HyMM: A Hybrid Sparse-Dense Matrix Multiplication Accelerator for GCNs

Hunjong Lee†, Jihun Lee†, Jaewon Seo†, Yunho Oh†, Myung Kuk Yoon‡, and Gunjae Koo†
†*Department of Computer Science and Engineering, Korea University, Seoul, South Korea*
‡*Department of Computer Science and Engineering, Ewha Womans University, Seoul, South Korea*
E-mail: †{hunjong, jihunlee, yarks, yunho_oh, gunjaekoo}@korea.ac.kr, ‡myungkuk.yoon@ewha.ac.kr

***Abstract*—Graph convolutional networks (GCNs) are emerging neural network models designed to process graph-structured data. Due to massively parallel computations using irregular data structures by GCNs, traditional processors such as CPUs, GPUs, and TPUs exhibit significant inefficiency when performing GCN inferences. Even though researchers have proposed several GCN accelerators, the prior dataflow architectures struggle with inefficient data utilization due to the divergent and irregularly structured graph data. In order to overcome such performance hurdles, we propose a hybrid dataflow architecture for sparse-dense matrix multiplications (SpDeMMs), called HyMM. HyMM employs disparate dataflow architectures using different data formats to achieve more efficient data reuse across varying degree levels within graph structures, hence HyMM can reduce off-chip memory accesses significantly. We implement the cycle-accurate simulator to evaluate the performance of HyMM. Our evaluation results demonstrate HyMM can achieve up to 4.78× performance uplift by reducing off-chip memory accesses by 91% compared to the conventional non-hybrid dataflow.**

***Index Terms*—GCNs, Accelerator, SpDeMM**

I. Introduction

Graph convolutional networks (GCNs) adapt convolutional neural network (CNN) techniques to large-scale graph structures [1]–[3]. While CNNs are designed to extract features from regularly structured data such as images or audio, GCNs are capable of handling irregular data structures [4]–[9]. Hence, GCNs are widely deployed to applications that rely on irregular graph structures such as social network analysis, molecular modeling, genome analysis, and so on [2], [7], [10]–[12]. A single GCN inference layer includes two idiosyncratic processing steps: *combination* and *aggregation*. A combination step performs simple dot-product operations between weight parameters and features associated with each node. On the other hand, an aggregation step traverses graph nodes to collect features from neighbor nodes. Since typical graph structures include irregular and extremely sparse connections between nodes, an aggregation exhibits irregular data accesses to traverse adjacent graph nodes. Hence, aggregations perform poorly on typical parallel processor architectures such as GPUs and TPUs, which are optimized for regularly organized data structures [13], [14]. Recent studies reveal aggregation steps occupy a significant fraction of the overall execution time in GCNs [15], [16].

To tackle the performance challenges posed by GCN inferences, we propose HyMM, an efficient GCN inference accelerator architecture. Prior GCN accelerators have focused on optimizing hardware for handling sparse graph structures with large feature data [17]–[21]. However, we observe one of the major performance hurdles in GCN accelerators is the inefficient use of data locality in graph structures. This inefficiency arises because a single dataflow architecture cannot fully utilize data locality across nodes exhibiting divergent connection degrees. To overcome such limitations, we propose a hybrid dataflow for sparse-dense matrix multiplications (SpDeMMs). Our proposed architecture employs disparate dataflow architectures for graph tiles that exhibit different connection degrees. We implement a cycle-accurate simulator to evaluate HyMM. Our evaluation results exhibit our hybrid SpDeMM accelerator can improve the performance of GCN inferences up to 4.78× compared to the homogeneous dataflow architecture.

