An Efficient Hardware Accelerator Design for Dynamic Graph Convolutional Network (DGCN) Inference

Yingnan Zhao*, Ke Wang§, Jiaqi Yang*, Ahmed Louri*
* The George Washington University, Washington, D.C.
§ University of North Carolina at Charlotte, Charlotte, NC
{yzhao96,Yang_Jiaqi_Cute,louri}@gwu.edu,ke.wang@charlotte.edu

ABSTRACT

Dynamic graph convolutional networks (DGCNs) have been increasingly used to extend machine learning techniques to applications that involve graph-structured data with temporal changes. A typical DGCN model is comprised of graph convolutional network (GCN) layers to capture spatial information, followed by recurrent network (RNN) layers for temporal information. Designing a high-performance and energy-efficient DGCN accelerator is challenging due to the distinct computation and communication requirements of the GCN and RNN layers. Specifically, the computation of GCN layers can be abstracted as Sparse-dense and General Matrix-matrix Multiplication (SpMM and GeMM), while RNN layers involve extensive element-wise addition and Hadamard product in addition to SpMM and GeMM. For data communication, GCN layers necessitate irregular data memory access due to the unstructured distribution of vertices involved in graphs, whereas RNN layers exhibit a predictable memory access pattern. We propose E-DGCN, a high-performance and energy-efficient accelerator design for improved DGCN inference. The proposed E-DGCN comprises reconfigurable processing elements that efficiently support diverse types of data computations required by GCN and RNN layers, a flexible on-chip interconnection design with an adaptive dataflow to improve data reuse during DGCN inference, and a lightweight vertex caching algorithm to leverage data locality and reduce off-chip memory access while processing temporal information. Experimental results show that the E-DGCN achieves 2.2x speed-up and 2.6x energy savings on average as compared to existing DGCN accelerators.

1 INTRODUCTION

Dynamic graphs are pervasive data structures that model pairwise interactions between entities in systems that are constantly changing [1–4]. To extract spatial-temporal features, dynamic graph convolutional networks (DGCNs) have been developed to facilitate machine learning on dynamic graphs applied in a wide variety of application domains, such as social networks [5], recommendation systems, traffic forecasting [6], and many others. A typical DGCN model, as shown in Fig. 1, is comprised of two types of neural network layers, namely the graph convolutional network (GCN) layers to capture spatial dependencies and perform conventional static graph learning [7, 8], and the recurrent neural network (RNN) layers to process temporal information [1, 2, 6] that captures the dynamic change of graph representations.

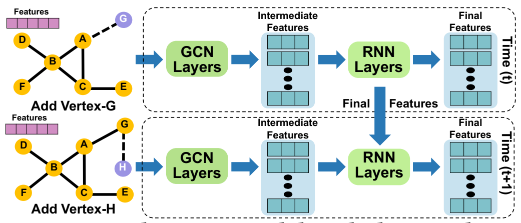

Figure 1: A typical DGCN model includes graph convolutional network (GCN) layers to capture spatial evolution and recurrent neural network (RNN) layers to capture temporal information of input graphs.

In DGCN models, GCN and RNN layers have distinct data computation and communication requirements. Specifically, for computation, each GCN layer takes the adjacency, feature, and weight matrices as input and performs Sparse-dense and General matrix-matrix multiplications (SpMM and GeMM). For each RNN layer, it takes the results of previous GCN and RNN output as input matrices to be multiplied by weight matrices to perform SpMM. Additionally, the intermediate matrices of each RNN layer are used to calculate Hadamard Product [9]. For data communication, GCN layers induce irregular data memory access and unpredictable data reuse, due to the varying numbers and locations of neighboring nodes of each vertex in the input graphs. On the contrary, since RNN layers recurrently use the output matrices of GCN layers whose vertices share the same weight matrices, it results in a predictable data reuse pattern and regular data memory access. Therefore, there is a strong need for a versatile design that can dynamically support diverse computation types required by RNN and GCN layers, along with a flexible dataflow to efficiently handle memory access, explore data locality, and support diverse data reuse patterns.

