HGNNs [17].

Recent work [17] has proposed an HGNN accelerator, HiHGNN. This work designs a multi-lane architecture to harness parallelism between semantic graphs. Furthermore, it strategically schedules the execution order of semantic graphs based on their similarity to exploit data reusability. Compared to the SOTA solution running on A100 GPU, HiHGNN achieves an 8.3× speedup [17]. However, the efficiency of HGNNs' acceleration is still hindered by the buffer thrashing issue. This issue, characterized by a high rate of swapping between the on-chip buffer and DRAM, is caused by the irregular memory access pattern. Our evaluations indicate that this issue results in a substantial number of redundant accesses to DRAM, leading to a significant degradation in performance.

In this work, we initiate our exploration by scrutinizing the topology of semantic graphs, highlighting their general bipartite nature [4]. This observation unveils an opportunity to tackle buffer thrashing in HGNN acceleration. To seize this opportunity, we propose a hardware frontend for restructuring semantic graphs, intended for integration into existing HGNN accelerators to mitigate buffer thrashing. Our proposed graph restructuring method involves both graph decoupling and graph recoupling. Graph decoupling aims to separate the original semantic graph into a set of edges that do not share common vertices. Subsequently, graph recoupling utilizes the outcomes of graph decoupling to identify a vertex group, ensuring that every edge in the original semantic graph shares at least one vertex within this vertex group. Ultimately, the semantic graph undergoes restructuring, resulting in a series of subgraphs, each characterized by a robust community structure defined by this group of vertices.

To summarize, we list our contributions as follows:

- We quantitatively analyze the buffer thrashing issue in HGNN acceleration. Additionally, we identify an opportunity to address this issue through an in-deep observation of the HetG topology.

[*] Corresponding author is Mungyu Yan (*yanmingyu@ict.ac.cn*).

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656540

Table 1: Notations and Corresponding Explanations.

Notation	Explanation	Notation	Explanation
G	heterogeneous graph	V	vertex set
$V_{src}(V_{dst})$	source (dest) vertex set	E	edge set
u, v	vertex	e ($e_{u,v}$)	edge (from u to v)
$N(v)$	neighbor vertex set of v	$\mathcal{T}^v$	vertex type set
$\mathcal{T}^e$	edge type set	$G^{\mathcal{P}}$	semantic graph
$G_s^{\mathcal{P}}$	subgraphs restructured by $G^{\mathcal{P}}$	$\mathcal{R}$	relation

- We propose a graph restructuring method to alleviate the buffer thrashing issue and reduce unnecessary memory accesses, capitalizing on the identified opportunity.
- We intricately map the proposed method into a frontend hardware, named GDR-HGNN, which can be seamlessly integrated into the current HGNN accelerator to restructure graphs on the fly for enhanced data locality, thereby reducing DRAM accesses and improving performance.
- We conduct a comprehensive evaluation of three HGNN models and three HetG datasets. Experimental results show that, with the assistance of GDR-HGNN, a SOTA HGNN accelerator achieves an average speedup of 14.6× and 1.78×, and reduces DRAM access by 91.3% and 42.9%, compared to the SOTA software framework running on A100 GPU and itself, respectively.

2 BACKGROUND

HetGs. A graph can be defined as $G = (V, E, \mathcal{T}^v, \mathcal{T}^e)$, where V is the vertex set, E is the edge set, $\mathcal{T}^v$ is the vertex type set and $\mathcal{T}^e$ is the edge type set. A graph is HetG when $|\mathcal{T}^v| + |\mathcal{T}^e| > 2$. Table 2 lists some typical HetG datasets. Each edge type is termed as a relation $\mathcal{R} \in \mathcal{T}^e$, while an edge $e_{u,v} \in E$ starts from the source vertex u and ends at the target vertex v. For example, the relation A → M in the IMDB dataset means that an actor A acts in a movie M.

Table 2: Information of HetG Datasets.

