I-GCN: A Graph Convolutional Network Accelerator with Runtime Locality Enhancement through Islandization

Tong Geng[†], Chunshu Wu[‡], Yongan Zhang[§], Cheng Tan[†], Chenhao Xie[†], Haoran You[§], Martin C. Herbordt[‡], Yingyan Lin[§], Ang Li[†]

† Pacific Northwest National Laboratory, Richland, WA
‡ Boston University, Boston, MA
§ Rice University, Houston, TX
{tong.geng, cheng.tan, chenhao.xie, ang.li}@pnnl.gov,{happycwu, herbordt}@bu.edu,{yz87,hy34,yingyan.lin}@rice.edu

ABSTRACT

Graph Convolutional Networks (GCNs) have drawn tremendous attention in the past three years. Compared with other deep learning modalities, high-performance hardware acceleration of GCNs is as critical but even more challenging. The hurdles arise from the poor data locality and redundant computation due to the large size, high sparsity, and irregular non-zero distribution of real-world graphs.

In this paper we propose a novel hardware accelerator for GCN inference, called I-GCN, that significantly improves data locality and reduces unnecessary computation. The mechanism is a new online graph restructuring algorithm we refer to as *islandization*. The proposed algorithm finds clusters of nodes with strong internal but weak external connections. The islandization process yields two major benefits. First, by processing islands rather than individual nodes, there is better on-chip data reuse and fewer off-chip memory accesses. Second, there is less redundant computation as aggregation for common/shared neighbors in an island can be reused. The parallel search, identification, and leverage of graph islands are all handled purely in hardware at runtime working in an incremental pipeline. This is done without any preprocessing of the graph data or adjustment of the GCN model structure. Experimental results show that I-GCN can significantly reduce off-chip accesses and prune 38% of aggregation operations, leading to performance speedups over CPUs, GPUs, the prior art GCN accelerators of 5549×, 403×, and 5.7× on average, respectively.

CCS CONCEPTS

• **Computer systems organization → Neural networks**; **Parallel architectures**; • **Computing methodologies → Parallel algorithms**.

ACM Reference Format:
Tong Geng, Chunshu Wu, Yongan Zhang, Cheng Tan, Chenhao Xie, Haoran You, Martin C. Herbordt, Yingyan Lin, Ang Li. 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), October 18–22, 2021, Virtual Event, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3466752.3480113

MICRO '21, October 18–22, 2021, Virtual Event, Greece

ACM ISBN 978-1-4503-8557-2/21/10…$15.00
https://doi.org/10.1145/3466752.3480113

1 INTRODUCTION

Conventional deep learning paradigms such as Convolution Neural Networks (CNNs) [26] and Recurrent Neural Networks (RNNs) [33] have been demonstrated to be quite efficient, but primarily for applications using Euclidean data [15, 16, 42, 44]. Many other applications, however, require that the relationships between data points be conserved and must therefore use non-Euclidean data structures such as graphs [10, 32, 36, 46, 49, 56]. To fill this need, Graph Neural Networks (GNN) have been proposed [6, 11, 25, 50].

Graph Convolutional Networks (GCNs) are a type of GNN that has drawn tremendous attention in the past three years due to their unique ability to extract latent information from graph data. Practical applications of GCNs include prediction of cascading power-grid failure [32], E-commerce [49], and etc [10, 36]. The deployment of GCNs in these applications typically poses strict constraints on latency and throughput.

To satisfy the increasingly stringent performance requirements, designing high-performance hardware accelerators for GCNs becomes necessary and urgent [14, 48]. Real world graphs tend to have large size, high sparsity, and extremely unbalanced non-zero distributions; therefore, the direct application of existing methods, such as Sparse CNNs (SCNNs) [19, 23, 52], has been reported to be insufficient [2, 17, 27, 45].

We briefly discuss the performance challenges of the two major graph aggregation methods used in GCN acceleration:

(1) In **PULL-based aggregation** nodes are evaluated sequentially, but for each node: the neighbor features are gathered (i.e., pulled) simultaneously for aggregation. The advantage of the *pull* method is that since nodes are processed in order, a small buffer is sufficient to accommodate the aggregation results. In other words, it shows good reuse for the result matrix. The major problem, however, is the *poor data locality for accessing the feature matrix*. Given that the adjacency matrix of real-world graphs is typically very sparse and imbalanced, parallel fetches of the corresponding neighbor features can be random and non-coalesced. Since the feature matrix can be too large to fit into on-chip memory, repeated irregular off-chip data accesses for the feature matrix are required; this process is bounded by off-chip bandwidth. HyGCN [48] uses the PULL approach. Although run-time data-aware sparsity-elimination hardware is used

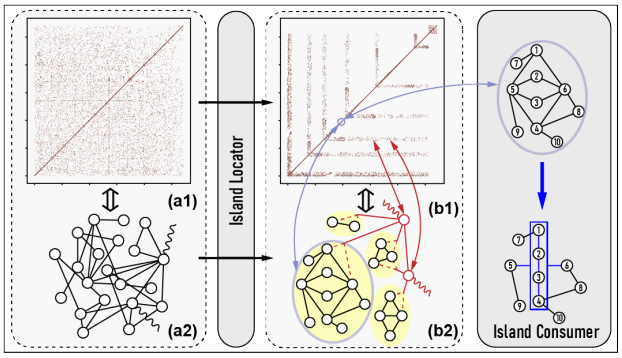

Figure 1: Workflow of I-GCN.

to reduce off-chip data accesses, the feature matrix still needs to be accessed many times. An HBM is required to avoid hardware starvation.

(2) In **PUSH-based aggregation** nodes are evaluated in parallel, but the feature data of the nodes are distributed sequentially (i.e. pushed) to their neighbors for aggregation. This avoids the irregularity in accessing the feature matrix, but essentially shifts the burden to the result matrix. Given that aggregation for a node is processed sequentially, a large buffer is required to hold the partial results. If there are too many nodes, the partial result buffer cannot fit into on-chip memory, leading to frequent off-chip access and bandwidth saturation. With varying numbers of neighbors, accesses to off-chip memory can be irregular and random, which can further cause workload imbalance. AWB-GCN [14] practices the PUSH approach. Although a run-time workload autotuning architecture is used to tackle the workload imbalance problem, it assumes sufficient on-chip storage and off-chip bandwidth for accessing the partial result buffer, which is not necessarily the case for large graphs.

Clearly, the key problem is the irregularity (i.e., distribution of non-zeros) of the adjacency matrix, which leads to poor data reuse in accessing either the feature matrix or the result matrix. A recent trend therefore is to rely on offline preprocessing to restructure the adjacency matrix to improve data locality, e.g. in Rubik [8] and GraphACT [51]. The implicit assumption is that the graph structure (i.e., the adjacency matrix) is fixed upon inference so that the large overhead of graph restructuring can be omitted in the critical path. However, this is not always the case, as real-world graphs are frequently updated (e.g., evolving graphs) or generated dynamically (e.g., inductive graphs). The high restructuring overhead, e.g., seconds in Rubik and GraphACT, is not tolerable when processed online. Besides, although both Rubik and GraphACT allow for the possibility of computation reuse for shared neighbors during aggregation, their complex software-based reordering algorithms introduce significant delay and are only feasible when processed offline.

In this paper, we propose a novel hardware accelerator, called I-GCN, which implements a new online graph restructuring algorithm – *islandization* – that can significantly improve data locality and reduce redundant computation for GCN inference acceleration. Specifically, I-GCN's *Island Locator* module, at runtime, is able to detect the *hub* nodes (i.e. nodes with high degree), mask them and their edges from the graph, and then iteratively find islands from the remaining nodes. *Islands* are groups of nodes with strong internal, but no external, connections other than with hubs. Note that islands often (but not always) have practical semantics: in a social network they might correspond to people working in the same institute; in a citation network they might correspond to papers published in the same conference series. Determining islands is non-obvious, especially in typical (huge and sparse) adjacency matrices.

Figure 1 gives an overview of I-GCN. **(i) Island Locator:** after identifying the islands, the non-zeros of the adjacency matrix become highly clustered – the none-zeros of hubs and islands form the L-shapes and the anti-diagonal respectively. **(ii) Island Consumer:** using the hub and island information, the Island Consumer performs aggregation and combination in a fine-grained pipelined manner. Redundant aggregation is skipped. This process continues until all nodes are determined to be either hubs or islands. The benefits of islandization are two-fold: (1) *Improving on-chip data locality.* Through clustering, accesses to the feature and result matrices can be constrained within a much smaller working-set (i.e., each L-shape and island in Figure 1). This greatly improves on-chip data reuse and avoids a tremendous number of off-chip accesses. (2) *Reducing redundant computation.* After clustering, nodes within a cluster tend to share a large portion of common neighbors. During GCN aggregation, rather than repeatedly counting each neighbor, aggregated information about the common neighbors as a whole can be distributed and reused, avoiding repeated aggregation calculation for common neighbors. This neighbor-sharing information is mostly ambiguous in the raw adjacency matrix, but becomes obvious after islandization (see node 1~4 in Figure 1). In summary, islandization resolves the locality issues in both Pull and Push approaches.

