CDA-GNN: A Chain-driven Accelerator for Efficient Asynchronous Graph Neural Network

Hui Yu[†], Yu Zhang[†], Ligang He[§], Donghao He[†], Qikun Li[†], Jin Zhao[†], Xiaofei Liao[†], Hai Jin[†], Lin Gu[†] and Haikun Liu[†]

[†]National Engineering Research Center for Big Data Technology and System
Service Computing Technology and System Lab, Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[§] Department of Computer Science, University of Warwick, United Kingdom
{huiy, zhyu}@hust.edu.cn ligang.he@warwick.ac.uk {hdh, lqk2021, zjin, hjin, xfliao, lingu, hkliu}@hust.edu.cn

Abstract

Asynchronous Graph Neural Network (AGNN) has attracted much research attention because it enables faster convergence speed than the synchronous GNN. However, existing software/hardware solutions suffer from *redundant computation overhead* and *excessive off-chip communications* for AGNN due to irregular state propagations along the dependency chains between vertices. This paper proposes a chain-driven asynchronous accelerator, *CDA-GNN*, for efficient AGNN inference. Specifically, *CDA-GNN* proposes a chain-driven asynchronous execution approach into novel accelerator design to regularize the vertex state propagations for fewer redundant computations and off-chip communications and also designs a chain-aware data caching method to improve data locality for AGNN. We have implemented and evaluated *CDA-GNN* on a Xilinx Alveo U280 FPGA card. Compared with the cutting-edge software solutions (i.e., Dorylus and AMP) and hardware solutions (i.e., BlockGNN and FlowGNN), *CDA-GNN* improves the performance of AGNN inference by an average of 1,173x, 182.4x, 10.2x, and 7.9x and saves energy by 2,241x, 242.2x, 12.4x, and 8.9x, respectively.

ACM Reference Format:
Hui Yu, Yu Zhang, Ligang He, Donghao He, Qikun Li, Jin Zhao, Xiaofei Liao, Hai Jin, Lin Gu and Haikun Liu. 2024. CDA-GNN: A Chain-driven Accelerator for Efficient Asynchronous Graph Neural Network. 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/10.1145/3649329.3656240

Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper.
This paper is supported by National Key Research and Development Program of China (No. 2022YFB2404202), Key Research and Development Program of Hubei Province (No. 2023BAB078), Knowledge Innovation Program of Wuhan-Basi Research (No. 2022013301015177), and Huawei Technologies Co., Ltd (No. YBN2021035018A6).

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656240

1 Introduction

Graph Neural Networks (GNNs) have been used in a wide range of real-world applications [10]. The vertices in GNNs follow a neighborhood aggregation scheme, communicating with their neighboring vertices for multiple iterations via synchronous message passing. However, the recent studies [4, 9] show that they suffer from poor performance because of the frequent global synchronization between layers. Thus, a more efficient GNN model called *Asynchronous Graph Neural Network* (AGNN) [2], emerges to address this problem. In contrast to the traditional GNN model, the state (i.e., feature vector) of each vertex in AGNN is asynchronously updated by aggregating and transforming the states of its neighbors until the whole process converges. The gradients and embeddings of the AGNN model have also been proven with the convergence guarantee [2].

Although several software AGNN systems [2, 4, 7, 9] and hardware GNN accelerators [8, 15] have been proposed, they suffer from redundant computations and excessive off-chip communications because of the following two reasons. First, within each iteration of AGNN, the active vertices processed by different cores irregularly propagate the most recent states to the neighbors along the dependency chains[1] between these vertices in an asynchronous way. As a result, most vertices may use the stale states of their neighbors to conduct state updates, which are unnecessary. Second, AGNN also suffers from serious *irregular memory access* because the vertices' states are too large to be stored in the on-chip memory [12] and sparsely dispersed in the off-chip memory.