Dataset	#Vertex	#Feature	Relations
IMDB	movie (M): 4932 director (D): 2393 actor (A): 6124 keyword (K): 7971	M: 3489 D: 3341 A: 3341 K: –	A → M M → A K → M M → K D → M M → D
ACM	paper (P): 3025 author (A): 5959 subject (S): 56 term (T): 1902	P: 1902 A: 1902 S: 1902 T: –	T → P P → T S → P P → S P → P -P → P A → P P → A
DBLP	author (A): 4057 paper (P): 14328 term (T): 7723 venue (V): 20	A: 334 P: 4231 T: 50 V: –	A → P P → A V → P P → V T → P P → T

HGNNs. To generate the final embedding of each vertex, HGNNs recursively aggregate the feature vectors of its neighboring vertices in each semantic graph and fuse the aggregated results across all semantic graphs, as illustrated in Fig. 1. The most prevalent HGNNs typically involve four stages. The SGB stage constructs semantic graphs for the subsequent stages by partitioning the HetG into a group of semantic graphs based on relations or metapaths. The FP stage projects the feature vector of each vertex in different types into the same dimensional space using an MLP within each semantic graph. The NA stage utilizes an attention mechanism to perform a weighted sum aggregation of features from neighbors within each semantic graph. The SF stage fuses the semantic information obtained from all semantic graphs, aiming to combine the results of the NA stage across different semantic graphs for each vertex.

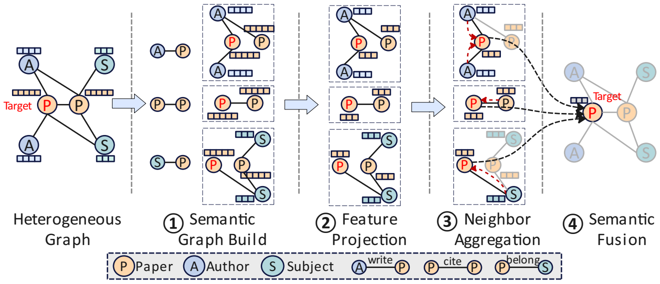


Figure 1: Illustration of HGNNs.

Differences between GNNs and HGNNs. Feature Projection: Vertices in HomoGs share the same vector space for joint feature projection; Vertices of different types in HetGs require separate feature projection. Aggregation: GNNs only perform neighbor aggregation; HGNNs use neighbor aggregation and semantic fusion.

3 MOTIVATION

This section quantitatively analyzes the buffer thrashing issue in HGNN acceleration to provide motivation for GDR-HGNN design.

This issue significantly impedes the efficiency of HGNN acceleration, especially considering that the NA stage dominates HGNNs, which constituting up to 74% of the total inference time [19]. It arises from the fact that the NA stage aggregates the features of neighboring vertices based on the irregular topology of semantic graphs, leading to irregular memory accesses to features. To delve deeper into this matter, we perform quantitative analysis using the NVIDIA T4 GPU and the SOTA HGNN accelerator HiHGNN [17].

Quantitative Analysis on T4 GPU. We conduct a quantitative experiment for the NA stage of RGCN [12] model using a state-of-the-art framework, DGL [16], running on an NVIDIA T4 GPU. The experimental results reveal that the L2 cache hit ratio in the processing of IMDB and DBLP is lower, reaching 30.1% and 17.5%, respectively. This reveals that a significant number of vertex features experience frequent replacements, contributing to the buffer thrashing issue and impairing overall performance.

Quantitative Analysis on HiHGNN. Fig. 2 gives statistics on the replacement times of vertex features from the buffer during the NA stage. The numbers on the horizontal axis represent the replacement times of vertices' features, while "Ratio of #Vertex" represents the ratio of the number of vertices with specific replacement times to the total number of vertices. Similarly, "Ratio of #Access" denotes the ratio of number DRAM accesses conducted by vertices with specific replacement times to the total number of DRAM accesses. The results indicate that a considerable number of vertex features undergo frequent replacements, contributing to the buffer thrashing issue and resulting in a substantial number of redundant DRAM accesses. This excessive data movement significantly hinders overall performance. It's worth noting the varying degrees of buffer thrashing across the three datasets, attributed to their different graph sizes. The DBLP dataset exhibits the most pronounced occurrence, primarily due to its significantly larger number of vertices compared to the other datasets.

4 DESIGN OF GDR-HGNN

In this section, we initially identify an opportunity to tackle the buffer thrashing issue. To capitalize on this opportunity, we propose a graph restructuring method to enhance data locality in HGNN

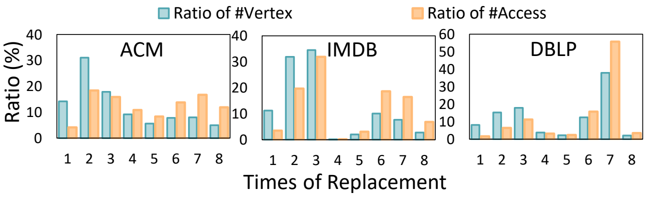


Figure 2: Analysis on HiHGNN with RGCN Model: Replacement Times of Vertices' Features during NA Stage.

acceleration. Ultimately, we design GDR-HGNN, an HGNN accelerator frontend that incorporates this graph restructuring method.

