GNN-PIM: A Processing-in-Memory Architecture for Graph Neural Networks

Zhao Wang[1], Yijin Guan[2], Guangyu Sun[1(✉)], Dimin Niu[2], Yuhao Wang[2], Hongzhong Zheng[2], and Yinhe Han[3]

[1] CECA, Peking University, Beijing, China
gsun@pku.edu.cn
[2] Alibaba DAMO Academy, Hangzhou, China
[3] Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China

Abstract. Graph neural networks (GNNs) have attracted increasing interests in recent years. Due to the poor data locality and huge data movement during GNN inference, it is challenging to employ GNN to process large-scale graphs. Fortunately, processing-in-memory (PIM) architecture has been widely investigated as a promising approach to address the "Memory Wall". In this work, we propose a PIM architecture to accelerate GNN inference. We develop an optimized dataflow to leverage the inherent parallelism of GNNs. Targeting the dataflow, we further propose a hierarchical NoC to perform concurrent data transmission. Experimental results show that our design can outperform prior works significantly.

1 Introduction

As the volume of non-Euclidean data [32] keeps growing, graph neural networks (GNNs) have attracted great attention because of their capability to express complex relationships and inter-dependency between objects. Inspired by the success of convolutional neural networks (CNNs) in computer vision domain, spatial-based GNNs have advanced rapidly in recent years. Most GNNs are composed of several "convolutional" layers, as those in CNNs [11,19].

The "convolution" operation in GNN can be roughly divided into two phases [30]. **Aggregation Phase** aggregates nodes' information from their multi-hop neighbours by pointer-chasing operations. This phase incurs intensive random memory accesses. **Handling Phase** feeds the aggregated features into a neural network to generate new features. Both computation and aggregation are regular in this phase. Having totally different processing patterns, the two phases consume considerable yet distinct resources. Thus, each of them may become the bottleneck of the whole system. We can tell that GNN processing encounters similar challenges as those in the graph processing and CNN inference, which have been well studied separately.

Recently, researchers propose to accelerate GNN processing by optimizing both software framework [20] and hardware architecture [30]. Ma et al. [20] developed

https://doi.org/10.1007/978-981-15-8135-9_6

a programming framework for GNN processing by extending conventional edge-centric processing framework. Yan et al. [30] employs a hybrid architecture to handle the two phases separately. However, the size of graph datasets for GNNs keeps increasing in many applications, such as, social networks and e-commerce transactions [11,22,28]. As a result, the requirements for large capacity and high bandwidth of the memory subsystem are further raised.

To address this challenge, the processing in memory (PIM) architecture has been considered as a promising solution. It can greatly alleviate the overheads of data movement by offloading computation tasks to the storage. PIM architectures have been extensively explored for both DNN models and graph processing applications [8,12,24,27]. However, it is inefficient to directly employ prior works for GNNs. For example, PRIME [8] proposes a comprehensive design of processing elements for NN processing on ReRAM. But its processing dataflow is not suitable for GNNs, and the bus-based communication among PEs may cause significant performance loss during the random sampling in the aggregation phase. Customized for graph processing, GraphR [27] addresses the random access problem in aggregation phase by employing edge-center processing model. However, the architecture for a conventional graph task lacks the support for efficient tensor processing.

Taking both aggregation phase and handling phase into consideration, we propose a PIM architecture equipped with a carefully designed NoC, called GNN-PIM. Our architecture employs SAGA [20] as the programming model and provides efficient PIM implementation. In GNN-PIM, we first leverage the computing capability of PIM architecture to support operations in handling phase. Then, we design a hierarchical interconnection network for efficient data movement in the aggregation phase. To the best of our knowledge, this is the first PIM accelerator proposed for GNNs. The contributions of this work are summarized as follows:

- By exploiting the inherent parallelism of GNNs, we propose a PIM architecture called **GNN-PIM** to accelerate the inference.
- We propose an optimized dataflow to map GNN inference efficiently on GNN-PIM, which is compatible with SAGA programming model.
- To facilitate the dataflow, we develop a hierarchical NoC providing high-bandwidth for data transmission.

The rest of this paper is organized as follows. In Sect. 2, we briefly review the computation of GNN inference with SAGA model. In addition, the basics of PIM architecture are introduced. In Sect. 4, we propose the GNN-PIM architecture and describe the execution dataflow on it. The hierarchical interconnection network of GNN-PIM is then presented in Sect. 3. Section 6 provides evaluation to demonstrate the efficiency of GNN-PIM, and Sect. 7 concludes this paper.