In this paper, we propose E-DGCN, an efficient architecture design aimed at accelerating DGCN inference with improved performance and reduced energy consumption. The key contributions are abstracted as follows:

- E-DGCN features a unified computing array that includes multiple reconfigurable Processing Elements (PEs). Each PE can dynamically adjust its functionality to efficiently support various computations required by GCN and RNN layers, including SpMM, GeMM, element-addition, Hadamard product, and diverse activation functions.
- E-DGCN deploys an adaptive on-chip dataflow facilitated by a flexible interconnection design. According to GCN and RNN workloads, the flexible interconnection enables

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3658254

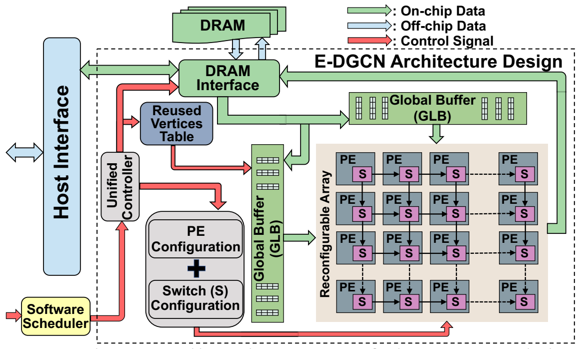

Figure 2: The overall architecture of the proposed design includes Processing Elements (PEs) and Switches (S) used for on-chip data computation and communication, respectively. The Global Buffer (GLB) stores the required data loaded from DRAM. The Unified Controller, connected to the software scheduler, selects the configuration of PEs and switches based on the input workload. The Reused Vertices Table records the indices of vertices that can be further reused, as determined by the proposed algorithm.

the most efficient dataflow (i.e., inter-PE data transmission, spatial-temporal decision, and on-chip buffering strategy) that explores data locality to improve data reuse among PEs.

- E-DGCN integrates a lightweight vertex caching algorithm that exploits the intra-layer and inter-layer data dependencies to locate the dynamic change of the graph (e.g., addition and removal of nodes/edges). Using the proposed algorithm, E-DGCN selectively loads the vertices impacted by the temporal change from the main memory to the accelerator during DGCN inference, which significantly contributes to reducing off-chip data memory access.

Evaluation results with real-world datasets show that the proposed E-DGCN achieves a factor of 2.2x speed-up and 2.6x energy savings on average as compared to previous designs.

2 PROPOSED E-DGCN DESIGN

2.1 Architecture Overview

Fig. 2 demonstrates the overall architecture of the proposed E-DGCN. E-DGCN implements a tile-based unified reconfigurable array that includes customized Processing Elements (PE) and an interconnection design with reconfigurable switches (S) to fulfill the diverse data computation and communication requirements that exist in the DGCN model, respectively. Specifically, based on the input workload, the proposed PE can adjust its functionality to perform various types of computations, including matrix-matrix multiplication, Hadamard product, and activation functions. Each PE is connected to its local reconfigurable switch, and switches are interconnected to support an adaptive dataflow and manage the on-chip data communication efficiently. Details of the proposed E-DGCN PEs and switches are introduced in Sec. 2.2 and Sec. 2.3, respectively. The Global Buffer (GLB) is implemented as a multi-bank scratchpad memory, where each bank can be shared by different PEs through E-DGCN switches. Additionally, E-DGCN features a unified controller connected to the off-chip software scheduler. In response to the input workload from the software scheduler, the unified controller selects the configuration for both E-DGCN PEs

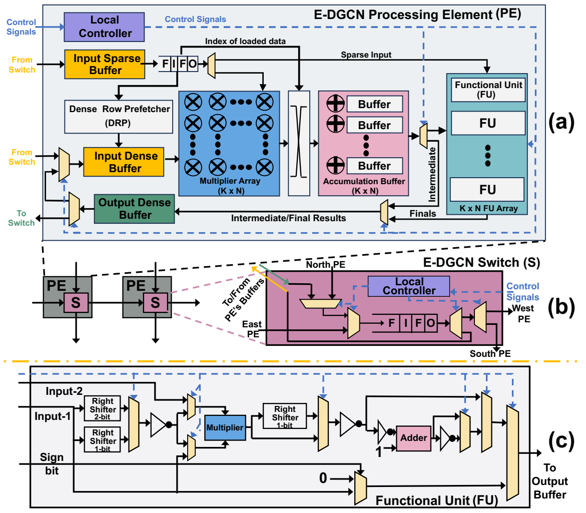

Figure 3: Architecture design of the proposed E-DGCN processing elements (PEs). The multiplier array followed by accumulation buffers, is implemented for Sparse-dense, General matrix-matrix multiplication (SpMM or GeMM), and element-wise addition. (b) Architecture design of the proposed switch (S) for managing on-chip data communication. (c) Functional units (FUs) perform Hadamard Product and activation functions for both GCN and RNN layers. Black and blue links represent data and control signal transmission.