4.1 Opportunity to Address Buffer Thrashing

To address the buffer thrashing issue in HGNN acceleration, an effective idea is to identify a group of incoming vertices that share the same neighboring vertices and aggregate their neighboring vertices during the same time frame. This exploitation of data locality can alleviate the irregular accesses to DRAM, resulting in a reduction of on-chip data replacement.

As mentioned in Section 2, HGNNs do not directly operate on the original HetG. Instead, the original HetG is processed to build several directed and bipartite semantic graphs [4]. A bipartite graph can produce a maximal matching, namely largest number of edges that owning unique vertices. Based on the maximal matching, a minimal subset of vertices can be generated, within which all edges of the graph possess at least one endpoint. This subset of vertices is termed the graph backbone in this paper, and it can facilitate the classification of all vertices into four distinct parts: ❶ Src_{in}: Source vertices included in the backbone. ❷ Src_{out}: Source vertices excluded from the backbone. ❸ Dst_{in}: Destination vertices included in the backbone. ❹ Dst_{out}: Destination vertices excluded from the backbone.

By definition, there is no such edge whose endpoints are both outside the graph backbone, which derives the non-connectivity between Src_{out} and Dst_{out}, and further a partition of the original graph into three distinct subgraphs. These subgraphs demonstrate specific characteristics: all vertices connect to Src_{out} belong to Dst_{in}, while all vertices connect to Dst_{out} belong to Src_{in}.

By eliminating irrelevant vertices from each subgraph, the processing of those shared neighboring vertices within the same timeframe, optimizing buffer utilization and mitigating the issue of buffer thrashing.

4.2 Graph Restructuring Method

To harvest the above opportunity, we propose a graph restructuring method to reshape input semantic graphs, leveraging the inherent properties of bipartite graphs to improve data locality.

Fig. 3 offers a toy example to elucidate the workflow of the graph restructuring method. This method unfolds in two stages: graph decoupling and graph recoupling. In the former stage, the maximum matching algorithm is employed to identify graph backbone candidates, effectively decoupling the semantic graph into a distinct edge group. In the latter stage, the backbone is selected from this discrete edge group to reassemble the graph into three subgraphs, each characterized by a robust community structure.

Graph Decoupling. The graph decoupling focuses on discovering the maximum matching within semantic graphs to identify graph backbone candidates, as illustrated in Algorithm 1. Essentially, it draws inspiration from the Hungarian Algorithm [6]. To optimize its execution, we design customized hardware by leveraging FIFO and hash table functionalities, as detailed in Section 4.3.

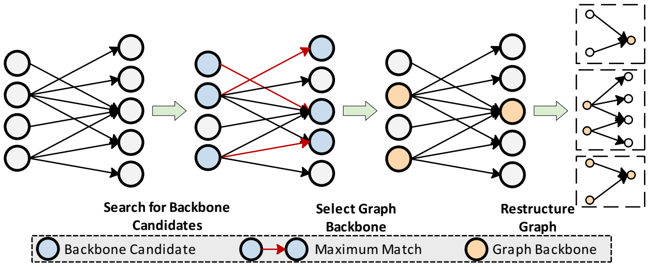


Figure 3: Toy Example of Graph Restructuring Method.

This algorithm begins by initializing all vertices without a match (line 2), subsequently scanning each vertex u to identify its neighbor v and commence the search. Initially, for each neighbor v of u, u is added to the $Matchin\_FIFO[v]$ (line 12) for temporary storage. If v is unmatched, the algorithm records vertices u and v as a matched pair and frees the previous match associated with u through iterative steps (lines 14-18). In instances where all of u's neighbors are matched, the vertices matched to u's neighbors are pushed into $Search\_List$ (lines 22-26) to find another match, leaving the match for u. After searching for the maximum match, the final matches are stored in $Match\_Pair$. Additionally, the matching vertices are recorded as graph backbone candidates, termed as M.