2 Background

In this section, we first introduce the operations of GNN inference and a dedicated programming model called SAGA. Then, we present basics about PIM designs.

2.1 GNN Inference and SAGA

GNNs are emerging neural networks that operate directly on graphs with non-Euclidean data. The idea of GNNs roots in CNNs and graph embedding [32]. Inheriting the ideas of parameter sharing from CNN and recursive execution from RNN, GNN convolution with kernel size one could be formulated as follows:

$$h_v^t = f(h_{v' \in N(v)}^{t-1}, h_v^{t-1}, I_e)$$

where $N(v)$ is the neighbourhood of v, h_v^t is the feature of vertex v in iteration t, and I_e are the labels of edges that are connected with v. To process the operations involved in GNN inference, vertices need to fetch features from their neighboring vertices. Since the neighbours of nodes are relevant to the topology of graph, random memory access occurs frequently. Such random memory access pattern can cause significant bandwidth waste and performance degradation.

Algorithm 1. GGCN in SAGA

```
input: p = $[W_H^l, W_C^l, W_e^l]$, $vertex^l$
output: $vertex^{l+1}$
 1: $edge^l$ = Scatter($vertex^l$)
 2: acc = ApplyEdge($edge^l, p$)
 3: function ApplyEdge($edge^l, p$)
 4:     $\eta$ = sigmoid(p.$W_H^l \otimes edge^l.src$ + p.$W_C^l \otimes edge^l.dst$)
 5:     return $\eta \odot edge^l.src$
 6: end function
 7: accum = Gather(acc)
 8: $vertex^{l+1}$ = ApplyVertex($vertex^l$, accum, p)
 9: function ApplyVertex($vertex^l$, accum, p)
10:     return ReLU(p.$W^l \otimes accum$);
11: end function
```

To regulate the processing of GNN inference, Neugraph [20] proposes a programming model called SAGA. SAGA converts a primitive program to the edge-centric dataflow and alleviates random memory access. In SAGA, a GNN is decomposed and converted into *ApplyEdge* and *ApplyVertex* to handle edges and vertices, respectively. Combined with *Scatter* and *Gather* for data transmission, the processing of GNN inference is illustrated in Algorithm 1.

ApplyEdge takes parameters of the network (p) and scattered features (*edge*) as input, and generates the partially accumulated features (*acc*) which is further reduced by *Gather*. The inputs of *ApplyVertex* consist of *accum*, vertex data tensor (*vertex*), and *p*. *ApplyVertex* outputs new vertex features through a neural network, usually an MLP. *Scatter* and *Gather* perform data broadcasting before *ApplyEdge* and data gathering before *ApplyVertex*, respectively.

2.2 PIM Basis

PIM architecture breaks the “Memory Wall” by moving computation to where the data are stored. Thus, it is friendly for memory-intensive applications such as graph processing and deep neural networks. Recently, various PIM architectures have been proposed. Basically, these approaches can be categorized according to the computation unit.

For the first type, the computation unit is still based on the traditional logic, which is integrated close to or inside the memory array [6,7,10,25]. This is also known as near-data-processing (NDP). For the second type, the computing unit is just built using the memory cell. Prior works have demonstrated that both traditional SRAM and DRAM technologies [1,17,23] and emerging memory technologies, such as ReRAM [24,29,31], MRAM [3–5], and PCM [18], can be leveraged for PIM designs.

Recently, many works [2,8,12,21,24,26,27] have revealed the potential benefits of using PIM architectures for both DNN and graph computing applications. However, these accelerators either lack the ability to handle graph-like data transmission pattern during GNN inference, or are unable to process NN computation efficiently. As a result, we propose the GNN-PIM architecture to simultaneously address the two drawbacks of prior works.

3 GNN-PIM Architecture

In this section, we introduce GNN-PIM’s hierarchical architecture. The top hierarchy is **Node Cluster**, composed of a number of **Node**s. Each node owns several memory chunks and a set of **processing elements** (PEs).

3.1 Node

Node is the unit for performing computation on the sub-graphs. They are fundamental modules to build the whole architecture. As shown in Fig. 1, the micro-architecture of nodes includes several memory chunks and a processing core.