and switches, forwarding the corresponding control signals to the entire reconfigurable array. To efficiently manage data communication between E-DGCN and DRAM while avoiding repeated access to the same set of vertices, a table is utilized to record indices of reused vertices that is directed by a proposed lightweight vertex caching algorithm which is detailed in Sec. 2.4.

2.2 E-DGCN Processing Element (PE)

Fig. 3 (a) depicts details of the proposed E-DGCN Processing Element (PE). Specifically, considering that the input matrices for both GCN and RNN layers consist of both sparse and dense matrices stored in different formats, each modified PE incorporates separate input sparse and dense buffers to store input data separately for improved storage efficiency. To facilitate data synchronization during computations, when the sparse input data is streamed into the First-In-First-Out (FIFO) from the input sparse buffer, the indexes of non-zero elements are sent to the Dense Row Prefetcher. Subsequently, these indexes are forwarded to the input dense buffer to load the specific row. Additionally, each PE integrates a dense output buffer to store the intermediate results or the final outputs of the RNN and GCN layers. The local controller receives the control signal based on the input workload and configures MUX-DeMUXes to manage the local communication. To perform diverse types of computations, each PE consists of three main hardware components: a multiplier array, an accumulation buffer array, and a versatile functional unit (FU) array.

For GCN layers: Each GCN layer takes the normalized adjacency matrix ($\hat{A}$), feature matrix (X), and weight matrix (W) as inputs to calculate the final output $X^{(K+1)}=\sigma(\hat{A}X^{(K)}W^{(K)})$, where K represents the number of GCN layer. The computation consists

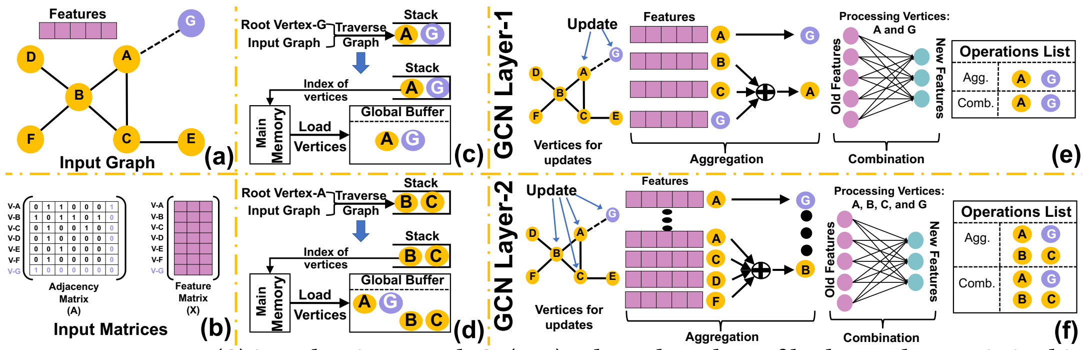

Figure 4: (a) Insert a new vertex (G) into the given graph G=(V, E), where the values of both V and E are six in this example. (b) The update of both adjacency and feature matrices after inserting vertex-G. Note that in real applications, the adjacency matrix is typically normalized. In this example, we set values of the adjacency matrix to 0 or 1 for simplicity. (c) and (d) illustrate the evolution of loaded vertices stored in the global buffer for two GCN layers, respectively. The stack is used to record the datapath during graph traversal. Additionally, (e) and (f) depict the processing of aggregation and combination phases of loaded vertices, providing a detailed list of the operations that occurred.

of an aggregation phase (SpMM) and a combination phase (GeMM). In this case, each PE activates the unified multiplier array followed by accumulation buffers to perform the required computations. Those multipliers located within the same row/column share the same data from the input dense/sparse buffers, respectively. Partial sums of multipliers are then forwarded to the accumulation buffers through a permutation network for further addition. The permutation network is controlled by the index of data stored within the FIFO and is used to pair the multipliers with the corresponding accumulation buffers. Additionally, each PE applies the Outer Product (OP) loop unrolling technique when performing SpMM and GeMM. Compared to the other two common loop unrolling techniques, namely Inner Product (IP) [10] and Row-based Product (RP), OP can mitigate the impact of the workload imbalance induced by the sparsity of the graph's structure [11–14]. Each functional unit, as shown in Fig. 3 (c), is configured to perform activation functions, such as ReLU and Sigmoid.