To address the issues mentioned above, this paper designs a chain-driven hardware accelerator *CDA-GNN* for efficient AGNN inference, which supports an effective chain-driven

[1]In AGNN inference, each vertex's state depends on its neighbor's state, because this vertex updates its state through gathering its direct neighbor's state along an edge. It means that the graph path between two vertices will incur a dependency chain between these two vertices' states along this path, because of the dependency between each vertex and its direct neighbor caused by the edge on this path.

asynchronous execution approach to ensure fewer vertex state updates and better data locality. Specifically, in AGNN inference, when the active vertices on a dependency chain are processed consecutively and asynchronously along this dependency chain during an iteration, the other vertices need fewer state updates in this iteration. Thus, *CDA-GNN* tracks the dependency chains between the active vertices on the fly and consecutively processes the active vertices on each tracked chain along their order on this chain. By such means, the data accessing and processing associated with many vertex state updates can be spared. Due to the real-world graphs' power-law property [5], most vertex state propagations occur only on a small set of dependency chains, which indicates that these chains are frequently-used ones. Thus, for better data locality, *CDA-GNN* specializes the memory subsystem to preferentially cache the states on the frequently-used dependency chains in the on-chip memory.

We have implemented and evaluated *CDA-GNN* on a Xilinx Alveo U280 FPGA card. The experimental results show that *CDA-GNN* outperforms the cutting-edge software AGNN solutions (i.e., Dorylus [9] and AMP [2]) running on Intel Xeon CPU and NVIDIA A100 GPU by, on average, 1,173x and 182.4x with 2,241x and 242.2x energy savings, respectively. Compared with the state-of-the-art GNN accelerators, i.e., BlockGNN [15] and FlowGNN [8], *CDA-GNN* achieves the speedup of 8.6x-12.7x and 5.8x-9.7x with 9.4x-21.3x and 6.5x-15.9x energy savings, respectively.

2 Background and Motivation

2.1 Background of AGNN

Existing AGNN models use the popular *Compress Sparse Row* (CSR) [14] format to store the graph and follow an asynchronous message passing scheme [2]. In general, the AGNN adopts the vertex-program abstraction into three phases [2]: *ApplyEdge*, *Aggregation*, and *ApplyVertex* to abstract the dataflow of asynchronous message passing. The *ApplyEdge* phase evaluates each edge's *message* using neural networks to transform the states of the source vertex and this edge, and the *Aggregation* phase asynchronously gathers the neighbors' states. In the end, the *ApplyVertex* produces a new state for the target vertex using neural networks by combing the previous state of the target vertex and the state gathered in the *Aggregation* phase. Figure 1 gives an example to show the execution of AGNN inference. Each vertex (e.g., v_3) scatters the edge's message (e.g., *Message A*) to its neighbor (e.g., v_2) (❶), and the neighbor (i.e., v_2) asynchronously gathers the received message (❷) and combines its previous state and the received message to produce a new state (❸).

AGNN can generally be classified into two types of execution ways [2]: the ones without *ApplyEdge* and the ones with *ApplyEdge*. The first type, e.g., asynchronous GCN [10], only consists of an *Aggregation* phase and an *ApplyVertex* phase. Compared with the first type, the second type, e.g., asynchronous GIN [11], involves an additional phase (i.e., the *ApplyEgde* phase) before the *Aggregation* phase. Note that

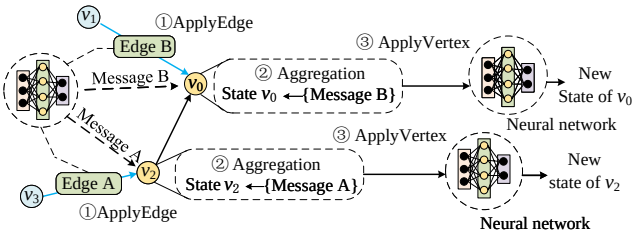
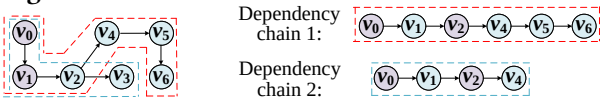


Figure 1. Illustration of the execution of AGNN inference



(a) An example graph (b) Two dependency chains of the example graph

Figure 2. An example how to divide the graph into dependency chain

the *ApplyEdge* and *ApplyVertex* phases can be represented by *Matrix-Vector Multiplication* (MVM).

2.2 Limitation of State-of-the-art Solutions

Many solutions [2, 4, 7, 9] have been proposed to handle the AGNN model on general-purpose processors. However, AGNN inference exhibits *redundant computations* and *excessive off-chip communications* due to its inherent irregular vertex state propagation. Figure 2 gives an example of the execution of AGNN, where the vertices v_0 and v_1 are assumed to be activated in an iteration. When processing v_0 and v_1 in this iteration, v_0 scatters its state to its neighbor v_1, while v_1 will scatter its state to its direct neighbor v_2. Nevertheless, v_1's state needs to be recalculated and updated when v_1 receives the latest state of v_0. It indicates the state updates of v_1 and its successors (i.e., v_2, v_3, v_4, v_5, and v_6) are unnecessary.