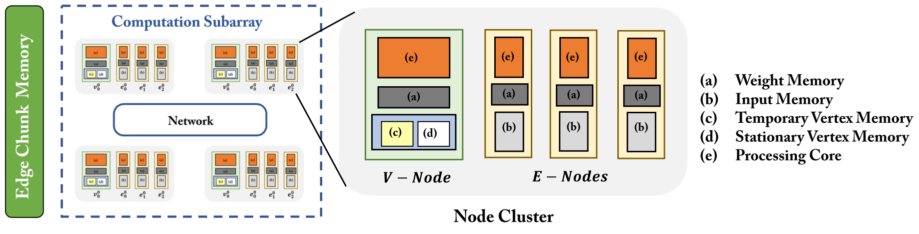

Fig. 1. Overall architecture of GNN-PIM

A node can be configured as either an E-Node or a V-Node, depending on the type of data stored in the memory chunk that they are working on. During GNN inference, E-Nodes are assigned with computation on edges, i.e. *ApplyEdge*. V-Nodes handle remaining tasks, consisting of *Scatter, Gather* and *ApplyVertex*. To simplify hardware design and provide more flexibility for mapping strategy, E-Node and V-Node share the same architecture and just differ in control logic which could be configured according to graph status.

We equip each V-Node with a processing core, weight memory for the storage of neural weights of *ApplyVertex*, along with temporary vertex memory and stationary vertex memory to store source vertex chunks and destination vertex chunks, respectively. An E-Node also consists of a processing core, a weight memory, as well as an input memory for storing input features of the layer that they are assigned to.

Matrix-Vector (MV) multiplication is the key operation of a processing core. It is used to process the most computing intensive functions in the phases of *ApplyEdge* and *ApplyVertex*. The computing architecture for MV operations have been extensively studied in prior works. Note that several non-MV functions are also needed, which are implemented with dedicated logic.

3.2 Node Cluster

Multiple Nodes can be grouped into a **Node Cluster** connected by NoC. The node cluster is composed of several nodes connected by a interconnection network. Some nodes are configured to be V-Nodes and the others are E-Nodes according to the sub-graph topology. In this way, more edges exist in the sub-graph, more nodes are configured to be E-Nodes. Since we employ a homogeneous design for the underlying hardware, all the nodes can be configured arbitrarily. As a result, GNN-PIM is able to configure nodes to balance the workloads according to the graph topology. Nodes in GNN-PIM work more like Multi-Processors System-on-Chip (MPSoC), transmitting data via the network-on-chip (NoC). In order to design an efficient NoC, we first need to understand the execution dataflow, which is introduced in the next section.

4 Execution Dataflow

We design an optimized dataflow that can map SAGA to GNN-PIM in a pipelining style. It is based on the phase sequence of *Scatter*, *ApplyEdge*, *Gather*, and *ApplyVertex*.

4.1 Mapping Strategy

As shown in Fig. 2, the whole graph is divided into multiple sub-graphs by dividing vertices into multiple chunks. Edges, which take the vertices in the chunk as destination, are also contained in the sub-graph. The set of edges in the sub-graph could be further divided by their source vertices. We put the edges with

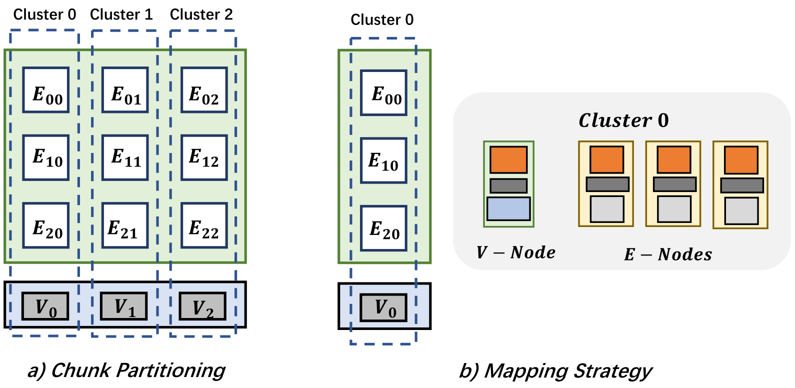

Fig. 2. Edge and vertex chunks partitioning and mapping strategy

source vertices in the same sub-graph into a block, named as edge chunk. And we assign a cluster of our architecture with a sub-graph, which contains a column of edge chunks for processing *ApplyEdge* on, together with the vertex chunk for handling *ApplyVertex* phase. In a cluster, we let an E-Node process a layer of neural network of *ApplyEdge*, and V-Nodes are responsible for handling *Scatter*, *Gather* and *ApplyVertex*.