Algorithm 1: Graph Decoupling

```
   Input: $G^{\mathcal{P}}$: The input semantic graph;
   Output: $Match\_Pair$: Backbone candidate list;
1  $Match\_Pair$, $Search\_List$ = { };
2  Clear all $Matching\_FIFO$;
3  for each vertex $n$ in $G^{\mathcal{P}}$ do
4      if $Match\_Pair[n] < 0$ then
5          Push $n$ to $Search\_List$;
6          while $Search\_List$ is not empty do
7              Pop $u$ from $Search\_List$;
8              for each neighbor $v$ of $u$ do
9                  if $v$ is visited then
10                     continue;
11                 end
12                 Push $u$ to $Matching\_FIFO[v]$;
13                 if $Match\_Pair[v] < 0$ then
14                     while $Match\_Pair[u] > 0$ do
15                         $Matching\_FIFO[Match\_Pair[u]]$.pop();
16                         Change $Match\_Pair$;
17                         $u = Match\_Pair[Match\_Pair[u]]$;
18                     end
19                     break while;
20                 end
21             end
22             if $Match\_Pair[v] < 0$ then
23                 for each matched neighbor $u$ of $v$ do
24                     Push $Match\_Pair[u]$ to $Search\_List$;
25                 end
26             end
27         end
28     end
29 end
30 return $Match\_Pair$;
```

Graph Recouping. The graph recoupling aims to select the graph backbone from the candidates and generate subgraphs, as shown in Algorithm 2.

Initially, the algorithm initiates backbone selection. It commences by exploring all matched vertices categorized by source and destination. For each matched source vertex, the algorithm identifies its unmatched neighbors and put them to Dst_{out} while put itself to Src_{in}. If all its neighbors are matched, the algorithm will just skip to the next matched source vertex. Also, all matched destination vertices will be examined after the matched sources vertices for the same procedure. After all, all left vertices are then classified to Src_{out} or Dst_{out}, according to whether they belong to V_{src} or V_{dst}. Once the backbone is selected, subgraphs are generated for the subsequent execution.

Algorithm 2: Graph Recoupling

```
Input: $G^{\mathcal{P}}$: The input graph; $M$: Backbone candidate vertex set;
Output: $G^{\mathcal{P}}_{s_1}, G^{\mathcal{P}}_{s_2}, G^{\mathcal{P}}_{s_3}$: The subgraphs generated from original
        semantic graph.
$Src_{in}, Src_{out}, Dst_{in}, Dst_{out} = \{\}$;
$S \leftarrow V_{src} \cap M, \overline{S} \leftarrow V_{src} \setminus M, T \leftarrow V_{dst} \cap M, \overline{T} \leftarrow V_{dst} \setminus M$;
for each $v$ in $S$ do
    $X_v \leftarrow N(v) \cap \overline{T}$;
    if $X_v$ is not $\varnothing$ then
        Push $v$ to $Src_{in}$;
        Push $X_v$ to $Dst_{out}$;
    end
end
for each $u$ in $T$ do
    $X_u \leftarrow N(u) \cap \overline{S}$;
    if $X_u$ is not $\varnothing$ then
        Push $u$ to $Dst_{in}$;
        Push $X_u$ to $Src_{out}$;
    end
end
Push the other source vertices to $Src_{out}$;
Push the other destination vertices to $Dst_{out}$;
$G^{\mathcal{P}}_{s_1}, G^{\mathcal{P}}_{s_2}, G^{\mathcal{P}}_{s_3}$ = GenerateGraph($Src_{in}, Src_{out}, Dst_{in}, Dst_{out}$);
return $G^{\mathcal{P}}_{s_1}, G^{\mathcal{P}}_{s_2}, G^{\mathcal{P}}_{s_3}$;
```

4.3 Hardware Implementation

In this section, we detail the algorithm mapping to the hardware and the microarchitecture of GDR-HGNN.

Fig. 4 offers an overview of the GDR-HGNN architecture, which primarily comprises two modules: Decoupler and Recoupler. The Decoupler undertakes graph decoupling and is constructed with components of the Hash Table, FIFOs, bitmaps (Bm.), and buffers. On the other hand, the Recoupler is tasked with executing graph recoupling and consists of the Backbone Searcher, a collection of FIFOs, and the Graph Generator.