This is the first work, to the best of our knowledge, that tackles the fundamental data locality problem of GCN acceleration and efficiently skips redundant aggregation through online hardware-based graph restructuring. This paper makes the following contributions:

- We propose a novel islandization algorithm for efficient runtime parallel graph restructuring, which can significantly improve on-chip data locality.
- We design a new hardware accelerator architecture called I-GCN that effectively implements the islandization algorithm, harvesting the data locality exposed through islandization and avoiding redundant aggregation among shared neighbors.
- Experimental results show that I-GCN can significantly reduce off-chip accesses and prune 38% of aggregation operations, leading to performance speedups over PyG- & DGL-based CPUs, PyG- & DGL-based GPUs, and prior art GCN accelerators of 9568× & 1243×, 368× & 453×, and 5.7×, respectively.

2 BACKGROUND AND MOTIVATION

We first introduce GCN algorithms and then elaborate the existing GCN processing methodologies and their challenges.

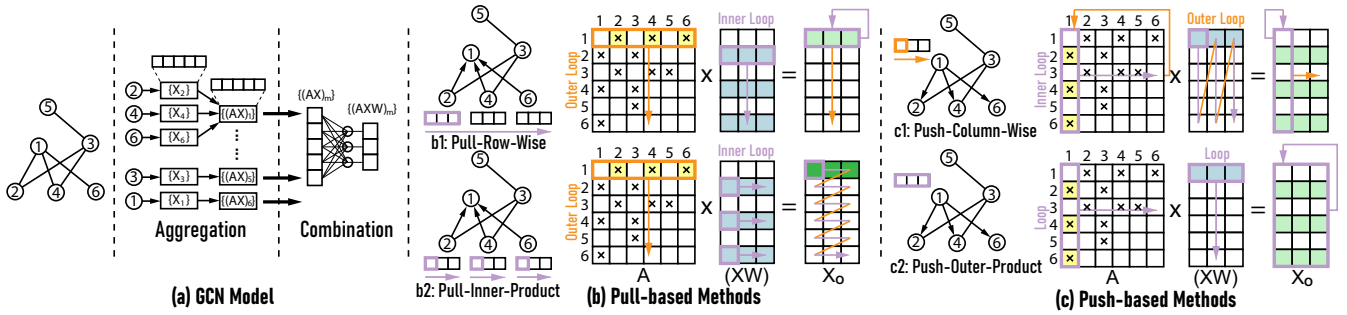

Figure 2: PULL-based (Row-wise & Inner-Product) and PUSH-based (Column-wise & Outer-Product) methods.

2.1 Graph Convolutional Networks

GCNs are composed of stacked GraphCONV layers. The computation flow of each GraphCONV layer includes two phases: *Aggregation* and *Combination*, as illustrated in Figure 2-(a). In the aggregation phase, each node gathers and aggregates features of its neighbor nodes to update the local feature-vector. In the combination phase, the updated feature-vectors are further merged to extract high-level abstraction through a local Multi-Layer Perceptron (MLP) network. From the perspective of linear algebra, the layer-wise forward propagation of GCN can be expressed as Equation 1:

$$X^{(l+1)} = \sigma(\tilde{A}X^{(l)}W^{(l)}) \tag{1}$$

where $\tilde{A}$ is the graph adjacency matrix. $X^{(l)}$ is the matrix of input features in layer-l; W^l is the weight matrix of layer-l. $\sigma(.)$ denotes non-linear activation functions [26].

There are many modalities beneath the umbrella of GCNs, e.g. GraphSage [7, 18] and Graph Isomorphism Network (GIN) [47]. As shown in [21], the forward propagation of most GCNs can be abstracted and expressed as Equation 1.

2.2 Design Space Exploration

We first discuss the design choices related to execution order of the two phases in GraphCONV. We then present the design choices within each phase.

2.2.1 Execution Order of a GraphCONV Layer. There are two alternative computation orders for a GraphCONV layer: *aggregation first* ($(AX) \times W$) and *combination first* ($A \times (XW)$). Existing work [14, 30] has shown that the combination first approach can reuse the same Sparse-dense Matrix Multiplication (SpMM) kernel for both multiplications XW and $A(XW)$ and incorporates less arithmetic computation. I-GCN follows this combination first approach.

2.2.2 Aggregation Phase. We first discuss the design choices of the aggregation phase, which is normally the performance bottleneck of GCN processing. There are two typical methods of aggregation: PULL and PUSH. For clarity, we compare them from a linear algebra perspective as illustrated in Figure 2. We discuss the correspondence between PULL/PUSH-based graph computation methods and four SpMM approaches as summarized in [39]: inner-product, outer-product, row-wise, and col-wise.

PULL-based methods aggregate nodes sequentially. For each particular node, the feature vectors of all its neighbors are gathered and aggregated in two ways: (1) PULL-Inner-Product: features from different channels are calculated sequentially, analogous to the inner-product approach of SpMM. (2) PULL-Row-Wise: features from all channels are calculated in parallel, similar to the row-wise-product approach of SpMM. As shown in Figure 2-(b), PULL-based methods always process non-zeros of matrix A and produce the aggregation result matrix X_o by rows (***outer loop***). To calculate each row of X_o (***inner loop***), PULL-Row-Wise (Figure 2-(b1)) fetches the entire feature vectors (i.e. entire rows of XW) of required nodes and performs vector accumulation sequentially; in contrast, PULL-Inner-Product fetches features by channel (i.e. column of XW) and computes the output features in X_o sequentially (Figure 2-(b2)).

PULL-based methods have their advantages and disadvantages. On the plus side, they reuse matrix A and only require relatively small on-chip buffers to conserve the partial aggregation results (since they are produced row by row). However, they both suffer from poor data reuse of matrix XW. Due to the scattered and irregular distribution of non-zeros in matrix A, the rows to be accessed in XW can be random. Given that the height of XW equals the number of nodes in the graph, which can be very large, XW (which is dense) can rarely be stored on-chip, leading to frequent data fetches and inferior performance. Ideally, if the non-zeros from the same column of A can be clustered, they can reuse the same row of XW, improving data locality.

PULL-Row-Wise is more popular than PULL-Inner-Product in prior art mainly due to: (1) accessing the entire rows is more efficient than randomly fetching elements from different columns in terms of off-chip access; (2) as XW is dense, the processing of a row in a fixed size can be parallelized without introducing workload imbalance.

PUSH-based aggregation, in contrast, calculates the aggregated features of all nodes simultaneously by broadcasting features of each node to all its neighbors one after another. Once a node receives the features from its neighbors, it updates its local feature vector. There are two ways of feature broadcasting: PUSH-Column-Wise and PUSH-Outer-Product. With PUSH-Column-Wise, the input features are broadcast by channel; the output features are calculated by channel. As shown in Figure 2-(c1), at iteration k of ***outer loop***, each node only pushes its features at channel k (i.e. column k of XW) to the neighbors. Once all nodes have broadcast their features at channel k to the neighbors (***inner loop***), every node has the complete aggregated features at channel k (i.e. column k

	On-chip Storage	Off-chip Access	Reuse XW	Reuse A	Reuse X_o	Load Imbalance	Redundancy Removal
PULL	Low	High	Low	High	High	No	Hard
PUSH	High	High	High	Low	Low	Yes	Hard
Islandization	Low	Low	High	High	High	No	Easy

Table 1: Comparison of PULL, PUSH, Island methods.

of X_o). In contrast, PUSH-Outer-Product (Figure 2-(c2)) method processes all channels at the same time. The entire input feature vector of each node is broadcast to all its neighbors in parallel. Once done, the feature aggregation of the entire graph finishes. In the matrix perspective, PUSH-Outer-Product processes the non-zeros of matrix A by column, accesses the features of XW by row, and updates the corresponding partial results of X_o with respect to the row IDs of non-zeros in matrix A.

The advantage of PUSH-based methods over PULL is that the data of XW can be fully reused via feature broadcasting. However, the accesses to X_o then become scattered and random. Given the height of X_o is equal to the number of nodes in the graph, which can be large, even a single column of X_o may not be able to be buffered on-chip. Frequent accesses to X_o thus render repeated data fetches from off-chip DRAM. Ideally, if the non-zeros from the same row of A can be clustered, they can reuse the same row of X_o, improving data locality and performance.

PUSH-Column-Wise method is more popular than PUSH-Outer-Product in prior art because for graphs that a single column of X_o can be put on-chip, PUSH-Column-Wise method can avoid the repeated off-chip data accesses over X_o. However, this does not work for large graphs. Consequently, PUSH-Column-Wise does not fundamentally handle the data locality issue. Additionally, it needs to repeatedly access matrix A which also incurs additional off-chip access.

Table 1 summarizes the advantages and disadvantages of both approaches. The proposed islandization method is capable of overcoming all the drawbacks by clustering the non-zeros at runtime and provides nearly optimal data reuse in GCN inference. Furthermore, with islandization, the unnecessary aggregation for commonly shared neighbors can be identified much more easily for effective pruning.