As for most of natural graphs obey the power-law, the edges in each chunk may be diverse significantly, resulting in imbalanced workload for clusters. To handle this situation, we employ the Index Mapping Interval-Block Partition (IMIB) algorithm introduced in [9]. IMIB firstly removes the blank vertices, then hashes the vertices into different chunks using the modulo function. The time complexity of this algorithm is $O(m)$, where m denotes the size of edges.

4.2 Setup and Terminology

ApplyEdge is the most time-consuming state, which is normally composed of multiple layers of neural networks. To simplify discussion, we assign the computation task of each layer to an individual E-node. In the example of Fig. 2, the *ApplyEdge* phase employs a three-layer MLP. Thus, we have three E-nodes in a cluster for each layer computation.

Terminologies

– E_{ij} denotes the edge chunk located at the i-th row and the j-th column, and V_i means the i-th vertex chunk.
– L_i denotes the i-th E-Node, which handles the computation of i-th layer ApplyEdge.
– $S(E_{ij})$ denotes the output features of *Scatter*, which take V_i and V_j as input and process the operations related to E_{ij}.
– $L_t(E_{ij})$ denotes the output features of t-th layer on the edge chunk E_{ij}.

4.3 Dataflow Description

We divide the whole execution flow into multiple rounds. A round can be further divided into two sub-rounds. During the computation sub-round, input features and neural weights stored in their local buffer are fed into processing arrays to do the computation. During each communication sub-round, nodes forward their output features, and clusters exchange vertex chunks stored in temporary memories with each other in a circular manner, as shown in Fig. 3.

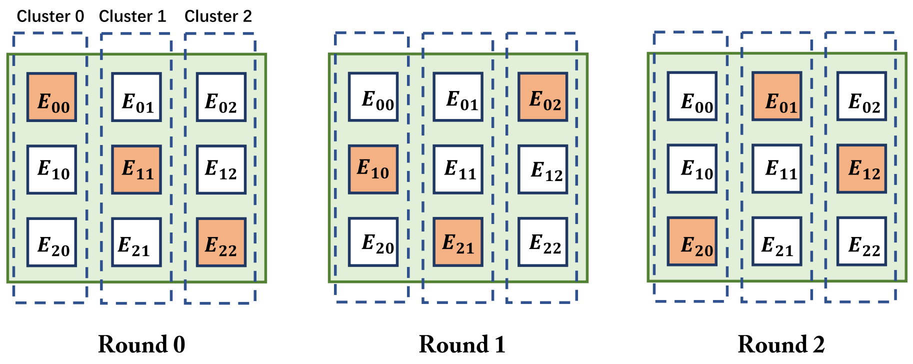

Fig. 3. Edge chunks being processed in each round

Initially, the V-Node in the cluster loads the i-th vertex chunk and copies it into both the stationary vertex memory and temporary vertex memory. In the following paragraphs, we will introduce the details in Round 0, Round 1 to illustrate how the dataflow works.

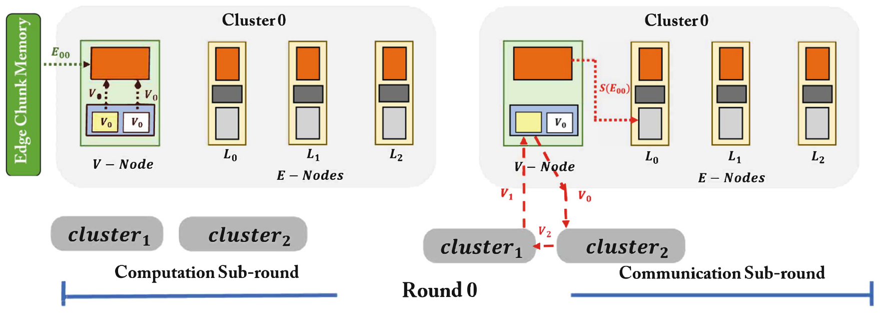

Fig. 4. GNN-PIM dataflow during Round 0

The dataflow during Round 0 is shown in Fig. 4. During the computation sub-round, clusters load edge chunks in the diagnose of the adjacent matrix of the graph, as shown in Fig. 3. Then V-Nodes feed data in edge and vertex chunks into processing cores, respectively. During the communication sub-round, the V-Node in cluster 0 forwards its output features $S(E_{00})$ to the edge node. Cluster 0 transmits its vertex chunk V_0 to its neighbor cluster 2. At the same time, it receives the vertex chunk V_1 from cluster 1.