II. Background

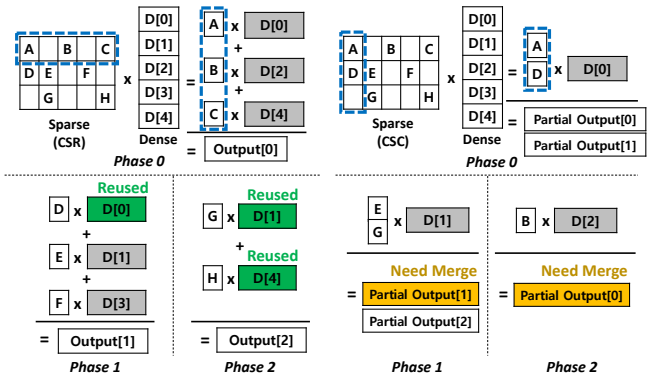
A. Graph Convolutional Networks

GCNs are widely used neural network models designed for graph structures [1], [2]. Each GCN layer comprises two main processes: aggregation and combination. In the aggregation phase, the GCN kernel traverses the graph to aggregate feature data from neighbor nodes. The combination phase performs dot-product operation between parameter weights and features associated with each node. GCN kernels execute multiple layer operations, thus these aggregation and combination phases are performed repeatedly. The equation below represents the operation of a single GCN inference layer, marked as (l).

$$H^{(l+1)} = \sigma\left(\hat{A}X^{(l)}W^{(l)}\right) \tag{1}$$

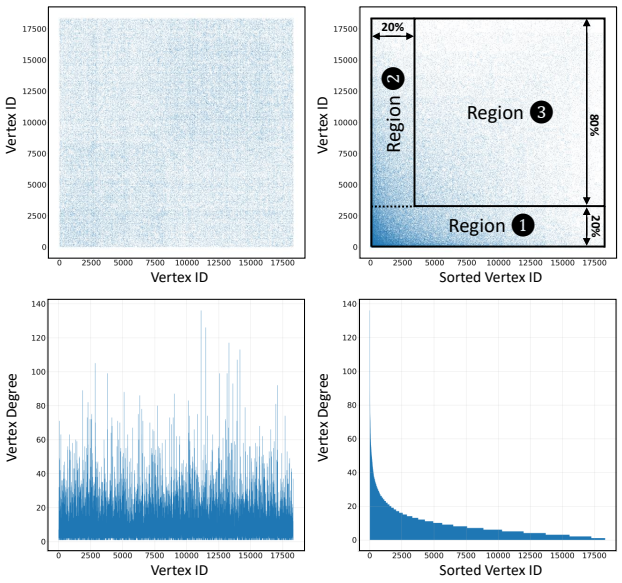
In this equation, A is an adjacency matrix that represents edges between connected nodes. Each node is associated with a feature vector, and the feature vectors of all nodes collectively form a feature matrix ($X^{(l)}$). The first term of the layer operation (i.e. $AX^{(l)}$) represents the aggregation of feature vectors from neighbor nodes. The aggregated features are normalized (i.e. $\hat{A}$) since nodes exhibit different edge counts. This is the aggregation phase of layer l. Note that the adjacency matrix exhibits extremely high sparsity as most nodes have only a few connections. In contrast, some nodes have a large number of connections. Hence, the adjacency matrix exhibits extremely imbalanced degrees across all nodes. Consequently, the aggregation phase provokes irregular memory accesses while traversing the graph. On the other hand, the combinations can be efficiently accelerated by conventional parallel processors such as GPUs since the combination involves dot-product operations between features (X) and weights (W), both of which are represented as dense matrices.

GCN inferences repeatedly execute the layer operations to derive more accurate graph features. Researchers revealed the



(a) Row-wise product (RWP) (b) Outer product (OP)

Fig. 1: SpDeMM dataflows



(a) Original graph (b) Sorted graph

Fig. 2: Graph degree distribution

sequence of aggregation and combination impacts the overall execution time of GCN inference. AWB-GCN employs a combination-first approach since it can reduce the number of multiplications, thereby decreasing inference time [17]. Several GCN accelerators adopt this combination-first approach to alleviate computation burdens [19]–[21]. Additionally, the combination-first approach lowers the hardware cost for implementing SpDeMM engines. In this work, we also employ the same approach to design our GCN accelerator.

B. Dataflows for SpDeMM

SpDeMM operations are essential computation kernels for GCN inferences. Since GCNs handle extremely sparse matrices (i.e. adjacency matrices), GCN accelerators leverage specialized dataflow architectures to design efficient SpDeMM engines. Figure 1 illustrate two common dataflow architectures for SpDeMM engines: the row-wise product (RWP) and the outer product (OP) approaches.