2.2.3 Combination Phase. As both combination and aggregation are based on SpMM kernels, the combination phase shares similar issues with aggregation phase. The only difference is that the weight matrix W in combination is normally much smaller than the feature matrix XW in aggregation, and can be more likely stored on-chip. Therefore, the data locality issue of the PULL-based method is less prominent than in aggregation. Consequently, I-GCN adopts the PULL-based method for combination.

3 ALGORITHMS AND ARCHITECTURES

This section first presents the overall workflow and hardware architecture of I-GCN. It then introduces the algorithms and the corresponding architectures of two major components of I-GCN: the *Island Locator* and the *Island Consumer*.

3.1 I-GCN Overview

3.1.1 I-GCN Workflow. Figure 3 illustrates how I-GCN improves data locality through islandization: the clustering of graph nodes

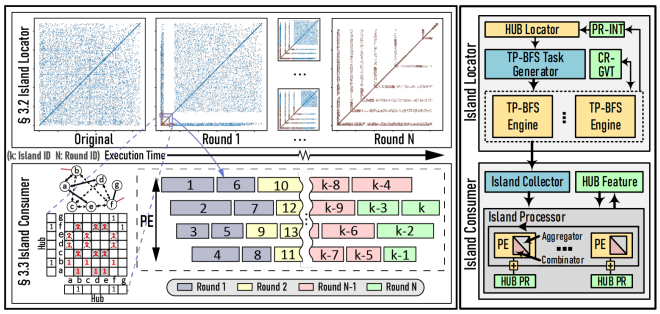

Figure 3: Detailed I-GCN workflow and overview of I-GCN architecture

into islands. Elements of an island tend to have strong internal connections, and are connected to other islands through ***hubs*** – nodes with high fan-in/out degrees, which act as the points of contact for islands. Islandization is the process of finding these structures hidden in the original graph. At high level, I-GCN processing begins by locating these islands, after which it processes the graph at the granularity of hubs and islands rather than nodes, thus being able to improve locality and reduce redundant computation.

I-GCN creates the islands gradually by round. Figure 3 shows the process using the CORA dataset. The scattered non-zeros belonging to the same island are clustered around the diagonal. The islands (including the hubs and nodes) are discovered in parallel by a hardware module called the **Island Locator**. In each round, the Island Locator uses different thresholds to recognize hubs and then shapes the island by scanning the hubs' neighbor nodes. As shown in Figure 3, the processing moves from the bottom-left corner of the adjacency matrix to the upper-right corner, along the anti-diagonal. In each round a portion of the non-zeros are gathered into an L-shape cluster (except the non-zeros around the anti-diagonal), leaving the remaining non-zeros untouched. With new hubs located at each round, new L-shape clusters are formed. This process continues until all non-zeros are clustered either into L-shaped islands or around the anti-diagonal (see *Round N* in Figure 3).

In parallel, whenever an island is formed its adjacency information is forwarded to the second module called **Island Consumer**. Island Consumer then processes the island as a small but dense sub-graph, fetches its node features, and performs the required combination and aggregation of the GCN. Note that although the Island Locator forms the islands gradually by round, and requires per round synchronization, the Processing Elements in the Island Consumer can process an island as soon as it is formed without synchronization at the completion of each round: it does not need to wait until all the islands in an islandization round are formulated. I-GCN overlaps graph restructuring and graph processing.

As shown in Figure 3, each node in an island, labeled as an island node, only connects to the nodes of the same island and the hub nodes connected to the island. This ensures that the space between the L-shapes is purely blank. Therefore, when processing a GraphCONV layer, the adjacency and feature data of an island node are only needed when the island is being processed. Consequently, they only need to be fetched from off-chip once. The hubs do have the chance of being used multiple times during the processing of different islands and inter-hub connections. However, since hubs

are normally a small fraction of the entire graph, their associated data will likely be stored on-chip and sufficiently reused. Even if the hubs' associated data is too large to fit in the on-chip memory, our method still reduces off-chip data movement.

In summary, for GNN processing of real-world graphs with component structures using I-GCN, most data are fetched only once, except the adjacency data of some island nodes which may need to be accessed multiple times during the multi-round island locating. We evaluate this in Section 4.2. Note that component structures are commonly observed in real-world graphs.

Another benefit of islandization is that it assembles the redundant aggregation among shared neighbors in the processing of each island. These redundant operations, originally hidden in the large graph, are illuminated after islandization.

As shown in Figure 3, islands are formed by nodes with intensive internal links, most of which are among shared neighbors. During the processing of the small and dense islands, it becomes easy to recognize these repeated computations and skip them. As an example, Figure 3(A) shows the sub-graph structure of the sixth island found in the first round. The bitmap at the lower-left corner is the purple block in the Round 1 adjacency matrix enlarged. The two hub vectors are from the L-shape, while the 7×7 matrix includes the adjacency data of the island node. During the evaluation of the island, Island Consumer will find the redundant aggregation operations for the shared neighbors and skip them. The redundancy removal methodology is discussed in detail in Section 3.3.

3.1.2 I-GCN Overall Architecture. The overall architecture of I-GCN, including Island Locator and Island Consumer, is shown in Figure 3(B). The HUB Locator in the Island Detector is responsible for recognizing hubs. The hubs found are forwarded to TP-BFS Task Generator. TP-BFS is short for **Threshold-based and Parallel Breadth-First Search**. Task Generator will generate and assign BFS tasks for islandization. These tasks are conducted by TP-BFS Engines. Once TP-BFS locates an island, the related adjacency and node ID information are forwarded to Island Collector in Island Consumer. Island Collector distributes the island information to an idle PE for performing its combination and aggregation jobs. Furthermore, Island Collector also generates and distributes new tasks which include inter-hub connections based on the hub information collected by TP-BFS engines at each round. Detailed architectures are presented next.

Algorithm 1 Island Locator Algorithm; Th1, Th2, and Th3 are executed concurrently and asynchronously.

1: **Inputs:** N: Node list of the input graph; $(P1, P2)$: parallel factors for hub detection and island searching; TH_o: initial threshold for hub nodes; c_{max}: the max number threshold of the searched nodes; Decay(): hub threshold decay function
2: **Outputs:** $l_{islands}$: list of islands' nodes and hubs info
3: $TH_{tmp} = TH_o$; $l_{islands} = \{\}$
4: **while** $|N| > 0$ **do**
5: $task = \{\}$; $hub\_buffer = \{\}$
pop hub nodes from graph N to hub_buffer
6: Th1: **detect_hub**(N, $P1$, TH_{temp}, hub_buffer)
pop neighbors of nodes in hub_buffer to $task$
7: Th2: **task_assign**(hub_buffer, $task$)
explore islands from nodes in $task$ using BFS
8: Th3: **TP-BFS**($task$, $P2$, TH_{temp}, c_{max}, $l_{islands}$)
9: **[Hold until parallel Th1/2/3 finish]**
10: $TH_{tmp} = \text{Decay}(TH_{tmp})$
11: **return** $l_{islands}$

Algorithm 2 detect_hub: sweep nodes and move nodes with degrees larger than thresholds to container hub_buffer

1: **Inputs:** N: input graph node list; $P1$: parallel factor; TH_{tmp}: threshold for hub nodes' degree; hub_buffer: container of the hub nodes
2: **Outputs:** None (modify hub_buffer in place)
3: **for** $b = 0$ **to** $\lceil |N|/P1 \rceil$ **do**
4: **for** $p = 0$ **to** $P1$ **in parallel do**
5: Check node $N[b * P1 + p]$ as n_o
6: **if** $n_o \in l_{islands}$ **then**
7: Pop n_o from N
8: **else if** $n_o.degree \geq TH_{tmp}$ **then**
9: Pop n_o from N to hub_buffer

Algorithm 3 task_assign: pop nodes from hub_buffer and add them and their neighbors to task queue

1: **Inputs:** hub_buffer: container of the hub nodes; $task$: container of nodes to be chosen as potential starting points for BFS
2: **Outputs:** None (modify $task$ in place)
3: **while** $|hub\_buffer| > 0$ **do**
4: Pop a node from hub_buffer
5: Append the popped hub node and each of its neighbors to $task$, in the form of tuples.

3.2 Island Locator

We first introduce the algorithm used in the Island Locator and then present its architectural support. For clarity we use a homemade graph (Figure 4) to illustrate the process.