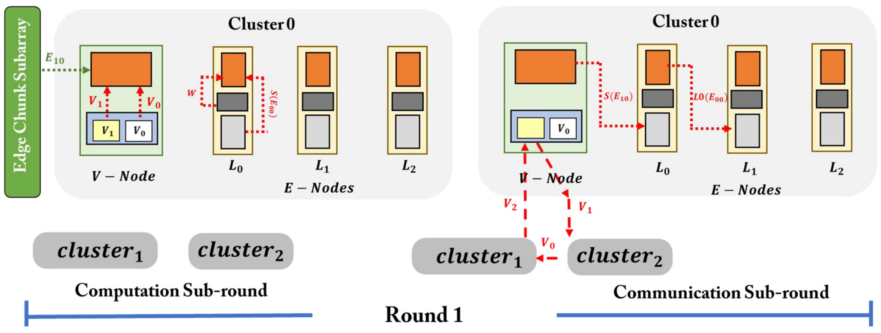

Fig. 5. GNN-PIM dataflow during Round 1

The dataflow during Round 1 is illustrated in Fig. 5. During the computation sub-round, node clusters shift the edge chunks they works on, loading new edge chunks from edge chunk memory, as Fig. 3 shows. Noted that the vertex chunks stored in stationary vertex memory and temporary vertex memory are exactly the destination vertices and source vertices of the new edge chunk respectively. E-node L_0 starts its computation in this round. During the communication sub-round, both V-Nodes and E-Nodes forward their outputs to their following nodes in the pipeline. At the same time, each cluster also forwards its vertex chunk accordingly.

In the subsequent rounds, GNN-PIM repeats this process until clusters receive the same data as the one in their stationary memory, which indicates that *ApplyEdge* and *Gather* finish. After that, V-Nodes continue to perform *ApplyVertex* on the aggregated data to generate new features. The *ApplyVertex* phase is not shown due to page limitation.

5 Interconnection Hierarchy

As reported by Ji et al. [12], data transmission may consume considerable time, even a number of times longer than computation does. For GNN inference processing, the transmission is much more complex by combining all the 4 different phases in SAGA. According to previous work [16], it is inefficient to employ basic interconnection topology such as bus or mesh for handling GNN's transmission patterns. As a result, we develop a 2-layer hierarchical interconnection network for handling not only local communication efficiently in the cluster but also global communication between clusters with low power and area overhead.

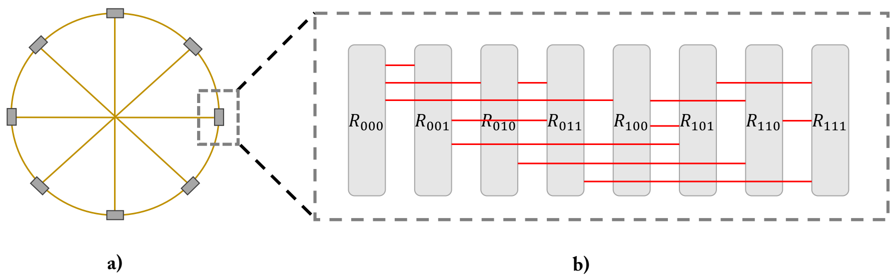

Fig. 6. Network hierarchy

Inter-cluster Interconnection Networks: We use the Octagon topology [15] for the global network, as shown in Fig. 6(a). Each block on the octagon is a node cluster. The $O(n)$ ring topology is cost effective, and can fit data movement pattern of the node cluster architecture well. The Octagon topology is constructed based on a ring network. It guarantees that there exist at most two hops of a transaction between two arbitrary routers in the same ring. As for cross-ring transaction, the maximum number of hops increases to $2n$, where n is the number of pass-by rings. Furthermore, the low complexity of global wiring demonstrates the great scalability of GNN-PIM's networks.

Noted that the main cost of *Scatter* is to move data between clusters. To illustrate the superiority of our dataflow on this topology, we use the total hops in global network as the metric since traversing through global wires consumes much more time and energy than local ones. There are 8 clusters in total and they are connected by a single *Octagon* Ring. Without considering deadlock or non-optimal routing, the minimal global hops of preparing data for an E-Node is $1 \times \frac{3}{7} + 2 \times \frac{4}{7} = 1.57$ on average. And the number decreases to 1 by employing the optimized dataflow, as each transaction is only between two adjacent clusters. Furthermore, most nodes keep busy and no spatial broadcast is needed, which eliminates the bandwidth demand of global interconnection network. The storage overheads consist of the vertex of the graph and one edge chunk per cluster. These overheads can be ignored, compared with holding the whole edge data.