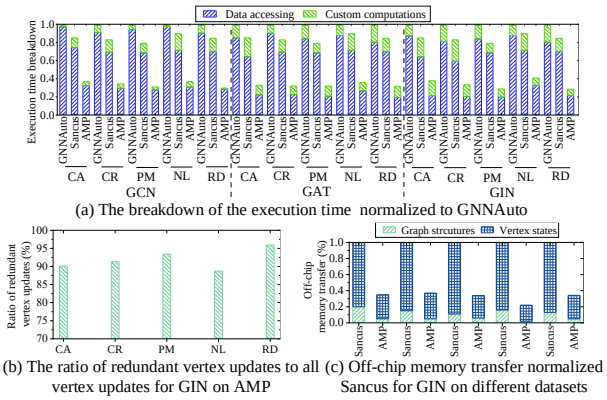
To prove it, we measure the performance of three cutting-edge software AGNN systems (i.e., GNNAuto [4], Sancus [7], and AMP [2]) running on NVIDIA A100 GPU. Section 4 presents the details of the platform and benchmarks. As shown in Figure 3 (a), AMP outperforms other systems in all cases. However, as illustrated in Figure 3 (b), only 8.3%-10.2% of the total updates performed by AMP are useful in the GIN model. As shown in Figure 3 (b) and (c), the unnecessary off-chip communications take up more than 86.8% of the overall execution time, and accesses to vertex states account for 81.2%-91.4% of all accesses.

Compared with the solutions based on the general-purpose processors, specialized hardware accelerators [8, 12, 15] gain significant energy saving and performance speedup for GNN. However, they are also inefficient for AGNN because they also suffer from significant redundant computations and off-chip communications due to the above described unnecessary updates. In other words, the conventional architectures are also not the best platforms for running AGNN, which requires designing a new hardware accelerator to address the challenges of irregular state propagation for AGNN.

2.3 Motivation

2.3.1 Chain-driven Asynchronous Execution Approach.

To improve AGNN inference's efficiency, we propose a chain-aware asynchronous execution approach. It initiates with

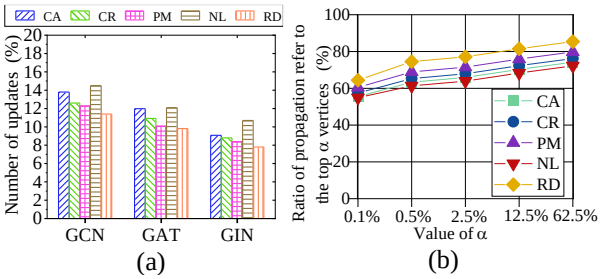


(a) The breakdown of the execution time normalized to GNNAuto

(b) The ratio of redundant vertex updates to all vertex updates for GIN on AMP

(c) Off-chip memory transfer normalized Sancus for GIN on different datasets

Figure 3. Performance of existing solutions



(a) (b)

Figure 4. Performance of software approach: (a) the number of updates of our software approach normalized to that of AMP over one GPU; (b) the ratio of propagations passing through the chains between the top α highest degree vertices

an active vertex, employing on-the-fly depth-first traversal throughout each iteration to track dependency chains among active vertices until all active vertices are visited. Then, the active vertices are processed in a sequential and asynchronous manner, adhering to their respective positions within these chains, which can significantly reduce the number of updates because it ensures that the most recent states of active vertices are efficiently propagated to other vertices in the graph.

Take Figure 2 as an example, where the vertices v_0 and v_1 are assumed to be activated in an iteration. There is a dependency chain (i.e., $v_0 \rightarrow v_1$) between the states of v_0 and v_1. It means that the state propagation originated from v_0 will pass through v_1. Therefore, in our approach, only when the state propagation originated from v_0 has reached v_1, and has been gathered by v_1, v_1 propagates its new state to its neighbors (e.g., v_2) for the state updates of these neighbors. It indicates that the state propagations originated from both v_0 and v_1 can pass through their common successors (e.g., v_2, v_4, v_5, and v_6) together along the dependency chain. In this way, for the state propagations of both v_0 and v_1, the graph data associated with v_2, v_4, v_5, and v_6 are loaded and processed only once in an iteration, which offers the faster state propagation. Figure 4 (a) shows the vertex state updates needed by our approach is less than 14.5% of that of AMP.