For RNN layers: Each RNN layer utilizes the output of previous GCN and RNN layers as inputs (x(t) and h(t-1)) multiplied by weight matrices to produce the final result. The computation can be abstracted as $h(t) = f(U \times x(t) + W \times h(t-1))$, where U and W represent the weight matrices, and f is a function that calculates the Hadamard product. Similarly to the GCN layer, the SpMM in the RNN layer is performed by the multiplier array and the accumulation buffers. Additionally, the accumulation buffers also perform element-wise addition on the matrix level with intermediate results of RNN layers. Each FU serves two functionalities: (1) calculating the Hadamard products, and (2) performing activation functions, including ReLU, Sigmoid, and Tanh.

2.3 Flexible Interconnection Design

To effectively manage data communication among PEs and support the adaptive dataflow, the proposed design integrates a flexible interconnection design with reconfigurable switches, as shown in Fig. 3 (b). Each switch features two input ports, two output ports, a local port, and a First-In-First-Out (FIFO) buffer. The input/output ports are used to share data communication with adjacency switches. The FIFO is used to store the data received from the corresponding PE or neighboring switches. Additionally, several MUX/DEMUXes are used to adjust the directions of data transmission, which are configured by the local controller according to the required dataflow. To ensure data synchronization and eliminate per-hop control timing overhead, those switches within the same row or column work in a systolic manner, wherein all the tiles are simultaneously sent across the PE array. The working of the proposed interconnection network and supported dataflows for the GCN and RNN layers are detailed as follows.

For GCN layers: Given that E-DGCN follows a tile-based architecture, both adjacency and feature matrices are partitioned into compact tiles to align with on-chip storage constraints. Each PE handles the computation of a specific pair of tiles. During the **aggregation phase**, to maintain data synchronization, PEs in the same row are assigned to the same set of vertices, while those PEs in the same column manage identical sets of feature vectors. Additionally, each simplified switch configures local MUX-DeMUXes to establish links with neighboring switches in the same row, and each E-DGCN PE is set up to execute Sparse-dense/General Matrix Multiplication (SpMM/GeMM) computations. During the **combination phase**, the horizontal connections remain active. Given that the weight matrix of the GCN layer is small and dense [13], each PE stores the weight matrix directly in its local input dense buffer without sharing. Once the combination phase is completed, different features of the same vertex in those PEs in the same row are streamed to the GLB and then forwarded to the DRAM.

For RNN layers: Each RNN layer includes multiple gate functions and each gate function has two types of input matrices: one is the result of the current GCN layer (A) and the other is the result of the latest RNN layer (B). Each gate function utilizes two distinct weight matrices to separately multiply two types of input and perform the element addition at the end [1, 2]. Therefore, the two types of input matrices are shared among diverse gate functions. Under this circumstance, E-DGCN assigns the entire row of PEs to the same set of vertices and the entire column of PEs to the same type of weight matrix. During the computations, the tiles of the input matrix are shared by PEs horizontally. After finishing the transmission of the input B matrix, the A matrix will be streamed in the same way as the B matrix. Concurrently, the result of each PE stays locally for accumulation. When completing both types of

Algorithm 1 Light-weight Vertex Caching Algorithm

```
1: Inputs: The number of vertices (V), the graph structure
   (adjacency_matrix), the root vertex (R), the array includes the
   id of vertices located in global buffer (array_reuse_vertices[]),
   and the layer number of the ongoing GCN model (n)
2: Outputs: The array that includes the required vertices to be
   loaded from the main memory.
3: Begin:
4: // record the visit status of vertices.
5: boolean visited[V]
6: level = 0 // record the height of the DFS tree
7: // Initialize the status of all vertices to un-visited
8: for v in V do
9:     visited[v] = false
10: end for
11: stack(G) // the datapath for recording traversal
12: // A DFS-based recursive traverse function
13: traverse(R, adjacency_List[R], level)
14: // record the vertices needed from the main memory
15: array = stack() - array_reuse_vertices[]
16: return array
```