3.2.1 Algorithm. Algorithm 1 describes the simplified workflow of the Island Locator; a toy example can be found in Figure 4. Overall the purpose of the algorithm is to locate the hubs and use their neighbors as starting points to search for islands (with the **TP-BFS** algorithm). To boost parallelism, the algorithm comprises three concurrent asynchronous tasks: *hub detection* (line 6), *BFS task generation* (line 7), and *TP-BFS execution* (line 8). Hub detection and TP-BFS are additionally performed in parallel across the $P1$ and $P2$ **parallel for loops** in algorithm 2 (line4) and 4 (line5). As mentioned in Section 3.1, locating islands is performed by rounds, which are iterations of line 4. TH_{tmp} represents the most current hub detection threshold, which is modified at run-time. Note that synchronization is required among the three tasks at the start of each round (line 9). Specifically, the algorithm takes the input of (1) node list N; (2) parallel factors $P1$ and $P2$, which define the numbers of the parallel FIFOs and TP-BFS engines, respectively; (3) the initial hub threshold TH_o, which marks the initial minimum degree of the hub nodes; (4) the maximum number of nodes in an island, c_{max}; and (5) Decay(), which defines the function to decrease the hub detection threshold. The algorithm then outputs the abstract container $l_{islands}$, which encapsulates all island-related information. This includes island nodes' indices and neighbors, the connected hub nodes' indices and neighbors, the number of all the

islands and hub nodes, and etc. Note that this abstract container is used just for clarity.

Island location starts with parallel hub detection. This is shown in Algorithm 2, which uses a threshold-based method to find hubs by rounds. If the degree of a node is above the current threshold TH_{tmp}, the node is identified as a hub and inserted into a container(buffer) hub_buffer. TH_{tmp} is reduced each round (line 10 of Algorithm 1) until all nodes are classified as island or hub nodes. To accelerate parallel hub detection, in each round we remove the nodes already classified as hub and island nodes in the previous round. At the end of island location, the node list N should be empty.

Hub detection is followed by BFS task generation (Algorithm 3). Once a hub node is detected, the Island Locator finds its neighbors by accessing its adjacency list and caches these neighbor nodes in a task queue, $task$. The Island Locator sends neighbor nodes to the task queue, $task$, and then to parallel BFS engines where the nodes are used by TP-BFS as the starting points for forming islands. Here we use neighbor nodes as the starting nodes (instead of hubs) in order to extract higher parallelism from TP-BFS. This significantly improves the scalability of the Island Locator.

Once the first task is generated and stored in the task queue, TP-BFS starts (see Algorithm 4). The algorithm keeps track of the number of nodes whose neighbors have been explored exhaustively ($query$) and the total number of visited nodes ($count$). Once $query$ catches up with $count$, an island is found with all the nodes and their neighbors within the island completely explored and the island information is returned. Multiple TP-BFS engines are able to work in parallel on different neighbors of different hubs. Multiple v_{local}s are used to keep track of the nodes visited locally by each TP-BFS engine while v_{global} is used to keep track of the nodes visited by any of the TP-BFS engines.

Each engine begins island searching by (a) recording the initial node, a_o, as the first visited node in a locally visited list, v_{local}; (b) setting the $query$ pointer to zero indicating no node's neighbors have been fully explored; and (c) accessing the node's neighbors. Note that if an engine finds a_o also a hub, it will drop the task and forward this inter-hub connection information to Island Collector.

As the Island Locator overlaps hub detection and TP-BFS, the TP-BFS engine does not know which nodes are the hubs and should be masked from the graph. Therefore, when the neighbors of the first node arrive, the engine checks whether it is a hub node (line 11 of Algorithm 4). In addition, the engine must also check whether it has itself visited this node during the execution of the current task (line 12 of Algorithm 4). When either happens, the engine needs to skip the node and work on the next neighbor. Otherwise, the engine would have appended the node to the local visited list and increment the $counter$ (line 16 of Algorithm 4).

To accelerate BFS and reduce off-chip accesses to adjacency data, redundant search must be avoided. To achieve this, the Island Locator keeps a record of the IDs of nodes visited by all TP-BFS engines during a certain round in a global visit list, v_{global}. When a BFS engine reaches a node labeled as visited in the global, but not in the local visited list, it knows that this region has been searched previously by other engines. The BFS engine then drops this task and waits for the next (line 20 of Algorithm 4 and Figure 5(A)).

As shown in the Algorithm 4 (line 14), the TP-BFS engine checks that a node is not on the global visit list before adding it to the local

Algorithm 4 Simplified TP-BFS: fetch nodes from the task queue and use them as starting points in island detection

1: **Inputs:** $task$: container of nodes to be chosen as potential starting points for BFS; TH: threshold for hub nodes' degree; c_{max}: the max number of node in an island; $l_{islands}$: list of islands' nodes and hubs info; $P2$: parallel factor
2: **Outputs:** None (modify $l_{islands}$ in place)
3: $v_{global} = \{\}$
4: **while** $|task| > 0$ **do**
5: **for** $p = 0$ **to** $P2$ **in parallel do** {# distributed across $P2$ engines}
6: Pop $\{hub_o, a_o\}$ from $task$ {# Pop the hub and one of its neighbors}
7: If a_o is not a hub: $v_{local}=\{a_o\}$; $h_{local}=\{hub_o\}$; $query$=0; $count$=1;
8: **while** $query \neq count$ **do** {# if there exist unexplored nodes}
9: $node_o = v_{local}[query]$
10: **for** $n \in node_o.neighbors$ **do**
11: **if** $n.degree < TH$ **then** {# hub node or not}
12: **if** $n \in v_{local}$ **then** {# n locally explored by engine p}
13: Skip neighbor n
14: **else if** $n \notin v_{global}$ **then** {# not explored by other engines}
15: $count += 1$
16: Append n to v_{local} and v_{global}
if exceeding the max number of nodes in an island
17: **if** $|v_{local}| > c_{max}$ break **while(line 8)**
18: **else** {# already explored by other engines}
19: remove v_{local} from v_{global}
20: break **while(line 8)**
21: **else** {# else it's a hub node}
22: Add n to h_{local}
23: $query += 1$
24: Append (v_{local}, h_{local}) to $l_{islands}$

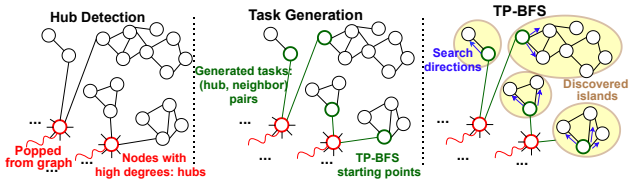

Figure 4: Toy example of the island locator algorithm.

visited list and incrementing the visited node counter. If the counter is over the threshold (expected maximum island size), the engine drops the task and waits for a new one ((line 17) and Figure 5 (B)).

When all neighbors of the initial node have been scanned, the engine checks whether the query pointer value equals the counter value. This indicates that all nodes have been searched, that TP-BFS is done without reaching the island size threshold, and that an island is found. If this happens, the engine sends the connection information of this island to the Island Consumer and requests a new task from the Task Generator (see Figure 5 (C)). Otherwise, it accesses the adjacency list of the node pointed to by the query pointer and explores all its neighbors. This process continues until one of the three task-break conditions is triggered.

3.2.2 Architecture. Figure 6 shows the architecture support of Island Locator. The blue part in Algorithm 1 is realized as the Hub Detector. The degree information of the nodes are distributively stored in the Node Degree Buffers which are implemented with

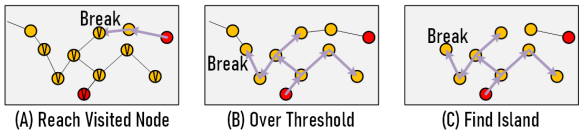

Figure 5: Task break conditions of TP-BFS engine.

loop-back FIFOs. The number of FIFOs determines the parallelism of the hub detection (P1 in Algorithm 1). The Island Node Filters (IFs) check whether the nodes popped out of the FIFOs are among the island nodes in the previous round (line 6 in Algorithm 2) by checking Island Node Table (PR-INT). If yes, these nodes are discarded; if no, the comparators check whether the nodes are hubs (line 8 in Algorithm 2). The nodes recognized as hubs are sent to a hub buffer which is also implemented with multi-bank FIFOs; while the remaining nodes are sent back to the Node Degree Buffers for the next round detection.

The TP-BFS Task Generator realizes the Algorithm 3. At each cycle, one hub node is popped out of the hub buffer and Task Generator accesses its adjacency list from global memory. The node IDs of the neighbors in the list received by Task Generator and the IDs of their hubs form task tuples which are cached in TP-BFS Task Queues. These tasks are further assigned to idle TP-BFS engines.

The simplified architecture of TP-BFS is shown in Figure 6(B). We implement the orange block of Algorithm 1 as a three-stage Finite State Machine (FSM). At Stage 0, TP-BFS is ideal and sends requests of new tasks to the Task Generator. When the new task arrives, TP-BFS engine moves to Stage 2 from Stage 0 with Query Pointer pointing to row 0 of the Local Visited Table (LVT) and the Island Node Counter as 1. In general, at Stage 2, the engine first checks whether the query pointer value and the Island Node Counter are the same. If yes, the engine finds an island, forwards the data stored at the output terminal to the Island Consumer, and records island nodes in CR-INT. Otherwise, the engine accesses the adjacency list of the node pointed by the Query Pointer from global memory, adds one to the Query Pointer, cache the connection information in the island bitmap buffer at output terminal and then moves to Stage 1. At Stage 1, the TP-BFS engine tries to find un-visited nodes from the newly arrived adjacency data by scanning them each per cycle. If the node being scanned is not a hub and also not visited, the Island Node Counter increases by one. If the counter value overpasses the maximum number of nodes for an island, the engine is reset to Stage 0. Otherwise, the engine keeps scanning the list and at the end moves to state 2.