Besides, because of the power-law property [5], as shown in Figure 4 (b), in AGNN, more than 61% of the propagations

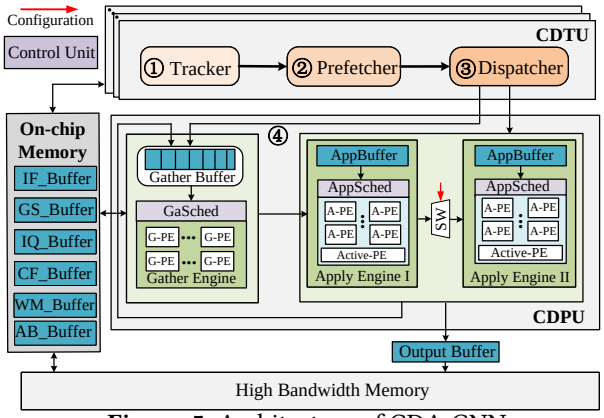


Figure 5. Architecture of CDA-GNN

need to pass through the dependency chains between the top 0.5% highest degree vertices. Thus, our approach also caches the states of the vertices on these frequently-used chains in the on-chip memory for fewer off-chip communications.

2.3.2 Challenges. Although our chain-driven asynchronous execution approach ensures much fewer redundant computations and data accesses, applying our approach into AGNN inference remains challenging, because the runtime overhead occupies 86.3%-96.4% of the total execution time for our tested cases. It is because it needs to take the active vertices as the root to track the dependency chain by irregularly traversing graph on the fly. Besides, it incurs additional instructions and low instruction level parallelism in the general-purpose processors due to the data-dependent branches (which depend on irregular graph structures) of these instructions. It motivates us to design this approach into our hardware accelerator, i.e., *CDA-GNN*, for efficient AGNN inference.

3 *CDA-GNN* Architecture

3.1 *CDA-GNN* Overview

Figure 5 shows the architecture of *CDA-GNN*, which is designed to efficiently support the proposed chain-driven asynchronous execution approach to reduce unnecessary computations and off-chip communications for AGNN inference. *CDA-GNN* contains two key hardware units, i.e., *Chain Driven Traveler Unit* (CDTU) and *Chain Driven Processing Unit* (CDPU), and on-chip buffers. The main functionalities of CDTU, CDPU, and on-chip buffers are as follows.

CDTU. CDTU contains three key parts (i.e., the *Tracker*, the *Prefetcher*, and the *Dispatcher*). For efficient tracking of the dependency chains, an *Intermediate_Queue* (i.e., *IQ_Buffer*) is created by the CDTU to store the intermediate information for graph traversal, and a *Chain FIFO* buffer (i.e., *CF_Buffer*) maintains the visited vertices. Meanwhile, the *Active Bitvector* buffer, i.e., *AB_Buffer*, is also created by CDTU to record the active vertices of the graph, avoiding redundant graph exploration. The *Tracker* is used to track the dependency chains between the active vertices' states, and then the *Prefetcher* is responsible for prefetching the graph data on these chains based on the tracked information. Afterward, the *Dispatcher* dispatches the tasks of state updates to the CDPU.

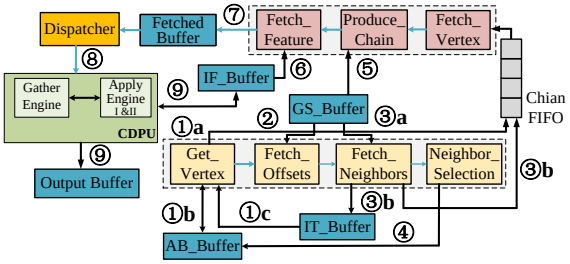


Figure 6. Microarchitecture of CDTU

CDPU. The CDPU is divided into distinct computational modules: a *Gather Engine*, an *Apply Engine I*, and an *Apply Engine II*, each optimized for specific tasks within the AGNN inference process. The *Gather Engine* is used to asynchronously gather the neighbors' states for each target vertex at the *Aggregation* phase, and the *Apply Engines* are used to perform dense and regular MVM operations at both *ApplyEdge* and *ApplyVertex* phases. For AGNN variants with *ApplyEdge* phase, *Apply Engine* I performs the *ApplyEdge*, while *Apply Engine* II performs *ApplyVertex*. *CDA-GNN* implements a traditional systolic array architecture for *Multiply-Accumulate* (MAC) computations, while Active-PE manages the activation functions crucial for GNN inference. The *GaSched* of the *Gather Engine* and the *AppSched* of the *Apply Engine* guarantee the pipeline execution and assign the workloads to the *Gather Engine* and the *Apply Engine*, respectively, which works in a task disperse aggregation mode similar to HyGCN [12] for workload balance and task-level parallelism.