Algorithm 2 Recursive Traverse Function

```
1: Inputs: Root vertex (R), adjacency List of vertex-R
   (adjacency_List[R]), the degree of the DFS tree (level)
2: Begin:
3: void traverse (Vertex R, Adjacency_List adjacency_List[R], int
   level)
4: for v in adjacency_List[R] do
5:     if visited[v] then
6:         // v is already in the stack
7:     else
8:         stack(v)
9:         visited(v) = true
10:    end if
11:    if level == n then
12:        // Begin backtrace, and the recursion is finished.
13:    else
14:        traverse (v, adjacency_List[v], level+1)
15:    end if
16: end for
```

input matrices, vertical links are active while horizontal links are disabled for result accumulation.

2.4 Lightweight Vertex Caching Algorithm

Dynamic graphs encompass various operations involving the addition or removal of edges, features, and vertices over time [3, 15]. These operations impact DGCN inference as changes occur in the input adjacency and feature matrices. Fig. 4 shows an example of the addition of vertices in DGCN, with other operations being abstracted similarly. We utilize a DGCN model that includes two conventional GCN layers in this example. As shown in Fig. 4 (a), the system inserts a new vertex (G) to the given graph G=(V, E), where V and E represent the number of vertices and edges, respectively (both V and E are six in this example). By adding the new vertex, modifications occur in both the adjacency and feature matrices, as illustrated in Fig. 4 (b). In this scenario, only vertices A and G undergo the aggregation and combination phases of GCN Layer-1. This is because the addition of vertex-G specifically influences the arrangement of vertex-A's neighbors. Following the completion of Layer-1, the updated features of vertices A and G necessitate updates in the second GCN layer for vertices A, B, C, and G, as illustrated in Fig. 4 (d). This is because the update of vertex A has an impact on both vertices B and C. With two GCN layers, the operation of inserting a new vertex G only impacts the results of vertices A, B, and C, which is limited for DGCN inference. Additionally, throughout the entire GCN inference, the values of the adjacency list and intermediate results of vertices A and G can be reused.

Based on the aforementioned observation, E-DGCN deploys a lightweight vertex caching algorithm based on Deep-First Search (DFS) to efficiently locate those vertices impacted by the update, efficiently manage, and avoid redundant data memory access, as illustrated in Algorithm-1. Specifically, the proposed algorithm designates the updated vertex (N) as the root node and loads the adjacency list of the vertex N to an array. Subsequently, the proposed algorithm calls a recursive traverse function, as shown in Algorithm-2, which takes the root node and the adjacency list as inputs to traverse the graph, identifying those vertices impacted by the updated vertex N. The termination condition for the recursive function is based on the height of the search tree. The maximum value of the height is determined by the total number of GCN layers involved in DGCN models. After the traverse function is completed, the system receives an array (stack), including vertices that need to be loaded for the current GCN layer. As the number of processing GCN layers increases, the height of the traversal tree also grows. Moreover, vertices at the top of the tree are frequently accessed. To avoid redundant off-chip memory access of these vertices, the proposed design records the index of these vertices using an additional array and stores the intermediate results of these vertices on-chip for subsequent reuse, thus reducing data memory access. Given a DGCN model with N GCN layers, the time complexity of the proposed algorithm is O(V+E), where V and E represent the number of vertices and edges traversed by the algorithm.

An example of the proposed vertex caching algorithm is demonstrated in Fig. 4. As shown in Fig. 4, the proposed algorithm designates the inserted vertex G as the root node. Subsequently, the system traverses all its neighbors (A in this example) and stacks them for the first GCN layer. Fig. 4 (c) and (e) illustrate the vertices stored inside the global buffer and the detailed operations performed on the loaded vertices for the first GCN layer, respectively. After completing the aggregation and combination phases for all the loaded vertices, the proposed algorithm uses these loaded vertices as root nodes to explore new required vertices at a deeper level through the traverse function. Simultaneously, the results of both vertex A and G from the first GCN layer are stored within the on-chip buffer for future data reuse. The index and the height of A and G are stored in the on-chip table for easy lookup. Fig. 4 (d) and (f) illustrate the vertices stored inside the global buffer and the detailed operations performed on the loaded vertices for the Layer-2 of GCN inference, respectively.