3.3 Island Consumer

Island Consumer follows Island Locator and conducts the combination and aggregation of islands and their hubs. Before introducing the algorithm and architecture adopted in Island Locator, we first discuss where the redundant calculation of aggregation comes from.

As islands contain nodes with strong internal connections, it is highly likely that multiple nodes have more than one shared neighbors in which case the aggregation result of these shared neighbors can be reused multiple times with one-time calculation. Figure 7 uses the graph structure and adjacency matrix of a typical island as a motivational example. As mentioned in Section 3.1, the adjacency matrix of an island includes all connections between island nodes

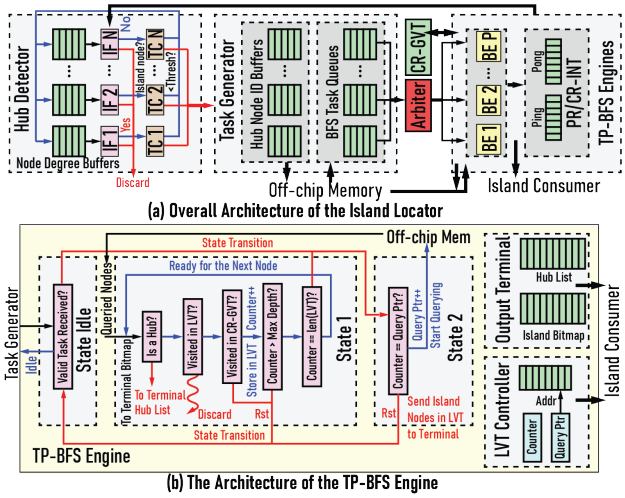

Figure 6: Simplified architecture of Island Locator.

and the hubs connected to them. During island processing, all these connections are evaluated. The example graph has seven island nodes from 'a' to 'g' and one hub 'H'.

As shown in Figure 7(A1), nodes d,e,f, and g are the shared neighbors of nodes b and c; as an undirected graph, nodes b and c are also the shared neighbors of nodes d, e, f, and g. For the aggregation of nodes b and c, the feature vectors of nodes d, e, f, and g need to be accumulated twice; to process nodes d, e, g, and f, the feature vectors of nodes b and c need to be accumulated four times. If the feature vector length is L, these accumulations take $16 \times L$ operations. However, if we can pre-calculate the accumulation results and reuse them, then only $10 \times L$ operations are needed. Figure 7(A2) shows how the pre-calculated results are reused during aggregation. We add two virtual nodes whose feature vectors are the accumulation results d,e,f,g and c,b in the graph. We connect them to the real nodes according to the accumulation requirements. During the aggregation phase, the pre-aggregated feature vectors are forwarded directly to the target nodes, so that the accumulation from shared neighbors is only done once. Note that in the original graph without island-based locality enhancement, these nodes are highly scattered, as is the processing of their aggregation operations. It is therefore prohibitive to find and prune these repeated operations from shared neighbors at runtime. Through I-GCN, the nodes with strong interconnects are all clustered, which makes redundancy removal feasible.

3.3.1 Algorithm. We first introduce the calculation methodology of the Island Consumer. Once the information of an island is forwarded to the Island Consumer, it assigns that information to a PE which is waiting for new calculation tasks. The information includes island node IDs, hub node ID, the local adjacency bitmap, the round IDs, and etc as shown in Figure 7. The PE performs first combination and then aggregation reusing the same MAC units.

The PE starts the combination of all hub and island nodes by first accessing their input feature vectors from global memory and then conducting PULL-based combination. Different from conventional combination, the Island Consumer conducts pre-aggregation at the completion of the combination of every k node. Specifically,

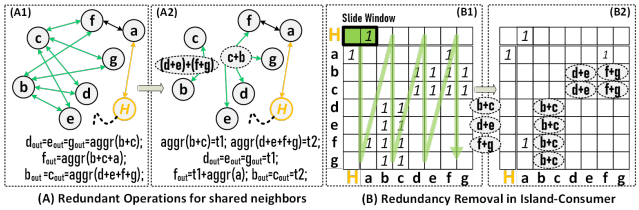

Figure 7: Redundancy removal of a typical island.

after the combination results of k nodes have been calculated, we sum them up and use the results in aggregation and so skip the redundant operations. k can be customized.

When the combination results of all nodes in an island and their pre-aggregated results are ready, the PE starts aggregation by scanning the local adjacency bitmap. Figure 7(B) shows how the Island Consumer uses these results in aggregation with redundancy removal. The scan starts from the top-left corner of the bitmap and slides towards the bottom-right following the green trace. The size of the scan window is $1 \times k$ where k is the number of nodes whose feature vectors are pre-aggregated during the combination phase. If the number of non-zeros covered by the sliding window is less than $k/2$, it is more efficient to accumulate the feature vectors of nodes that are connected (column id of non-zeros under the sliding window) to the node being scanned (row id of the sliding window); otherwise, it is more efficient to subtract the feature vectors of nodes that are not connected (column id of zeros under the sliding window) from the pre-aggregation results. The Island Consumer can automatically pick the one that demands the fewest operations.

Figure 7(B) gives an example. For clarity, k is set to 2. The scan starts once the combination of node-H and node-a finishes and their results are accumulated in pre-aggregation. For each scan, if both bits are non-zeros, i.e. the nodes under-scan are the common neighbors of node-H and node-a, instead of redoing the accumulation, Island Consumer will directly use the pre-aggregation result, saving one vector addition operation. After the first two columns are scanned, Island Consumer proceeds to columns b and c with the combination and pre-aggregation results of node-b and node-c. After the entire bitmap is scanned, both GraphCONV's aggregation and combination of the island have been completed.

3.3.2 Architecture. The architecture of the Island Consumer is illustrated in Figure 8. The island information sent from the Island Locator is received by the Island Collector and stored in the distributed memory of the Island Evaluation Task if the incoming island is not redundant. The major information of each island task includes: numbers of hubs and island nodes, hub node IDs, island node IDs, adjacency bitmap, and the round ID. The arbiters in island collector prefetch evaluation tasks every clock cycle and forward them to the idle PEs. Once a PE receives an evaluation task, it performs the aggregation and combination. At the end of evaluation of each island, the PE produces complete output features of all island nodes and partial results of the output features of hubs. The complete final outputs of island nodes are stored back to the global memory, while the incomplete results of hubs are sent to the corresponding bank of a HUB Partial Result Cache through the ring-based reduction network to update the corresponding partial sums calculated previously. To obtain complete aggregation results

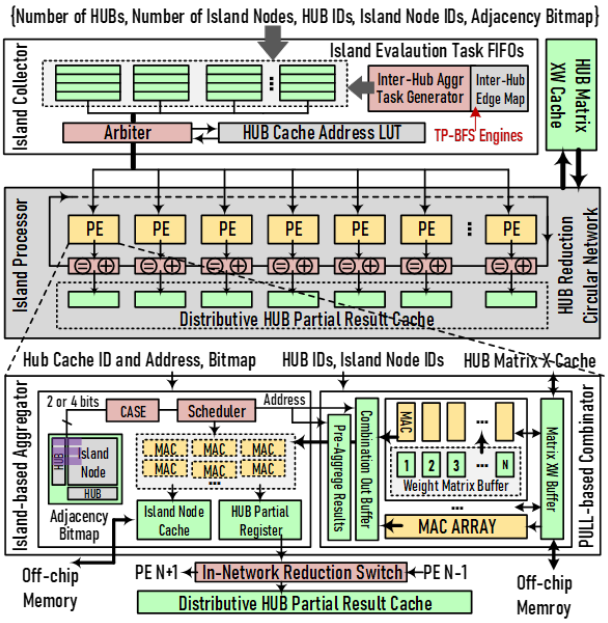

Figure 8: Simplified architecture of Island Consumer.

of hub nodes, it is necessary to evaluate not only the islands but also the inter-hub connections, which are not included in islands. To perform the aggregation of inter-hub connections, Island Collector maintains an inter-hub edge map based on the information provided by TP-BFS engines during island locating as mentioned in Section 3.2.1, generates inter-hub aggregation tasks based on the edge map, and inserts the tasks into the evaluation task queues. More details about the creation of inter-hub edge map and the generation of inter-hub tasks are omitted due to the space limit. Once all islands and inter-hub tasks are evaluated, the complete hubs' results are obtained.