On-chip Buffers and Control Unit. The on-chip memory is composed of several buffers, e.g., *GS_Buffer*, *IF_Buffer*, and *Weight_Buffer*, which are employed to cache various data (e.g., *Graph Structure, Input Feature, and Weight Matrices*) to improve data reuse and reduce unnecessary off-chip communications. To fully exploit the data locality in AGNN inference, it also dynamically caches the states of the vertices on the frequently-used dependency chains, avoiding their data thrashing. Note that *CDA-GNN* adopts the ping-pong buffering technology [12] to hide the access latency.

Workflow of *CDA-GNN*. *CDA-GNN* adopts the vertex-centric programming for efficient AGNN inference. In detail, *CDA-GNN* loads the graph data from off-chip HBM to on-chip memory. The CDTU tracks the dependency chains between the active vertices' states until all active vertices are visited (the step ❶), and the *Prefetcher* takes each active vertex to prefetch the edges of this vertex and the states of both source/target vertices of these edges (the step ❷). These graph data of the prefetched edges are resident in the *Chain FIFO* buffer. After that, the *Dispatcher* dispatches the tasks of neural networks to the *AppBuffer* of the *Apply Engine* and the aggregation tasks to the *Gather Buffer* of the *Gather Engine*, respectively, for updating the target vertices' states (the step ❸). If a task exists in the *Gather Buffer* or the *AppBuffer*, *CDA-GNN* will trigger the *ApplyEdge*, the *Aggregation*, and *ApplyVertex* operations in parallel for target vertices until these buffers are empty (the step ❹).

Table 1. Dataset Information

Datasets	*#Vertices*	*#Edges*	Feature length
Cora (CA)[4]	2,708	10,556	1,433
Citeseer (CR)[5]	3,327	9,104	3,703
Pubmed (PM)[6]	19,717	88,648	500
Nell (NL)[7]	65,755	266,144	61,278
Reddit (RD)[8]	232,965	114,615,892	602

Table 2. Resource Utilization on Xilinx Alveo U280 FPGA

Resource	GCN	GIN	GAT
DSP	24.3%	17.5%	21.4%
LUT	34.7%	26.3%	24.2%
FF	31.4%	29.4%	39.1%
BRAM	42.1%	46.2%	44.5%

3.2 Chain-aware Regularized Data Accessing

Dependency Chain Tracking. To efficiently track the dependency chain for fast state propagation, within each iteration, the *Tracker* takes the unvisited active vertices as the roots to track the chains in a depth-first fashion on the fly until all vertices have been visited. As shown in Figure 6, the *Tracker* has four stages, which are implemented as a pipeline. At the *Get_Vertex* stage, if the *IQ_Buffer* is empty, the *Tracker* scans the *AB_Buffer* to find an active vertex (❶a), then marks this vertex as inactive intermediately for the correctness and inserts this found vertex into the *CF_Buffer* (❶b). If not, the *Tracker* fetches a vertex from the *IQ_Buffer* (❶c). After that, the *Fetch_Offsets* stage obtains the first and last offsets of the vertex from the *Vertice* array of the *GS_Buffer* (❷). The *Fetch_Neighbors* stage loads the unvisited neighbors of this vertex from the *Neighbor* array of the *GS_Buffer* (❸a), and these neighbors are pushed into the *IQ_Buffer* and *CF_Buffer* (❸b). At the *Neighbors_Selection* stage, these fetched neighbors are marked as inactive if these neighbors are active in the *AB_Buffer* (❹). The above procedures repeat until all active vertices have been visited. The sequence of visited vertices between two active vertices, which is resided in *CF_Buffer*, is approximately regarded as the dependency chain between the active vertices' states.