3 EVALUATION AND ANALYSIS

3.1 Experiment Setup

Hardware simulator. We build a cycle-accurate simulator in C++ language to model the hardware behavior and evaluate the performance of the proposed design. Specifically, the designed simulator

Table 1: Details of datasets used for evaluation

Datasets	Vertices	Edges	Features	Description
Wikipedia (WI)	9,227	157,474	172	Citation Graph
Reddit (RD)	55,863	858,490	602	Social Graph
Twitter (TW)	8,861	119,872	768	Sharing Graph
PubMed (PM)	1,917	88,648	500	Citation Graph

precisely counts the exact number of on/off-chip data memory read and write operations, which is used to estimate the energy consumption of the memory access according to [16]. To measure the area consumption, we model all the proposed hardware logic, including the unified reconfigurable array, controller, MUX-DeMUXes, FIFOs, and other hardware components. We use the Synopsys Design Compiler with the TSMC 45nm library for the synthesis. We set the clock frequency at 1 GHz. We use Cacti 6.0 [17] to estimate the area, power, and access latency of all types of on-chip buffers.

Architecture Configuration. We implement the proposed design including K×K PEs/simplified routers array. Each PE includes an N×N multiplier array connected to an accumulation buffer with N×N adders. Additionally, each PE includes N×N functional units for required activation functions. During the evaluation, we set the values of K and N to be 8 and 4, respectively. Additionally, in Sec. 3.5, we conduct a scalability analysis to illustrate the relationship between performance and the dimensions of the unified array. Since the proposed E-DGCN is a tile-based architecture, the capacity of the input sparse/dense buffer within PEs is 320 KB and 4 KB to accommodate the required data of each tile. The output buffer used to store the intermediate and final matrices is 256 KB. The capacity of the FIFO inside each simplified router is 320 KB, designed to accommodate the largest tile of data that needs to be transferred between modified PEs such as feature matrices.

Datasets and Benchmarks. Table 1 illustrates four dynamic graphs used for evaluation in this paper with the number of vertices and edges [1, 18, 19]. We consider two typical DGCN models: T-GCN [6] and CD-GCN [1]. Specifically, T-GCN includes GCN and Gated Recurrent Unit (GRU). CD-GCN includes GCN and Long-Short-Term-Memory (LSTM) models. The 32-bit floating-point representation is used in the evaluation, which proves to be sufficient for maintaining inference accuracy [20, 21].

Baselines. To perform fair cross-platform comparisons, we compare E-DGCN to the state-of-the-art DGCN software framework, PyGT [22], on CPU (Intel Xeon V4) and GPU (NVIDIA A100), identified as PyGT-CPU and PyGT-GPU, respectively. We also compare E-DGCN to prior work on DGCNs such as ReaDy [2]. Additionally, since DGCNs mainly include two basic kernels, GCN and RNN, we compare E-DGCN to the combination of state-of-the-art GCN (I-GCN [12]) and RNN (ERA-LSTM [23]) accelerators, connecting as a tandem-engine architecture (denoted as I-LSTM).

3.2 Data Memory Access

Fig. 5 shows the normalized off-chip data memory access of the proposed design (E-DGCN) compared to previous works. All the evaluation results are normalized to the proposed design, and a lower value indicates better performance. E-DGCN outperforms previous approaches for several reasons. Firstly, compared to conventional processing units (CPU and GPU), E-DGCN can directly utilize the sparse input matrices stored in a compressed format without under-utilizing hardware resources, such as on-chip data storage. In comparison to ReaDy, E-DGCN implements a dynamic dataflow when performing different neural network models to maximize on-chip data reuse, thereby reducing off-chip memory access. Additionally, benefiting from the proposed algorithm, E-DGCN can selectively load required vertices during each GCN layer instead of loading the entire graph. This approach minimizes unnecessary data memory access induced by unimpacted vertices when operations happen, such as when inserting or removing vertices from the graph. Moreover, the proposed algorithm allows E-DGCN to effectively store the intermediate results of inter-layers of the GCN model locally instead of writing them back to the main memory. All methodologies mentioned above help the proposed design reduce off-chip data memory access by nearly 50% on average compared to the ReaDy design.