As shown in Figure 8, all PEs are connected in a ring network. The partial results of hub nodes are distributively stored in the multi-bank HUB Partial Result Cache (DHUB-PRC). Each bank of DHUB-PRC is attached to one PE. Note that at the first appearance of each hub, the Island Collector will map it to an unused row in a certain bank. The bank ID and row address are attached to the hub before its island evaluation task is assigned to a PE. This bank ID and row address will be fixed and reused when this hub appears again in the future islands. At the end of each island processing, the PE will check the hubs' bank IDs. If the bank ID matches its PE ID, the partial results will be used to update the partial sum stored locally, otherwise it will be forwarded to the corresponding banks through the ring network.

To accelerate the partial sum update of hubs and reduce communication, the ring network is equipped with the support of in-network reduction. Particularly, the switch at each entry of the ring network will check the hub IDs of the partial results sent from local PE and from its left neighbor. If both are valid and the same, they are reduced at the entry and the new result will be sent to the right neighbor at the next cycle. This in-network reduction design is widely used in smart-NIC architectures, so we do not elaborate it due to page limit.

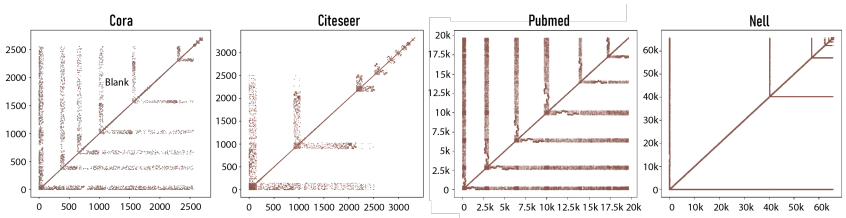

Figure 9: Islandization effect on Cora, Citeseer, PubMed, and NELL. Space between L-shapes is totally blank.

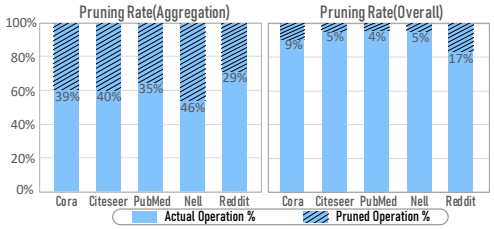

Figure 10: Pruning rates with redundancy removal.

The simplified architecture of each PE is illustrated at the bottom of Figure 8. The right part is the module for PULL-based combination. As mentioned in Section 3.3.1, this module accesses feature data of island nodes based on their IDs from off-chip. These data are accessed only once as all connections of these island nodes will be processed during the execution of this island task. Weight matrix is distributively stored on-chip in Weight Matrix Buffers of PEs if possible. During calculation, non-zeros of the same node will be sequentially broadcast to the same row of MAC units and are multiplied with the corresponding row of weight matrix. The partial combination results of the same row are locally accumulated. Once a certain node is completely calculated, the row of MAC units starts to work on the next node if applicable. If the node is a hub, the resulting combined features will be stored in HUB Matrix XW Cache for the reuse in future island and inter-hub tasks. Once the combination of all nodes of are done, the right module of this PE will be deactivated and the left one will be activated to start aggregation. The left module is designed strictly based on the calculation method introduced in Section 3.3.1. The purple blocks in the figure are the scan windows. To avoid pipeline bubbles in the aggregation module, the scans with all zeros need to be skipped. To achieve this, we scan multiple rows in parallel at each cycle and only forward the scans with non-zeros to FSMs, i.e. CASE and Scheduler modules in the figure, which access the required pre-aggregation and/or combination output results from the combination module and assign the aggregation task to idle MAC units.

4 EVALUATION

4.1 Experiment Setup

We evaluate I-GCN's latency, energy efficiency, off-chip bandwidth requirement, and hardware resource utilization with a Stratix 10 SX FPGA. We evaluate I-GCN on three different models – GCN, GraphSage, and GIN – using five datasets commonly used in GCN acceleration research [14, 30, 48]. These include Cora (CR), Citeseer (CS), Pubmed (PM), Nell (NE), and Reddit (RD). With respect to network structures, existing systems use various configurations: GCNs and GraphSage have two layers, while GIN has three layers. For GCNs and GraphSage, EnGN and AWB-GCN use the configurations reported in the original algorithm papers [25], while HyGCN uses its own configuration of 128 hidden channels for all datasets. Here we label GCN and GraphSage with original configurations, "GCN-algo" and "GS-algo" and label the ones used in HyGCN, "GCN-Hy" and "GS-Hy". For GIN, HyGCN is the only work that uses it in evaluation. We evaluate all these models and compare with the corresponding existing work.

To better demonstrate the efficiency of the proposed on-the-fly algorithm-architecture co-design for islandization, we compare I-GCN with 6 existing lightweight graph reordering algorithms (Section 4.5). These algorithms are realized on an Intel Xeon Gold 6226R CPU with 64 threads. For cross-platform comparison, we compare I-GCN with prior art GCN accelerators (HyGCN [48], AWB-GCN [14]), prior art SpMM accelerator (SIGMA [38]), NVIDIA V100 GPU, RTX8000 GPU, Intel E5-2683-V3 CPU, and Intel E5-2680-V3 CPU. The CPU and GPU results are based on PyTorch Geometric (PyG) [13] and Deep Graph Library (DGL) [41].

ALM(s)	Module	Number	Percentage
Island Locator	TP-BFS Engines	197K	24%
	Task Generator	21K	3%
	Hub Detector	61K	7%
Island Consumer	Island Processor	57K	7%
	Island Collector	51K	6%
	MAC Units	*450K*	*53%*
Overall		837K	100%

Table 2: Hardware Consumption of I-GCN.

4.2 Islandization Effect

We first evaluate the efficiency of the islandization algorithm. Figure 9 shows the effect and versatility of the proposed islandization on the adjacency matrices of real-world graphs with various statistics including size, sparsity and non-zero distribution. Among them, NELL is believed to be the most difficult to process due to its extremely high sparsity and imbalanced distribution[14]. As shown in Figure 9, for all these datasets, our islandization method is able to optimally cluster all non-zeros to the anti-diagonals and L-shaped clusters within several rounds. The islandization effect on NELL looks even more significant than the other datasets, since its adjacency matrix is the sparsest. To summarize, the proposed islandization algorithm is shown to be effective for graphs under various statistics.

4.3 Island-based Redundancy Removal

Here we evaluate the effects of shared-neighbor-aware redundancy removal in the Island Consumer. Figure 10 shows the operation pruning rates during aggregation phase. The Island-based Aggregator is able to skip over 38% of aggregation operations. These operations are all redundant and are for the aggregation among shared neighbors. The removal of these operations is lossless. Note that in combination-first calculation, aggregation phase takes 23% operations on average. Therefore, 9% of operations of the entire

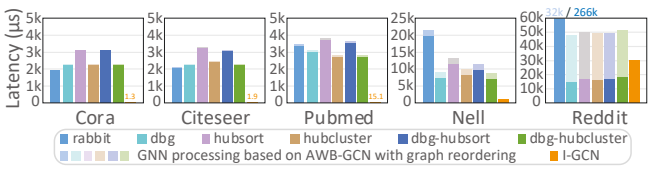

Figure 11: Latency of I-GCN vs AWB-GCN + light weight reordering algorithms.

processing can be eliminated without accuracy loss. Meanwhile, the theoretical latency is lowered by 9%.

4.4 Hardware Consumption and Scalability

Table 2 lists the hardware resource usages of the I-GCN with 4K MAC units and 64 TP-BFS Engines. In order to show comparable breakdowns with ASIC implementation, we normalize the usage of LUTs and Flip-Flops to the number of Adaptive Logic Modules (ALMs) which is the basic component in Intel FPGAs. The Island Locator only accounts for 34% of the entire accelerator. The 4K Floating-Point MAC units account for another 53%. Note that in practical FPGA implementations, the MAC units are normally instantiated with DSP slices. Here we normalize the usage of DSP slides to ALMs for a clearer area breakdown.

4.5 Comparison with Lightweight Reordering

To evaluate the benefits of I-GCN over lightweight graph reordering algorithms, we compare I-GCN with 6 baselines. These baselines use 6 traditional lightweight graph reordering algorithms [3, 5, 12, 53] for graph preprocessing to enhance locality and then use AWB-GCN [14], which is a prior art GNN accelerator, to process the reordered graphs. We use these open-source graph ordering codes[12] and run them using a high-end Intel Xeon Gold 6226R CPU with 64 threads enabled. There are two findings. First, I-GCN is much faster than the lightweight graph reordering approaches. As shown in Figure 11, the reordering latency alone is already higher than the entire I-GCN end-to-end inference latency (more than 100× for Cora, Citeseer, and Pubmed). And second, I-GCN generates better non-zero clustering. As shown in Figure 12, I-GCN's islandization process is able to push all the non-zeros into the L-shaped regions and the anti-diagonal, leaving the remaining area empty. In contrast, the graph reordering methods leave many outlying non-zeros, which introduces significant overhead for their special handling.