Workflow of Decoupler. In each execution epoch, the topology of original semantic graph is received and passed on to the hash table for FIFO allocation. The FIFOs, organized in a set-associative manner, store matched pairs and waiting list allocated to specific vertices. As illustrated in Fig. 5, during each cycle, source vertices are dispatched to their respective FIFOs, automatically triggering a pop operation for FIFOs if the match condition changes. The Matching Buffer stores replaced FIFO data. Upon identifying all matching edges, the resulting backbone candidates are stored in each FIFO and sent to the Candidate Buffer.

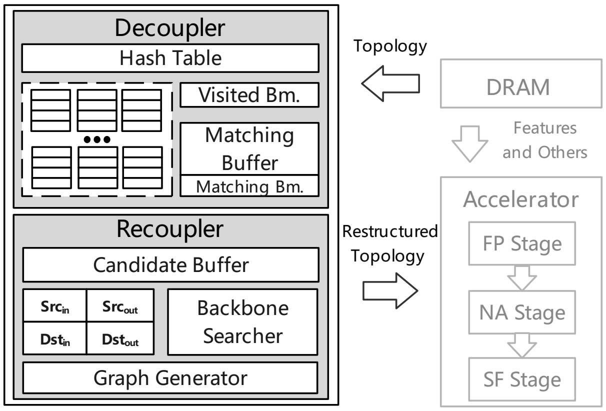


Figure 4: Design and Workflow Overview of GDR-HGNN.

Workflow of Recoupler. The identified candidates are then forwarded to the Backbone Searcher, as highlighted in Fig. 6. During this stage, the Candidate Buffer transmits the backbone candidates to the Backbone Searcher to identify the graph backbone, following Algorithm 2. Initially, each candidate is directed to the adjacency list buffers including the Src Adj. Buffer and Dst Adj. Buffer to obtain their respective neighbors. Subsequently, all obtained neighbors are checked in the Matching Bm. If any neighbors are not found in the Matching Bm., the candidate is sent to either Src_{in} or Dst_{in} FIFOs, depending on its origin, and the corresponding neighbors are dispatched to the Src_{out} or Dst_{out} FIFOs. The Graph Generator creates the subgraphs from these four designated buffers and forwards them for subsequent HGNN execution.

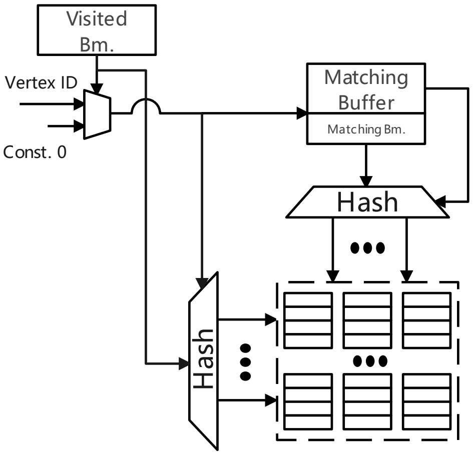


Figure 5: Micro-architecture of Decoupler.

Cooperation with HGNN Accelerators. To illustrate how to collaborate with HGNN accelerators, we present a case study using the SOTA accelerator HiHGNN. It's important to note that the graph restructuring method and its scheduling unit design can be integrated with other future HGNN accelerators for enhanced performance. Additionally, this method can be applied to subgraphs to generate smaller sub-subgraphs, thereby exploiting data locality in a smaller on-chip buffer. HiHGNN [17] employs a hybrid architecture that includes a systolic array module for matrix-vector multiplication and a SIMD module to perform element-wise operations during HGNN execution, covering FP, NA, and SF stages.

By establishing a dataflow between GDR-HGNN and HiHGNN, efficient HGNN acceleration can be achieved through the pipelining

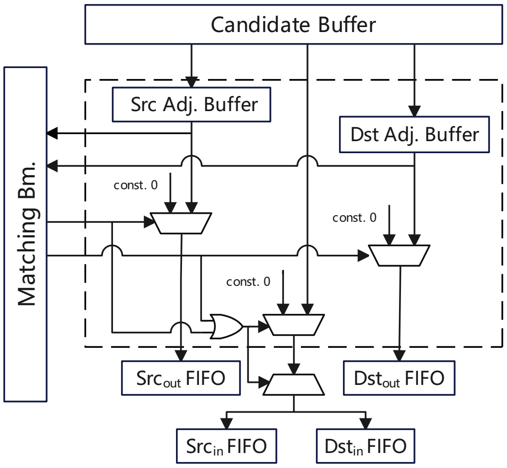


Figure 6: Micro-architecture of Recoupler.