Intra-cluster Interconnection Networks: The topology employed to connect nodes in the same group is illustrated in Fig. 6(b), and each router is in charge of the data transmission related to a node. In this example, n routers are connected via the local network. Each router is connected to other $logn$ routers, i.e. each router has the radix of $logn$. Since each number with $h = logn$ bits can change to another number with the same bit in h steps by inverting single bit per step, each transaction goes through at most $logn$ hops from source to destination.

6 Evaluation

In this section, we first present the detailed evaluation setup. It is worth noticing that GNN-PIM is applicable for various memory technologies, and we choose ReRAM to prototype GNN-PIM in this work. Then we perform comparison between GNN-PIM and prior works in performance. We also present comprehensive analysis on power consumptions of GNN-PIM.

6.1 Benchmark

We use the datasets mentioned in Neugraph [20] to evaluate GNN-PIM's performance and power consumptions. The vertex number, edge number, feature size, and model size of these real-world graphs are summarized in Table 1.

Table 1. Summary of real-world graphs

Dataset	Vertex	Edge	Feature	Storage
pubmed	19.7K	108.4K	500	500 KB
blog	10.3K	668.0K	128	2.6 MB
redditsmall	58.2K	1.4M	300	5.8 MB
redditfull	2.4M	705.9M	300	2883 MB
enwiki	3.6M	276.1M	300	1118 MB
amazon	8.6M	231.6M	96	960 MB

6.2 Methodology

Baseline: We employ a state-of-the-art design, PRIME [8], as our baseline. We adapt the size of PRIME circuits and scale the power and performance reported in that paper for a fair comparison. For simplicity, we treat its interconnection topology as a bus.

GNN-PIM Configuration: We set the overall size of ReRAM to 16 GB, and it is divided into 8 clusters with 2 GB each. There are 32 nodes in each cluster, connected by local networks. We statically assign 4 of them as V-Nodes while the other 28 nodes are N-Nodes. For processing elements, we adapt the same configurations used in PRIME [8]. Resolution of ADC and DAC is 5 bits and 2 bits, respectively. The power and area of the circuit are modeled based on ISAAC [24]. The HRS/LRS resistances are $25\,\mathrm{M\Omega}/50\,\mathrm{K\Omega}$, and read/write voltages are 0.7 V/2 V. The latency and energy cost of read/write are 29.31 ns/50.88 ns and 1.08 pJ/3.91 nJ, respectively. The on-chip network design adapts Booksim [13] to simulate the latency, and employ the model proposed in ORION [14] to simulate power consumption and area overhead of the NoC. Both global network and local network work on 1 GHz. 4 nodes share a physical router with 1 global channel and 7 local channels. To fully utilize the bandwidth of local networks, we place V-Nodes of a cluster in different physical routers.

6.3 Performance Results

The time consumption normalized to PRIME is illustrated in Fig. 7. The performance of PRIME is obtained by applying the dataflow generated by SAGA framework directly on its circuits. PRIME's primitive architecture leads to intensive competition and conflicts while using bus for data communication. As a result, the majority of PEs stay idle waiting for the data.

Therefore, the performance improvement of GNN-PIM comes from two folds. On one hand, hierarchy interconnection as well as optimized dataflow cooperate to reduce data transmission latency. On the other hand, parallel broadcast mechanism delivers data to multiple processing elements simultaneously, enabling more PEs working in parallel.

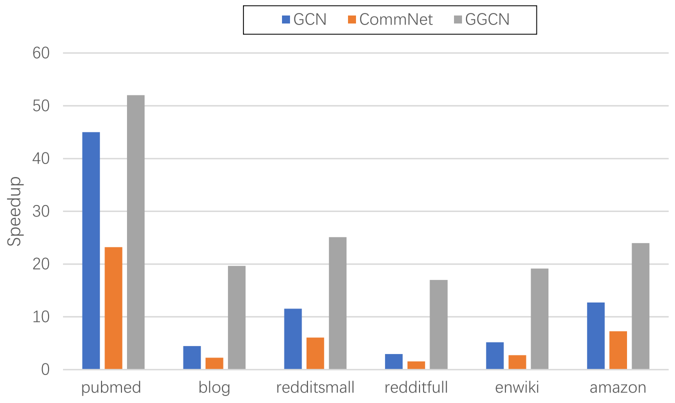

Fig. 7. Speedup over mapping GNNs directly on PRIME

Figure 8 illustrates the breakdown of power consumption, in which we choose GGCN as the benchmark. The power consumption of NoC could be heavy when