4.6 Cross-platform Comparison

4.6.1 Off-chip Bandwidth Requirement. Figure 13(A) compares the normalized numbers of off-chip data accesses of I-GCN with AWB-GCN, HyGCN, and PyG-CPU (Intel Xeon E5-2680-V3) using both GCN-Algo and GCN-Hy. Note that we count the numbers of off-chip accesses assuming that the adjacency matrix and input feature matrix are all stored off-chip at the start of processing. In practice, in the case that the on-chip memory is not fully occupied, these matrices can be partially or even completely stored on-chip to reduce the off-chip bandwidth requirements.

4.6.2 Latency. Figure 13(B) compares the end-to-end inference latency of I-GCN with SOTA GNN accelerators (AWB-GCN, HyGCN),

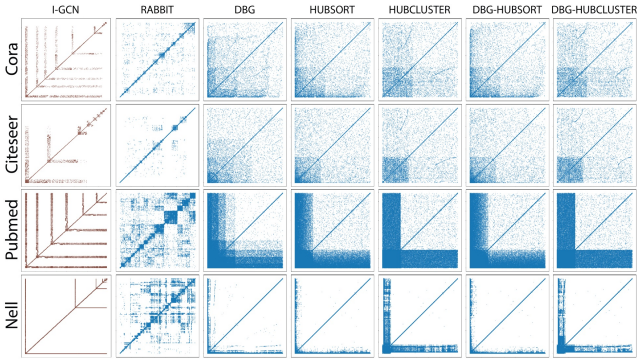

Figure 12: Comparison of non-zero clustering effects

SOTA SpMM accelerator (SIGMA), PyG-based CPU and GPU, and DGL-based CPU and GPUs. Results show I-GCN provides speedups of 9568× over PyG-based E5-2680-v3 CPU, 1243× over DGL-based E5-2683-v3 CPU, on average 368× over PyG-based GPUs (RTX8000 and V100), 453× over DGL-based V100, 16× over SIGMA, and on average 5.7× over GNN accelerators (HyGCN and AWB-GCN). Table 3 lists the absolute results of I-GCN and AWB-GCN. The speedups of I-GCN over AWB-GCN on Reddit is lower than other datasets, as Reddit graph has less significant component structures.

Fairness of evaluation: HyGCN is an ASIC design which uses 4608 fixed-point MAC units running at 1GHz; AWB-GCN is an FPGA design which uses 4096 floating-point MAC units (running at 330MHz). To provide a fair comparison, the I-GCNs used for evaluation are also equipped with 4096 floating-point MAC units running at 330MHz and consume less ALM resources of the same FPGA used by AWB-GCN.

5 RELATED WORK

Researchers have designed dedicated hardware architecture to accelerate GCNs [1, 4, 8, 14, 24, 29, 30, 48, 51, 55]. In [4], Auten et al., present the first GNN hardware accelerator. By designing four specialized modules – for graph traversals, dense matrix operations, data scheduling, and graph aggregations, respectively – the proposed accelerator provides high performance in tackling irregular data movement and intensive computation for GNN inference. *HyGCN* [48] is another of the earliest GNN accelerators. With the observation that GCNs are composed of two phases with different computation patterns, HyGCN introduces a hybrid architecture with dedicated modules for aggregation and combination, respectively. *AWB-GCN* [14] is another early study of GCN acceleration. It observes that the power-law distribution of the non-zeros in the adjacency matrix results in workload imbalance issues. To solve this problem, the authors propose a workload autotuning technique. *EnGN* [30] uses an unified architecture to accelerate GNNs and adopts a ring-based network to perform aggregation. The results produced by PEs are sent to the network where they are aggregated. Researchers have also designed hardware for training. *Rubik* [8] proposes an offline graph reordering method to improve data locality. *GraphACT* [51] uses heterogeneous platforms with CPUs and FPGAs and uses pre-processing to find and skip redundant operations among two-node shared neighbors. *G-CoS*[55] is the first GNN co-search framework for network structure and accelerator

	I-GCN						AWB-GCN					
	Device: Intel Stratix 10 SX; Frequency: 330MHz; Num of MACs: 4096											
		Cora	Citeseer	Pubmed	Nell	Reddit		Cora	Citeseer	Pubmed	Nell	Reddit
GCN_algo	***Latency***	1.3	1.9	15.1	5.8E2	3.0E4	***Latency***	2.3	4.0	30	1.6E3	3.2E4
	EE	7.1E6	3.7E6	5.3E5	1.3E4	3.5E2	***EE***	3.1E6	1.9E6	2.5E5	4.1E3	2.1E2
GCN_Hy	***Latency***	8.2	12.9	1.1E2	1.1E3	4.6E4	***Latency***	17	29	2.3E2	3.3E3	5.0E4
	EE	9.6E5	6.0E5	8.1E4	7.5E3	2.2E2	***EE***	4.4E5	2.7E5	3.2E4	2.3E3	1.5E2

Table 3: I-GCN's absolute results of Latency in μs and Energy Efficiency (EE) in Graph/kJ.

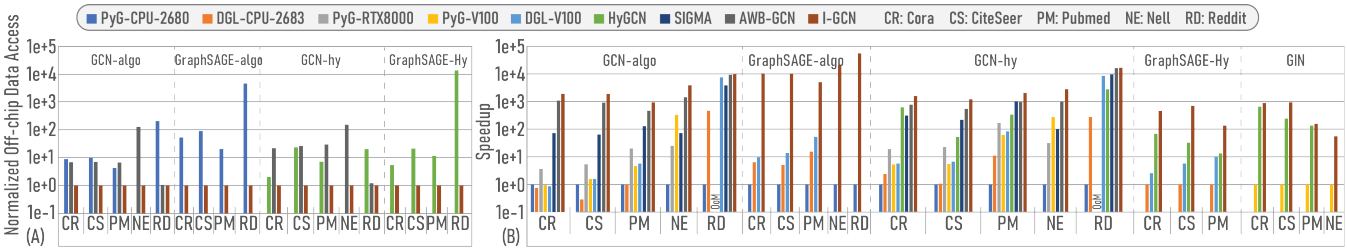

Figure 13: Cross-platform comparison of (A) Normalized off-chip data access and (B) Speedup (end-to-end latency).

architecture. G-CoS can automatically search for the matched GNN structures and accelerators to maximize both task accuracy and acceleration efficiency.

In differentiation from all prior work, the proposed I-GCN is designed to solve the data locality problem fundamentally. As I-GCN reorders graphs using hardware-only solutions, it is compatible with both static and dynamic graphs and both inductive and transductive GNN models. Furthermore, I-GCN finds and skips redundant operations among arbitrary numbers of shared neighbors at runtime.

In graph processing, various reordering algorithms [3, 5, 9, 12, 22, 28, 43, 53, 54] have been proposed for enhancing data locality. The evaluations of six traditional lightweight graph reordering algorithms [3, 5, 12, 53] in Section 4.5 demonstrates the high overheads of graph reordering, even for the lightweight ones, which is prohibitive for many real-time GNN inference tasks. There are also many other sophisticated graph reordering algorithms. SlashBurn [31] is one of them. Similarly to the proposed islandization algorithm, it finds components from high-degree nodes. SlashBurn is able to cluster non-zeros even better than islandization. However, it requires expensive and frequent node degree sorting, graph reconstruction, component size sorting, and node degree updating. Furthermore, SlashBurn is not designed to be parallelized. These make SlashBurn hardware-unfriendly and unsuited for GNN acceleration which normally poses strict constraints on latency.

Other architectural and software optimizations have been proposed that improve graph processing efficiency through cache-guided scheduling [34, 35]. Among them, HATS [34] appears to be the first hardware work that leverages the community structure of graphs without preprocessing. The main goal of HATS, which is tightly integrated with hierarchical cache systems, is to enhance the efficiency of the cache hierarchy for graph processing. In contrast, the islandization in our loosely-coupled I-GCN accelerator aims at clustering non-zeros through community identification for the purpose of extremely fast GNN inference (μs-level). Also, I-GCN and HATS define components in different ways. HATS detects very coarse-grained and large components by Bounded Depth-First Scheduling (BDFS), while I-GCN locates smaller and more fine-grained components through hub nodes.

Another related topic is hardware acceleration for SpMM [20, 37–40]. Prior art systems include MatRaptor [39], Extensor [20], SIGMA [38], and Tensaurus [40]. Although the major kernel of GNN processing is SpMM (see Equation 1), a high-performance GNN accelerator should be able to fully leverage unique graph features. For example, real-world graphs are extremely sparse, contain communities, and follow the power-law distribution. I-GCN, as a graph-specific architecture, can effectively detect communities from real-world large graphs, and process them more efficiently by avoiding repeated computation for common neighbors. In contrast, SpMM accelerators need to handle all different kinds of sparse matrices. They may behave better for general sparse matrices, but not in those likely to be processed by GCNs.

6 CONCLUSION

This paper proposes a novel hardware accelerator for GCN inference, I-GCN, which significantly improves data locality and reduces unnecessary computation through a new hardware runtime algorithm — islandization. Islandization finds clusters of nodes with strong internal but weak external connections which yields two major benefits: (1) by processing islands, data can be better reused on-chip which significantly relieves the off-chip bandwidth pressure; (2) there is less redundant computation as aggregation for shared neighbors in an island can be reused. Experimental results show that I-GCN provides speedups over CPUs, GPUs, prior art GCN accelerators of 5549×, 403×, and 5.7×, respectively.