of their execution. They operate concurrently and share the memory controller to manage data interactions between on-chip buffers and high-bandwidth memory (HBM). The process begins with the input of a semantic graph into GDR-HGNN, where the subgraphs, extracted from the FIFOs including Src_{out} and Dst_{in}, Src_{in} and Dst_{in}, and Src_{in} and Dst_{out}, are transmitted to HiHGNN. Once the graph backbone is identified. Meanwhile, GDR-HGNN continuously receives and restructures the next semantic graph.

5 EVALUATION

In this section, we compare GDR-HGNN to baselines and provide the optimization analysis in detail. We first describe our experimental setup in Section 5.1. Then, we demonstrate the advantages with the assistance of GDR-HGNN, comparing against a SOTA software framework operating on both NVIDIA T4 GPU and A100 GPU, as well as a SOTA HGNN accelerator HiHGNN in Section 5.2.

5.1 Experiment Setup

Evaluation Methodology. The performance metrics of GDR-HGNN are evaluated using the following tools.

Architecture Simulator. We implement GDR-HGNN in a cycle-level accurate simulator to measure execution time in the number of cycles. We also design a detailed cycle-accurate on-chip memory model and integrate the Ramulator [5] for FIFOs, buffers and memory simulation. This simulator models the microarchitectural behaviors of each module and the hardware datapath.

CAD Tools. We implement an RTL version of each hardware module and synthesize it to evaluate the area, energy consumption, and latency. We use the Synopsys Design Compiler with the TSMC 12 *nm* standard VT library for the synthesis and estimate the power consumption using Synopsys PrimeTime PX.

Memory Measurements. We estimate the buffer area, energy consumption, and access latency using Cacti [1]. We use four different scaling factors to convert them to 12 nm technology. The access latency and energy of HBM 1.0 are simulated by the Ramulator and estimated with 7 pJ/bit as HiHGNN, respectively.

Datasets and Models. We conduct the experiments on three different models including RGCN [12], RGAT [15] and Simple-HGN [9], using three datasets, ACM, DBLP, and IMDB, commonly used in HGNN research community [16, 17, 19]. The implementation of all models follows the specifications outlined in HiHGNN [17].

Baseline Platforms. We integrate GDR-HGNN into HiHGNN, creating the combined system HiHGNN+GDR-HGNN. We compare it with a state-of-the-art HGNN framework, DGL 1.0.2 [16], running on an NVIDIA T4 GPU, an NVIDIA A100 GPU, and HiHGNN. Table 3 lists the configurations for HiHGNN and GDR-HGNN.

Table 3: Platform Details of HiHGNN and GDR-HGNN.

	HiHGNN	GDR-HGNN
Peak Performance	16.38 TFLOPS, 1.0 GHz	–
On-chip Buffer	2.44 MB (FP-Buf), 14.52 MB (NA-Buf), 0.12 MB (SA-Buf), 0.38 MB (Att-Buf)	8 KB FIFOs, 160 KB Matching Buffer, 160 KB Candidate Buffer, 320 KB Adj. List Buffer
Off-chip Memory	512 GB/s, HBM 1.0	–

5.2 Evaluation Results

Speedup. Fig. 7 shows the speedup of A100 GPU, HiHGNN, and HiHGNN+GDR-HGNN to T4 GPU. The last set of bars, labeled as GEOMEAN, indicates the geometric mean across all HGNN models. HiHGNN+GDR-HGNN achieves an average speedup of 68.8×, 14.6× and 1.78× compared to T4 GPU, A100 GPU and HiHGNN, respectively. The performance improvement stems from two primary aspects. Firstly, the graph restructuring method facilitates the transformation of graph structures, evolving from a structure that generates a large number of random accesses to an organized form showcasing community locality. This restructure helps reduce buffer replacements, thereby enhancing overall performance. Secondly, the pipeline between Decoupler, Recoupler and accelerator ensures uninterrupted utilization of intermediate data. This design further reduces buffer replacements, leading to an overall performance boost.