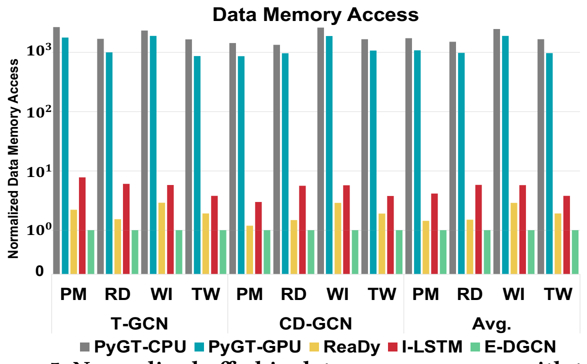

Figure 5: Normalized off-chip data memory access with two different DGCN models on four datasets (lower is better).

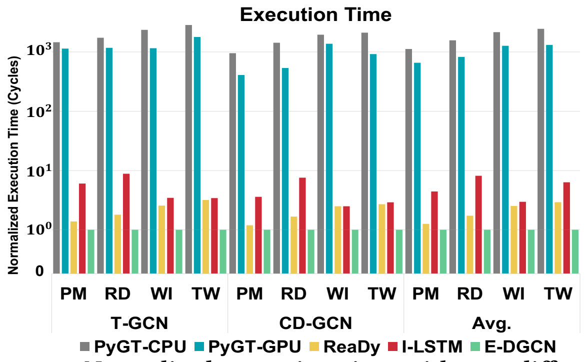

Figure 6: Normalized execution time with two different DGCN models on four datasets. All the data is evaluated in cycles (lower is better).

3.3 Execution Time

Fig. 6 illustrates the execution time of the proposed design in comparison to previous works, measured in terms of the total number of execution cycles. The execution time includes the duration of configuration calculations, the transmission of relevant control signals, and the setup of the entire architecture. E-DGCN achieves an average of 54.5% reduced execution time (2.2 × speedup) compared to

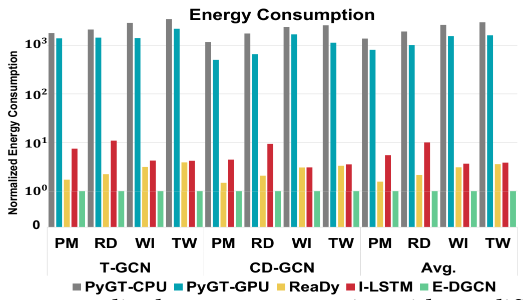

Figure 7: Normalized energy consumption with two different DGCN models on four datasets (lower is better).

previous works. This is because E-DGCN provides improved computation and communication. Specifically, compared to crossbar-based approaches, the proposed design utilizes the sparse data format directly, eliminating the time required for data format conversion from sparse to dense. Furthermore, compared to the tandem-engine design where two engines may idle while waiting for results from each other, the unified reconfigurable array helps minimize idle time, leading to improved performance. Furthermore, the flexible interconnection design and vertex caching algorithm contribute to performance improvement with reduced off-chip data memory access and fine-tuned dataflow.

3.4 Energy Consumption

All approaches estimate the related energy consumption according to [17], and are normalized based on the proposed design. As shown in Fig. 7, the proposed design achieves around 61.5% (2.6×) energy savings on average compared to previous customized designs. This is because our proposed architecture design has reduced DRAM accesses, improved on-chip data reuse, and efficient execution of data computations.

3.5 Scalability Analysis

We evaluate the scalability of the proposed architecture by varying the dimension of the unified reconfigurable array, using the T-GCN model with the Twitter dataset. The array size is scaled from 1×1 to 8×8. All performance metrics are measured in cycles and normalized to the performance of the 1×1 dimension. As depicted in Fig. 8, even though the scaling of PE array size continues to induce execution time reduction, the improvement is diminished. This is because the main memory bandwidth becomes the primary performance bottleneck as the dimension of the PE array increases, especially when substantial computing resources are available for the given DGCN models in use, according to the Roofline model [24].