ACKNOWLEDGMENTS

This work was supported by the Compute-Flow-Architecture (CFA) project under PNNL's Data-Model-Convergence (DMC) LDRD Initiative. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under Contract DE-AC05-76RL01830.

REFERENCES

[1] Sergi Abadal, Akshay Jain, Robert Guirado, Jorge López-Alonso, and Eduard Alarcón. 2020. Computing Graph Neural Networks: A Survey from Algorithms to Accelerators. arXiv:2010.00130 [cs.LG]

[2] A. Abou-Rjeili and G. Karypis. 2006. Multilevel algorithms for partitioning power-law graphs. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium.* 10 pp.–. doi: 10.1109/IPDPS.2006.1639360.

[3] Junya Arai, Hiroaki Shiokawa, Takeshi Yamamuro, Makoto Onizuka, and Sotetsu Iwamura. 2016. Rabbit Order: Just-in-Time Parallel Reordering for Fast Graph Analysis. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* 22–31. https://doi.org/10.1109/IPDPS.2016.110

[4] Adam Auten, Matthew Tomei, and Rakesh Kumar. 2020. Hardware acceleration of graph neural networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC).* IEEE, 1–6.

[5] Vignesh Balaji and Brandon Lucia. 2018. When is Graph Reordering an Optimization? Studying the Effect of Lightweight Graph Reordering Across Applications and Input Graphs. In *2018 IEEE International Symposium on Workload Characterization (IISWC).* 203–214. https://doi.org/10.1109/IISWC.2018.8573478

[6] Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. 2014. Spectral Networks and Locally Connected Networks on Graphs. In *the 2nd International Conference on Learning Representations.*

[7] Jie Chen, Tengfei Ma, and Cao Xiao. 2018. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247* (2018).

[8] Xiaobing Chen, Yuke Wang, Xinfeng Xie, Xing Hu, Abanti Basak, Ling Liang, Mingyu Yan, Lei Deng, Yufei Ding, Zidong Du, et al. 2020. Rubik: A Hierarchical Architecture for Efficient Graph Learning. *arXiv preprint arXiv:2009.12495* (2020).

[9] YuAng Chen and Yeh-Ching Chung. 2021. Corder: cache-aware reordering for optimizing graph analytics. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* 472–473.

[10] Connor W. Coley, Wengong Jin, Luke Rogers, Timothy F. Jamison, Tommi S. Jaakkola, William H. Green, Regina Barzilay, and Klavs F. Jensen. 2019. A graph-convolutional neural network model for the prediction of chemical reactivity. *Chemical Science* 10 (2019), 370–377. Issue 2. doi: 10.1039/C8SC04228D.

[11] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Proceedings of the 30th International Conference on Neural Information Processing Systems.* 3844–3852. doi: 10.5555/3157382.3157527.

[12] Priyank Faldu, Jeff Diamond, and Boris Grot. 2019. A Closer Look at Lightweight Graph Reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC).* 1–13. https://doi.org/10.1109/IISWC47752.2019.9041948

[13] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *Computing Research Repository (CoRR) in arXiv* abs/1903.02428 (2019).

[14] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt, and M. C. Herbordt. 2020. AWB-GCN: a Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture.* 922–936. doi: 10.1109/MICRO50266.2020.00079.

[15] T. Geng, T. Wang, C. Wu, Y. Li, C. Yang, W. Wu, A. Li, and M.C. Herbordt. 2021. O3BNN-R: an Out-Of-Order Architecture for High-Performance and Regularized BNN Inference. *IEEE Transactions on Parallel and Distributed Systems* 32, 1 (2021), 199–213. doi: 10.1109/TPDS.2020.3013637.

[16] T. Geng, T. Wang, C. Wu, C. Yang, W. Wu, A. Li, and M.C. Herbordt. 2019. O3BNN: an Out-Of-Order Architecture for High-Performance Binarized Neural Network Inference with Fine-Grained Pruning. *ACM International Conference on Supercomputing* 2160 (2019), 461–472. doi: 10.1145/ 3330345. 3330386.

[17] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation.* USENIX Association, 17–30. doi: 10.5555/2387880.2387883.

[18] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems.* 1024–1034. doi:10.5555/3294771.3294869.

[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture.* IEEE, 243–254. doi: 10.1109/ISCA.2016.30.

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

[21] Avinash Karanth Jiajun Li, Ahmed Louri and Razvan Bunescu. 2021. GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks. In *2021 IEEE International Symposium on High-Performance Computer Architecture.*

[22] Konstantinos I Karantasis, Andrew Lenharth, Donald Nguyen, Mara J Garzaran, and Keshav Pingali. 2014. Parallelization of reordering algorithms for bandwidth and wavefront reduction. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE, 921–932.

[23] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2017. A novel zero weight/activation-aware hardware architecture of convolutional neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017.* IEEE, 1462–1467.

[24] Kevin Kiningham, Christopher Re, and Philip Levis. 2020. GRIP: A Graph Neural Network AcceleratorArchitecture. *arXiv preprint arXiv:2007.13828* (2020).

[25] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *Computing Research Repository (CoRR) in arXiv* abs/1609.02907 (2016).

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems.* 1097–1105.

[27] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical computer science* 407, 1-3 (2008), 458–473.

[28] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H Noh, and Jiwon Seo. 2019. Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19).* 459–474.

[29] Shengwen Liang, Cheng Liu, Ying Wang, Huawei Li, and Xiaowei Li. 2020. DeepBurning-GL: an automated framework for generating graph neural network accelerators. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD).* IEEE, 1–9.

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

[31] Yongsub Lim, U Kang, and Christos Faloutsos. 2014. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering* 26, 12 (2014), 3077–3089.

[32] Yuxiao Liu, Ning Zhang, Dan Wu, Audun Botterud, Rui Yao, and Chongqing Kang. 2020. Guiding Cascading Failure Search with Interpretable Graph Convolutional Network. *Computing Research Repository (CoRR) in arXiv* abs/2001.11553 (2020).

[33] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *Eleventh annual conference of the international speech communication association.*

[34] Anurag Mukkara, Nathan Beckmann, Maleen Abeydeera, Xiaosong Ma, and Daniel Sanchez. 2018. Exploiting Locality in Graph Analytics through Hardware-Accelerated Traversal Scheduling. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 1–14. https://doi.org/10.1109/MICRO.2018.00010

[35] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. 2017. Cache-Guided Scheduling: Exploiting caches to maximize locality in graph processing. *AGP'17* (2017).

[36] Huy-Trung Nguyen, Quoc-Dung Ngo, and Van-Hoang Le. 2018. IoT botnet detection approach based on PSI graph and DGCNN classifier. In *2018 IEEE International Conference on Information Communication and Signal Processing (ICICSP).* IEEE, 118–122.

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

[38] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 58–70.

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

[40] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 689–702.

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

[42] T. Wang, T. Geng, A. Li, X. Jin, and M. Herbordt. 2020. FPDeep: Scalable Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters. *IEEE Trans. Comput.* 69, 08 (2020), 1143–1158. doi: 10.1109/TC.2020.3000118.

[43] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*. 1813–1828.

[44] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* (2020).

[45] Cong Xie, Ling Yan, Wu-Jun Li, and Zhihua Zhang. 2014. Distributed power-law graph computing: Theoretical and empirical analysis. In *Advances in neural information processing systems*. 1673–1681.

[46] Tian Xie and Jeffrey C Grossman. 2018. Crystal graph convolutional neural networks for an accurate and interpretable prediction of material properties. *Physical review letters* 120, 14 (2018), 145301.

[47] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[48] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. 2020. HyGCN: A GCN Accelerator with Hybrid Architecture. *Computing Research Repository (CoRR) in arXiv* abs/2001.02514 (2020).

[49] Hongxia Yang. 2019. Aligraph: A comprehensive graph neural network platform. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, 3165–3166.

[50] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. 2019. Graph Transformer Networks. In *Advances in Neural Information Processing Systems*. 11960–11970.

[51] Hanqing Zeng and Viktor Prasanna. 2020. Graphact: Accelerating gcn training on cpu-fpga heterogeneous platforms. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 255–265.

[52] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 20.

[53] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. 293–302. https://doi.org/10.1109/BigData.2017.8257937

[54] Yunming Zhang, Vladimir Kiriansky, Charith Mendis, Saman Amarasinghe, and Matei Zaharia. 2017. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 293–302.

[55] Yongan Zhang, Haoran You, Yonggan Fu, Tong Geng, Ang Li, and Yingyan Lin. 2021. G-CoS: GNN-Accelerator Co-Search Towards Both Better Accuracy and Efficiency. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2021)*.

[56] Marinka Zitnik, Monica Agrawal, and Jure Leskovec. 2018. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics* 34, 13 (2018), i457–i466.