Graph Data Prefetching. To efficiently prefetch graph data on the dependency chains, the *Prefetcher* manages a pipeline with three stages, i.e., *Fetch_Vertex*, *Produce_Chain*, and *Fetch_Feature* stage. At the *Fetch_Vertex* stage, the *Prefetcher* fetches a vertex (i.e., the source vertex ID) from the *CF_Buffer* sequentially. The *Produce_Chain* stage fetches this vertex's neighbors (i.e., the target vertices ID) from the *Edges* array to produce the chain (❺). At the *Fetch_Feature* stage, for each edge of this chain, the source vertex state, the target vertex state, and the edge state are accessed from the *IF_Buffer* (❻). The prefetched edges, the state of edges, and the vertices' states associated with these edges are pushed into the *Fetched Buffer* (❼). Note that each entry in *Fetched Buffer* takes the form of <*Source ID, Target ID, Source vertex state, Target vertex state, Edge state*>.

3.3 Chain-driven Asynchronous Processing

To significantly reduce the end-to-end inference latency of AGNNs, the CDPU orchestrates the operations of *Aggregation*, *ApplyVertex*, and *ApplyEdge* in an asynchronous

pipelined fashion for each vertex along the vertex chain. For each target vertex, the *Dispatcher* simultaneously assigns *Aggregation* tasks to the *Gather Engine* and *ApplyEdge* tasks to *Apply Engine II*. Once the *Gather Engine* and *Apply Engine* II complete their computations, the results are immediately dispatched to *Apply Engine* I for the final state computation. Specifically, *Dispatcher* transfers the states of source and target vertices from the *Fetch_Buffer* to the *Gather Buffer* of the *Gather Engine*. Simultaneously, it ensures the transfer of the state of the source vertex, along with the corresponding edge state to the *AppBuffer* of *Apply Engine* II.

Note that for the AGNN variants without *ApplyEdge* operation (e.g., GCN model [10]), CDPU via the control unit and a hardware *Switch* (i.e., SW), merges *Apply Engine* I and II into a singular larger *Apply Engine*. This merger effectively executes the *ApplyVertex* operation, optimizing hardware resource utilization and enhancing the efficiency of computational units. Specifically, this configuration is controlled by a 1-bit configuration word for the SW: '0' indicating independent operation of *Apply Engine* I and II, and '1' signifying their merger for *ApplyVertex* tasks.

To circumvent pipeline stalling associated with *Aggregation* operations, a unique G-PE design has been proposed. Each G-PE within the *Gather Engine* contains 8 parallel aggregation modules. Each module is equipped with two input queues, *Input Queue* 1 for storing source vertex states and *Input Queue 2* for the target vertex states. These are processed through an adder followed by a multiplexer, determining whether to output the result or cache it as an intermediate value in the *Partial_Buffer*. Besides, the *Gather Engine* adopts feature-level parallelism to execute *Aggregation*.

3.4 Chain-aware Data Caching

For efficient access of the *Input Feature*, *Graph Structure*, *Intermediate_Queue*, *Chain FIFO* buffer, *Weight Matrices*, and *Active Bitvector*, as shown in Figure 5, six on-chip buffers, i.e., *IF_Buffer*, *GS_Buffer*, *IQ_Buffer*, *CF_Buffer*, *WM_Buffer*, and *AB_Buffer*, are used to cache these data, respectively. To further reduce unnecessary off-chip communications, *CDA-GNN* designs a *Chain-Aware Data Caching* (CADC) scheme to manage the vertex states cached in the on-chip *IF_Buffer* because the off-chip communication of vertex states dominates the overall performance of AGNN inference. Note that the other buffers are managed using LRU [6] by default. When the states of a vertex v are requested, if the *IF_Buffer* is not full, v's states will be directly cached in the *IF_Buffer*. Otherwise, CADC first obtains v's priority, i.e., $Pri(v) = \frac{P_G(v)}{P_G} \times D_v$, where P_G denotes the number of chains in graph G, $P_G(v)$ represents the number of chains that include v as a critical vertex and can be calculated by CDTU, and v's degree D_v (i.e., subtracting v's beginning offset from its end offset) are obtained according to the *Vertices* array.

Once the priority score is computed, CADC compares v's priority with those of the vertices currently residing in the *IF_Buffer*. If v has a higher priority than the lowest-priority vertex in the buffer, CADC will replace the lower-priority vertex with v.