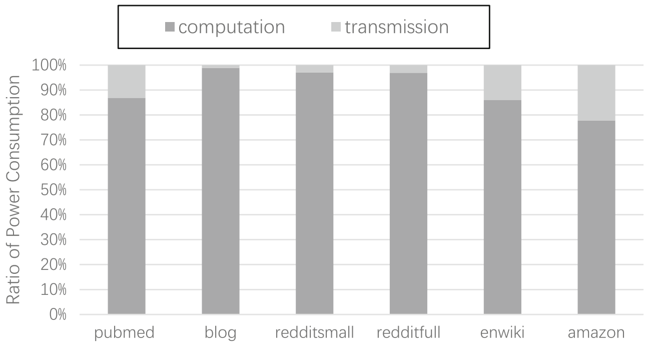

Fig. 8. Breakdown of power consumption

the graph is sparse or configured with short feature vectors. There are considerable redundant data transmission brought by edge-center programming model when the graph has high sparsity. Meanwhile, the feature size F will influence the computation with the factor of F^2 for matrix-vector multiplications, while for communication the factor is just F.

7 Conclusions

In this paper, we propose GNN-PIM, a PIM architecture for processing GNN inference. GNN-PIM employs a PIM-based hierarchical architecture with high throughput and efficiency. Besides, we perform customized optimizations on the dataflow, and propose a hierarchical NoC design to fully utilize the improvements brought by the optimized dataflow. Experimental results show that GNN-PIM achieves up to 52x speedup compared with prior designs.

Acknowledgement. This work is supported by National Key Research and Development Project of China (Grant No. 2018YFB1003304) and Beijing Academy of Artificial Intelligence (BAAI).

References

1. Aga, S., Jeloka, S., Subramaniyan, A., Narayanasamy, S., Blaauw, D., Das, R.: Compute caches. In: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp. 481–492. IEEE (2017)
2. Ahn, J., Hong, S., Yoo, S., Mutlu, O., Choi, K.: A scalable processing-in-memory accelerator for parallel graph processing. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, 13–17 June 2015, pp. 105–117 (2015). https://doi.org/10.1145/2749469.2750386
3. Angizi, S., He, Z., Fan, D.: PIMA-logic: a novel processing-in-memory architecture for highly flexible and energy-efficient logic computation. In: Proceedings of the 55th Annual Design Automation Conference, pp. 1–6 (2018)
4. Angizi, S., He, Z., Rakin, A.S., Fan, D.: CMP-PIM: an energy-efficient comparator-based processing-in-memory neural network accelerator. In: Proceedings of the 55th Annual Design Automation Conference, pp. 1–6 (2018)
5. Angizi, S., Sun, J., Zhang, W., Fan, D.: Aligns: a processing-in-memory accelerator for DNA short read alignment leveraging SOT-MRAM. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2019)
6. Asghari-Moghaddam, H., Son, Y.H., Ahn, J.H., Kim, N.S.: Chameleon: versatile and practical near-DRAM acceleration architecture for large memory systems. In: 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 1–13. IEEE (2016)
7. Boroumand, A., et al.: CoNDA: efficient cache coherence support for near-data accelerators. In: Proceedings of the 46th International Symposium on Computer Architecture, pp. 629–642 (2019)
8. Chi, P., et al.: PRIME: a novel processing-in-memory architecture for neural network computation in ReRam-based main memory. In: 43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, 18–22 June 2016, pp. 27–39 (2016). https://doi.org/10.1109/ISCA.2016.13