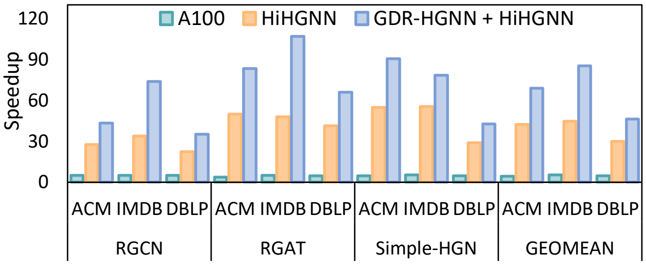


Figure 7: Speedup to T4 GPU.

Number of DRAM Accesses. To analyze the source of performance improvement, Fig. 8 presents the normalized DRAM access to T4 GPU. HiHGNN+GDR-HGNN accesses only 4.8%, 8.7%, and 57.1% compared to T4 GPU, A100 GPU, and HiHGNN, respectively. This result confirms that with the assistance of GDR-HGNN, HiHGNN significantly reduces the number of DRAM accesses, validating the source of speedup.

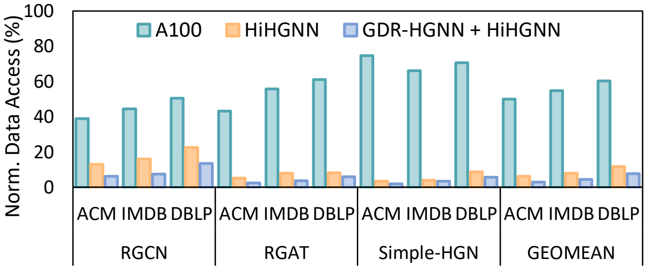


Figure 8: Normalized DRAM Access to T4 GPU.

Utilization of DRAM Bandwidth. Fig. 9 shows the utilization of the DRAM bandwidth of GDR-HGNN+HiHGNN and the baselines. GDR-HGNN+HiHGNN demonstrates 2.58× and 6.35× improvement on average in the utilization of DRAM bandwidth compared with T4 GPU and A100 GPU, respectively. In contrast to HiHGNN, HiHGNN+GDR-HGNN notably diminishes DRAM accesses using the graph restructuring method. However, this improvement comes with a marginal trade-off affecting overall bandwidth utilization, primarily due to increased strain on compute resources.

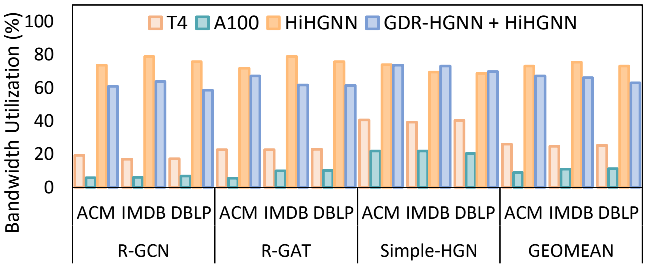
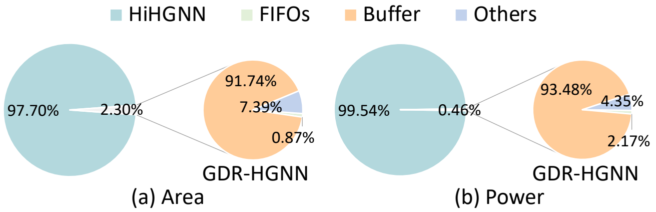


Figure 9: DRAM Bandwidth Utilization.

Area and Power Overhead. Fig. 10 displays the area and power characteristics of HiHGNN and GDR-HGNN. The results indicate that GDR-HGNN accounts for only 2.30% (i.e., 0.50 mm^2) and 0.46% (i.e., 55.6 mW) of the total area and power when combined with HiHGNN under TSMC 12 nm technology. This validates that the overhead of the graph restructuring method can be disregarded. The primary overhead of GDR-HGNN originates from buffers used to store edge and vertex indices for the restructure.



(a) Area (b) Power

Figure 10: Area and Power of HiHGNN and GDR-HGNN.

6 RELATED WORK

Given the remarkable learning capacity of GNNs with graph data, GNN accelerators have garnered significant attention from the architecture community [3, 10, 11, 18]. In terms of data locality exploitation, I-GCN [3] introduces a potent method called "islandization" to enhance data locality in GNNs. This method identifies clusters of vertices with strong internal connections but weak external connections. However, this method is not suitable for directed bipartite graphs, as the properties of such graphs cause the employed strategy to degrade into a process focused solely on finding the vertex with the largest degree.