3.6 Area Consumption

We evaluate the area consumption of the proposed architecture under TSMC 45 nm technology, and E-DGCN occupies a total area of 45.58 (mm^2). Specifically, for the proposed PE, the three types of input buffer consume around 68.6% of the total PE area, while the switch with embedding a FIFO (320 KB) consumes around 30.5%. In the overall architecture design, the 4 × 4 unified reconfigurable array occupies the majority of the area, accounting for approximately 72.3%. The reused vertices table consumes 0.2% chip area.

4 CONCLUSION

We propose an efficient accelerator design named E-DGCN for high-performance and energy-efficient Dynamic Graph Convolutional Network (DGCN) inference. Specifically, E-DGCN includes reconfigurable Processing Elements (PEs) with a flexible interconnection design to support diverse types of data computations and various dataflows, with maximized on-chip data reuse and hardware utilization. Additionally, a lightweight vertex caching algorithm is proposed to exploit data locality, enabling E-DGCN to selectively load required vertices from the main memory during DGCN inference, thereby reducing off-chip data memory access.

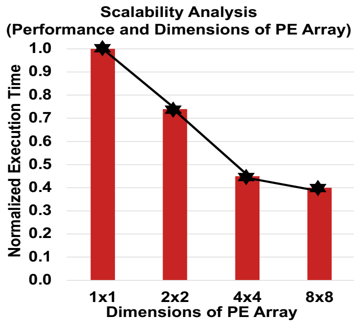

Figure 8: Normalized execution time when scaling the dimensions of the unified reconfigurable array (lower is better).

REFERENCES

[1] Aldo Pareja et al. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proc.of AAAI'20*, 2020.
[2] Yu Huang et al. Ready: A reram-based processing-in-memory accelerator for dynamic graph convolutional networks. *In TCAD*, 2022.
[3] Mahbod Afarin et al. Commongraph: Graph analytics on evolving data. In *Proc.of ASPLOS'23*, 2023.
[4] Emanuele Rossi et al. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.
[5] Avery Ching et al. One trillion edges: Graph processing at facebook-scale. *Proc. of the VLDB Endowment*, 2015.
[6] Ling Zhao et al. T-gcn: A temporal graph convolutional network for traffic prediction. *In ITS*, 2019.
[7] Justin Gilmer et al. Neural message passing for quantum chemistry. In *Proc. of ICML'17*, 2017.
[8] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.
[9] Roger A Horn. The hadamard product. In *Proc. Symp. Appl. Math*, 1990.
[10] Shmuel Winograd. A new algorithm for inner product. *In TC*, 1968.
[11] Tong Geng et al. Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing. In *Proc. of MICRO'21*, 2020.
[12] Tong Geng et al. I-gcn: A graph convolutional network accelerator with runtime locality enhancement through islandization. In *Proc. of MICRO'21*, 2021.
[13] Jiajun Li et al. Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks. In *Proc. of HPCA'21*, 2021.
[14] Jiajun Li et al. Sgcnax: A scalable graph convolutional neural network accelerator with workload balancing. *In TPDS*, 2021.
[15] Maciej Besta et al. Practice of streaming processing of dynamic graphs: Concepts, models, and systems. *In TPDS*, 2021.
[16] Song Han and Others. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 2016.
[17] Naveen Muralimanohar et al. Cacti 6.0: A tool to understand large caches. *University of Utah and Hewlett Packard Laboratories, Tech. Rep*, 2009.
[18] Palash Goyal et al. Dyngem: Deep embedding method for dynamic graphs. *arXiv preprint arXiv:1805.11273*, 2018.
[19] Ryan Rossi and Nesreen Ahmed. The network data repository with interactive graph analytics and visualization. In *Proc. of AAAI'15*, 2015.
[20] Seyed Mehran Kazemi et al. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research*, 2020.
[21] Mingyu Yan et al. Hygcn: A gcn accelerator with hybrid architecture. In *Proc. of HPCA'20*, 2020.
[22] Benedek Rozemberczki et al. Pytorch geometric temporal: Spatiotemporal signal processing with neural machine learning models. In *Proc. of CIKM'21*, 2021.
[23] Jianhui Han et al. Era-lstm: An efficient reram-based architecture for long short-term memory. *In TPDS*, 2019.
[24] Samuel Williams et al. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 2009.