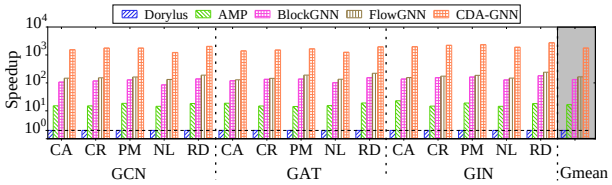


Figure 7. Performance normalized to that of Dorylus

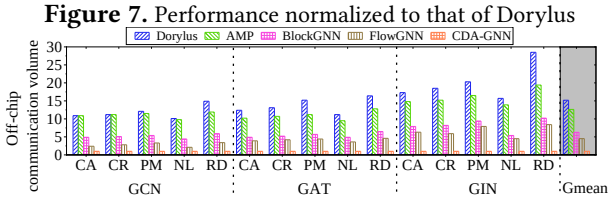


Figure 8. Off-chip communications normalized to *CDA-GNN*

4 EVALUATION

4.1 Experiment Setup

CDA-GNN Settings. We have implemented *CDA-GNN* on a Xilinx Alveo U280 FPGA card, which is equipped with a XCU280 FPGA chip. The FPGA provides 9 MB BRAM resources, 1.3 M LUTs, 2.6 M Registers, and two 4 GB HBM2 stacks. *CDA-GNN* contains 8 CDTUs and the on-chip memory of *CDA-GNN* is implemented using the BRAM resources. Similar to BlockGNN [15], for the PE Group of *CDA-GNN*, we adopt programmable ALUs implement different operations depending on the properties of the AGNN model. *Activate PEs* support activation operations, while the PEs in the *Apply Engines* are constructed as a systolic array. Each PE is a MAC unit with some glue logic. Note that we configured two systolic modules (each module has 32×128 arrays) and 512 G-PEs of the *Gather Engine* for the *CDA-GNN*. We get the clock rate of *CDA-GNN* using Xilinx Vivado 2019.1 and conservatively use 330 MHz in our experiments. The resource utilization of all models is reported in Table 2.

Datasets and Benchmarks. We use five datasets (listed in Table 1) and three GNN models using asynchronous message passing, i.e., GCN [12], GIN [7], and GAT [10], which are commonly used in GNN acceleration studies [15].

Baseline. The performance of *CDA-GNN* is compared with four solutions, i.e., Dorylus [9], AMP [2], BlockGNN [15], and FlowGNN [8]. Dorylus runs on the CPU platform (which has an Intel Xeon 6151 processor with 65 cores at 3.0 GHz and 696 GB DRAM), and AMP runs on the NVIDIA Tesla A100 with 6,912 cores and 80 GB HBM. Besides, *CDA-GNN* is also compared with the cutting-edge GNN accelerators, i.e., BlockGNN and FlowGNN, which run at 330 MHz on the Xilinx Alveo U280 FPGA.

4.2 Experimental Results

4.2.1 Overall Performance. Figure 7 shows the execution time of different solutions normalized to that of Dorylus. Compared with Dorylus, AMP, BlockGNN, and FlowGNN, *CDA-GNN* improves the performance by 1,055x-1,834x (1,173x on average), 142.2x-282.8x (182.4x on average), 8.6x-12.7x (10.2x on average), and 5.8x-9.7x (7.9x on average), respectively. The better performance achieved by *CDA-GNN* comes

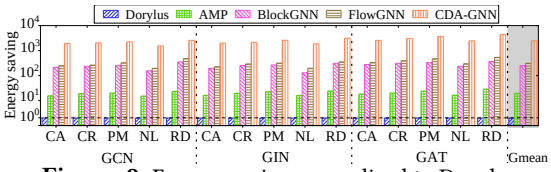


Figure 9. Energy saving normalized to Dorylus

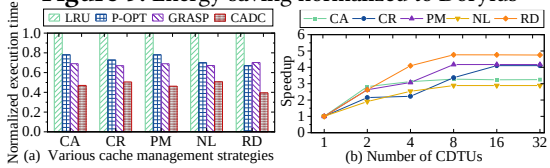


Figure 10. Sensitivity studies of *CDA-GNN*: (a) sensitivity to the replacement strategies; (b) sensitivity to the number of CDTUs