9. Dai, G., et al.: GraphH: a processing-in-memory architecture for large-scale graph processing. IEEE Trans. CAD Integr. Circ. Syst. **38**(4), 640–653 (2019). https://doi.org/10.1109/TCAD.2018.2821565
10. Farmahini-Farahani, A., Ahn, J.H., Morrow, K., Kim, N.S.: NDA: near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 283–295. IEEE (2015)
11. Hamilton, W.L., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, Long Beach, CA, USA, 4–9 December 2017, pp. 1024–1034 (2017). http://papers.nips.cc/paper/6703-inductive-representation-learning-on-large-graphs
12. Ji, Y., et al.: FPSA: a full system stack solution for reconfigurable ReRam-based NN accelerator architecture. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, 13–17 April 2019, pp. 733–747 (2019). https://doi.org/10.1145/3297858.3304048
13. Jiang, N., et al.: A detailed and flexible cycle-accurate network-on-chip simulator. In: 2012 IEEE International Symposium on Performance Analysis of Systems & Software, Austin, TX, USA, 21–23 April 2013, pp. 86–96 (2013). https://doi.org/10.1109/ISPASS.2013.6557149
14. Kahng, A.B., Li, B., Peh, L., Samadi, K.: ORION 2.0: a power-area simulator for interconnection networks. IEEE Trans. Very Large Scale Integr. Syst. **20**(1), 191–196 (2012). https://doi.org/10.1109/TVLSI.2010.2091686
15. Karim, F., Nguyen, A., Dey, S.: An interconnect architecture for networking systems on chips. IEEE Micro **22**(5), 36–45 (2002)
16. Kwon, H., Samajdar, A., Krishna, T.: Rethinking NoCs for spatial neural network accelerators. In: Proceedings of the Eleventh IEEE/ACM International Symposium on Networks-on-Chip, NOCS 2017, Seoul, Republic of Korea, 19–20 October 2017, pp. 19:1–19:8 (2017). https://doi.org/10.1145/3130218.3130230
17. Li, S., Niu, D., Malladi, K.T., Zheng, H., Brennan, B., Xie, Y.: DRISA: a DRAM-based reconfigurable in-situ accelerator. In: 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 288–301. IEEE (2017)
18. Li, S., Xu, C., Zou, Q., Zhao, J., Lu, Y., Xie, Y.: Pinatubo: a processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In: Proceedings of the 53rd Annual Design Automation Conference, pp. 1–6 (2016)
19. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, 2–4 May 2016, Conference Track Proceedings (2016). http://arxiv.org/abs/1511.05493
20. Ma, L., et al.: Neugraph: parallel deep neural network computation on large graphs. In: 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, 10–12 July 2019, pp. 443–458 (2019). https://www.usenix.org/conference/atc19/presentation/ma
21. Malewicz, G., et al.: Pregel: a system for large-scale graph processing. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, 6–10 June 2010, pp. 135–146 (2010). https://doi.org/10.1145/1807167.1807184
22. Sen, P., Namata, G., Bilgic, M., Getoor, L., Gallagher, B., Eliassi-Rad, T.: Collective classification in network data. AI Mag. **29**(3), 93–106 (2008). http://www.aaai.org/ojs/index.php/aimagazine/article/view/2157

23. Seshadri, V., et al.: Ambit: in-memory accelerator for bulk bitwise operations using commodity DRAM technology. In: 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pp. 273–287. IEEE (2017)
24. Shafiee, A., et al.: ISAAC: a convolutional neural network accelerator with insitu analog arithmetic in crossbars. In: 43rd ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2016, Seoul, South Korea, 18–22 June 2016, pp. 14–26 (2016). https://doi.org/10.1109/ISCA.2016.12
25. Singh, G., et al.: NAPEL: near-memory computing application performance prediction via ensemble learning. In: 2019 56th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2019)
26. Song, L., Qian, X., Li, H., Chen, Y.: Pipelayer: a pipelined ReRam-based accelerator for deep learning. In: 2017 IEEE International Symposium on High Performance Computer Architecture, HPCA 2017, Austin, TX, USA, 4–8 February 2017, pp. 541–552 (2017). https://doi.org/10.1109/HPCA.2017.55
27. Song, L., Zhuo, Y., Qian, X., Li, H.H., Chen, Y.: GraphR: accelerating graph processing using ReRam. In: IEEE International Symposium on High Performance Computer Architecture, HPCA 2018, Vienna, Austria, 24–28 February 2018, pp. 531–543 (2018). https://doi.org/10.1109/HPCA.2018.00052
28. Tang, L., Liu, H.: Relational learning via latent social dimensions. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, 28 June–1 July 2009, pp. 817–826 (2009). https://doi.org/10.1145/1557019.1557109
29. Xie, L., Du Nguyen, H.A., Taouil, M., Hamdioui, S., Bertels, K.: Fast boolean logic mapped on memristor crossbar. In: 2015 33rd IEEE International Conference on Computer Design (ICCD), pp. 335–342. IEEE (2015)
30. Yan, M., et al.: HyGCN: a GCN accelerator with hybrid architecture. CoRR abs/2001.02514 (2020). http://arxiv.org/abs/2001.02514
31. Yu, J., Du Nguyen, H.A., Xie, L., Taouil, M., Hamdioui, S.: Memristive devices for computation-in-memory. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), pp. 1646–1651. IEEE (2018)
32. Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., Sun, M.: Graph neural networks: a review of methods and applications. CoRR abs/1812.08434 (2018). http://arxiv.org/abs/1812.08434