The RWP method employs a standard matrix multiplication approach, performing dot products between rows of the sparse matrix and rows of the dense matrix [22]. As depicted in Figure 1a, the RWP engine multiplies the non-zero elements in a row vector of the sparse matrix with the corresponding vectors of the dense matrix. The computed partial products are accumulated to generate the target row vector of the output matrix. In the example in Figure 1a, the RWP engine fetches three vectors (*D[0]*, *D[2]*, and *D[3]*) in phase 0 as the first row of the sparse matrix has three non-zero elements (*A*, *B*, and *C*). The RWP engine can complete the computations for the first row of the output matrix since all required partial products are computed in phase 0. In the next phase, the RWP engine performs dot-product operations for the second row. In this example, the second row has a non-zero element *D*, which is associated with *D[0]* of the dense matrix. Since *D[0]* was fetched during phase 0, *D[0]* can be found in the cache hierarchy. In this case, the accelerator will not create off-chip accesses for fetching *D[0]*. Likewise, *D[1]* and *D[4]* can be reused in phase 2. Consequently, the RWP approach can be efficient if many non-zero elements are found in the same column of the sparse matrix within a small window.

On the other hand, the OP approach, depicted in Figure 1b, multiplies the column of the sparse matrix with a single row of the dense matrix [23]. Thus, for one-column operation, the OP engine generates the partial sums of the output matrix for the non-zero elements in the column of the sparse matrix. As shown in the example in Figure 1b, the OP engine multiplies non-zero elements (*A* and *D*) in the first column of the sparse matrix with the first row vector of the dense matrix. This operation generates two partial outputs in phase 0. For the next column operation, the non-zero elements (*E* and *G*) are multiplied with the second row (*D[1]*), then the partial sum for the element *E* is updated in the output matrix. In summary, the OP engine repeatedly reads and writes the partial outputs while the row vectors of the dense matrix are streamed. Hence the OP approach can exploit the data locality in the partial outputs.

C. Graph Clustering

Several GCN accelerators rely on graph preprocessing to improve the efficiency of the computation engines that handle extremely sparse adjacency matrix [24]. Namely, once the graph nodes are reorganized to be grouped in dense clusters by the preprocessing method, GCN accelerators can exploit the data locality observed in high-dgree nodes. For instance, GROW improves the performance of GCN inferences by grouping high-degree nodes as high-density clusters cached in the on-chip memory of the accelerator [21]. G-CoD employs graph clustering techniques to create dense and sparse clusters to apply specific approaches separately. Even though graph clustering is an effective approach that can improve the efficiency of accelerators, preprocessing overhead cannot be ignored [25].

III. Motivation

Most adjacency matrices in graph datasets follow a power-law distribution [26], [27]. Namely, a small fraction of nodes exhibit very high degrees whereas the majority have relatively few connections. As shown in Figure 2, the top 20% of high-degree nodes account for more than 70% of the total edge count. This observation motivates that GCN accelerators can exploit the data locality inherent in high-degree nodes when processing sorted graphs.

TABLE I: Comparison of HyMM to other GCN accelerator architectures

	AWB-GCN [17]	GCNAX [19]	G-CoD [20]	GROW [21]	HyMM (ours)
Aggregation dataflow	Column-wise product	Outer product	Outer product	Row-wise product	Hybrid (row + outer)
Combination dataflow	Column-wise product	Outer product	Row-wise product	Row-wise product	Row-wise product
Compression format	CSC	CSC	CSC (A), CSR (others)	CSR	CSC (region 1), CSR (others)
Graph preprocessing	None	None	Partitioning & tuning	Graph partitioning	Degree sorting

As explained in Section II-B, dataflow architectures utilize different types of data locality. However, due to the highly imbalanced degree levels across nodes in graph structures, a single architecture cannot efficiently handle the diverse types of nodes. Table I summarizes the features of HyMM and other GCN accelerators. G-CoD and GCNAX employ OP engines for aggregation [19], [20]. Even though OP engines can efficiently exploit the data locality by non-zero elements in the same row indices, such architectures cannot reuse row vectors associated with clustered non-zero data in the same columns. Conversely, GROW employs RWP approach that can exploit data locality when multiple non-zero elements are grouped in the same columns [21]. However, this approach fails to leverage temporal locality in the output matrix when the rows of the sparse matrix include many non-zero data. Note that the performance of GCN inferences can be significantly degraded by increased off-chip accesses if accelerators fail to efficiently exploit the data locality inherent in high-degree nodes.