from fewer redundant computations and off-chip communications. Note that our CADC strategy contributes 5.1%-8.4% of *CDA-GNN*'s performance improvement. For our tested instances, the number of vertex state updates of *CDA-GNN* is only 7.8%-14.5% of that of them. It is because the new states of vertices can be propagated along the dependency chains more efficiently in *CDA-GNN* than them. It also indicates that *CDA-GNN* can spare data access costs associated with these redundant updates. Thus, as shown in Figure 8, for the GIN model on *RD*, the amount of off-chip communications of Dorylus, AMP, BlockGNN, and FlowGNN are 28.5x, 19.4x, 10.2x, and 8.4x more than that of *CDA-GNN* on average, respectively, because of much more updates and poorer data locality when accessing vertex states. Figure 9 shows that the energy savings of *CDA-GNN* are 1,785x-2,815x (2,241x on average) and 186.4x-321.4x (242.2x on average) are higher than Dorylus and AMP, respectively. Compared with BlockGNN and FlowGNN, the energy savings of *CDA-GNN* are improved by an average of 12.4x and 8.9x.

4.2.2 Sensitivity Studies.

Figure 10 (a) shows the performance of *CDA-GNN* with different replacement strategies (i.e., LRU [6], P-OPT [1], and GRASP [3]) for the management of the *IF_Buffer* on asynchronous GIN model. It shows that our CADC scheme outperforms the others because states of the vertices on these frequently-used chains can be effectively cached in the on-chip memory. Figure 10 (b) describes the sensitivity to the number of CDTUs. It shows that *CDA-GNN* obtains better performance as the number of CDTUs increases until the number of CDTUs reaches eight because the memory bandwidth is saturated.

5 Related Work

Software AGNN Systems and GNN Accelerators. Dorylus [9] is the first system to support AGNN model. AMP [2] proposes a GPU-based approach for better performance of AGNN. Besides, to accelerate the GNN inference, FlowGNN [8] uses an approach based on the multi-queue streaming, while BlockGNN [15] proposes a pipelined architecture to support block-circulant matrices computation. However, these solutions suffer from irregular state propagation for AGNN.

Asynchronous Graph Processing Solutions. Maiter [13] proposes a delta-accumulative computation approach. AsynGraph [14] further extends this approach to support GPU. However, they can not support the neural network operation of AGNN inference [2].

6 Conclusion

This paper proposes a chain-driven hardware accelerator *CDA-GNN* for efficient AGNN inference. *CDA-GNN* regularizes the state propagations along dependency chains and efficiently cache the states of frequent-used vertices on dependency chains. Our evaluation demonstrates that *CDA-GNN* outperforms the cutting-edge AGNN solutions significantly.

References

[1] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical Optimal Cache Replacement for Graph Analytics. In *Proceedings of HPCA*. 668–681.

[2] Lukas Faber and Roger Wattenhofer. 2023. GwAC: GNNs with Asynchronous Communication. In *Proceedings of LoG*. 1–20.

[3] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-Specialized Cache Management for Graph Analytics. In *Proceedings of HPCA*. 234–248.

[4] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Jure Leskovec. 2021. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *Proceedings of ICML*. 3294–3304.

[5] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of OSDI*. 17–30.

[6] Daniel A. Jiménez. 2013. Insertion and promotion for tree-based PseudoLRU last-level caches. In *Proceedings of MICRO*. 284–296.

[7] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. SANCUS: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks. *Proceedings of the VLDB Endowment* 15, 9 (2022), 1937–1950.

[8] Rishov Sarkar, Stefan Abi-Karam, Yuqi He, Lakshmi Sathidevi, and Cong Hao. 2023. FlowGNN: A Dataflow Architecture for Real-Time Workload-Agnostic Graph Neural Network Inference. In *Proceedings of HPCA*. 1099–1112.

[9] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *Proceedings of OSDI*. 495–514.

[10] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24.

[11] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *Proceedings of ICLR*. 1–17.

[12] 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. In *Proceedings of HPCA*. 15–29.

[13] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2014. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Transanctions on Parallel and Distributed System* 25, 8 (2014), 2091–2100.

[14] Yu Zhang, Xiaofei Liao, Lin Gu, Hai Jin, Kan Hu, Haikun Liu, and Bingsheng He. 2020. AsynGraph: Maximizing Data Parallelism for Efficient Iterative Graph Processing on GPUs. *ACM Transactions on Architecture and Code Optimization* 17, 4 (2020), 29:1–29:21.

[15] Zhe Zhou, Bizhao Shi, Zhe Zhang, Yijin Guan, Guangyu Sun, and Guojie Luo. 2021. BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices. In *Proceedings of DAC*. 1009–1014.