Previous efforts [2, 17] have proposed several accelerators for HGNN acceleration. HiHGNN [17] strategically schedules the execution order of semantic graphs based on their similarity to exploit data reusability across different semantic graphs. MetaNMP [2] pioneers DIMM-based near-memory processing for HGNNs. It employs a cartesian-like product paradigm to dynamically generate metapath instances and aggregate vertex features from the starting vertex along these metapaths. In contrast, our work focuses on alleviating buffer thrashing issues by leveraging the opportunity presented by the semantic graphs in HGNNs.

7 CONCLUSION

This work identifies an opportunity for data locality exploitation via an in-depth analysis of the unique characteristics of HetGs. It introduces GDR-HGNN, designed to leverage this opportunity to address the buffer thrashing issue during HGNN execution. Experimental results show that GDR-HGNN outperforms state-of-the-art efforts and substantially reduces DRAM accesses.

ACKNOWLEDGMENT

This work was supported by National Key Research and Development Program (Grant No. 2022YFB4501400), the National Natural Science Foundation of China (Grant No. 62202451), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-029), and CAS Project for Youth Innovation Promotion Association.

REFERENCES

[1] [n. d.]. CACTI. http://www.hpl.hp.com/research/cacti/
[2] Dan Chen, et al. 2023. MetaNMP: Leveraging Cartesian-Like Product to Accelerate HGNNs with Near-Memory Processing. In *Proceedings of the 50th Annual International Symposium on Computer Architecture.* 1–13.
[3] Tong Geng, et al. 2021. I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement through Islandization. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21).* Association for Computing Machinery, New York, NY, USA, 1051–1063.
[4] Weihua Hu, et al. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020).
[5] Yoongu Kim, et al. 2015. Ramulator: A fast and extensible DRAM simulator. *IEEE Computer architecture letters* 15, 1 (2015), 45–49.
[6] Harold W Kuhn. 1955. The Hungarian method for the assignment problem. *Naval research logistics quarterly* 2, 1-2 (1955), 83–97.
[7] Ansong Li, et al. 2022. Disentangled graph neural networks for session-based recommendation. *IEEE Transactions on Knowledge and Data Engineering* (2022).
[8] Feng Luo, et al. 2021. IMAS++ An Intelligent Medical Analysis System Enhanced with Deep Graph Neural Networks. In *Proceedings of the 30th ACM International Conference on Information & Knowledge Management.* 4754–4758.
[9] Qingsong Lv, et al. 2021. Are we really making much progress? Revisiting, benchmarking and refining heterogeneous graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining.*
[10] Ruibin Mao, et al. 2023. ReRAM-based graph attention network with node-centric edge searching and hamming similarity. In *60th ACM/IEEE Design Automation Conference, DAC 2023, San Francisco, CA, USA, July 9-13, 2023.* IEEE, 1–6.
[11] R. Sarkar, et al. 2023. FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* IEEE Computer Society, Los Alamitos, CA, USA, 1099–1112.
[12] Michael Schlichtkrull, et al. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference.* Springer, 593–607.
[13] Chuan Shi, et al. 2016. A survey of heterogeneous information network analysis. *IEEE Transactions on Knowledge and Data Engineering* 29, 1 (2016), 17–37.
[14] Hanrui Wang, et al. 2020. GCN-RL circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *2020 57th ACM/IEEE Design Automation Conference (DAC).* IEEE, 1–6.
[15] Kai Wang, et al. 2020. Relational Graph Attention Network for Aspect-based Sentiment Analysis. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics.* 3229–3238.
[16] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop.*
[17] Runzhen Xue, et al. 2023. HiHGNN: Accelerating HGNNs through Parallelism and Data Reusability Exploitation. *arXiv preprint arXiv:2307.12765* (2023).
[18] Mingyu Yan, et al. 2020. Hygcn: A gcn accelerator with hybrid architecture. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 15–29.
[19] Mingyu Yan, et al. 2022. Characterizing and Understanding HGNNs on GPUs. *IEEE Computer Architecture Letters* 21, 2 (2022), 69–72.
[20] Michihiro Yasunaga, et al. 2019. Scisummnet: A large annotated corpus and content-impact models for scientific paper summarization with citation networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 7386–7393.