Based on the power-law distribution of degrees and the divergent data locality characteristics of the degree-sorted graph in Figure 2b, we can motivate a need for a hybrid dataflow architecture that can handle disparate degree-based clusters efficiently. By analyzing different regions in the adjacency matrix of the sorted graph, we can apply distinct dataflow approaches to maximize efficiency. Region ❶ favors OP engines. Note that many non-zero elements are found in the rows of region ❶ thus the accelerator can effectively exploit the temporal locality of the partial sums in the output matrix (i.e. AXW). In region ❷ RWP architecture can leverage the reuse of vectors in the combination result matrix (i.e. XW) since columns in the sparse matrix include many non-zero elements. Region ❸ is an extremely sparse part. We can deploy RWP engines to reduce the merging cost of partial outputs. Consequently, applying disparate dataflow architectures to different regions can enhance the performance of GCN inferences by efficiently reusing data elements within the accelerator's on-chip memory.

The proposed hybrid dataflow architecture can utilize integrated on-chip memory more flexibly. Prior GCN accelerators equip separated buffers for different types of matrices. On the other hand, the hybrid architecture includes a unified sparse/dense data buffer that can manage the space for input (XW) and output (AXW) data dynamically when performing GCN inferences in regions ❶ and ❷. For instance, while RWP dataflow is applied in region ❷, the unified buffer holds a substantial quantity of XW since RWP can leverage the data locality in the input matrix. On the other hand, the minimal space is assigned to the output matrix data not reused by RWP engines. In the same way, the hybrid architecture allocates more space for the output data while the OP approach is applied.

Determining the execution order of the two dataflows is critical due to their diverged characteristics. The OP architecture involves sequential input reads and irregular output writes, whereas the RWP dataflow demands random input reads and sequential output writes. We propose executing the OP mode first to prevent partial outputs from being evicted to off-chip memory. Additionally, placing a simple accumulator near the on-chip memory allows partial outputs of the same index to be accumulated, thereby reducing on-chip memory accesses.

IV. Hardware Architecture

In order to handle divergent (i.e. power-law distribution) degrees of nodes efficiently, HyMM adopts disparate dataflow architectures for different graph regions. Figure 3 describes the GCN accelerator architecture of HyMM. HyMM includes 16 processing engines (PEs) that support scalar-vector multiplications for SpDeMM operations. The PEs are designed to accommodate the dimension sizes commonly used in the GCN hidden layers. To reduce redundant memory accesses, a stationary buffer in each PE temporarily holds different types of elements (i.e. input and output matrix elements). HyMM employs on-chip buffers, a dense matrix buffer (DMB) and a sparse matrix queue (SMQ), to manage input/output elements for SpDeMM. The DMB serves as an integrated input/output element buffer that handles 64-byte vector data format. The controller in HyMM manages all data movements including load and store operations. We assume the bandwidth of the off-chip memory is 64 GB/s. The following sections describe the detailed operation of each hardware component in HyMM.

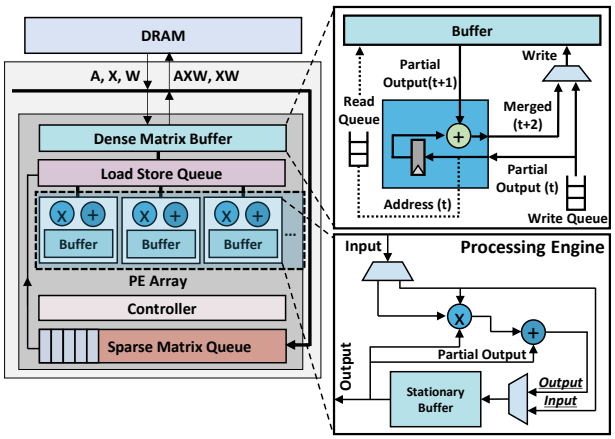


Fig. 3: HyMM architecture

A. Sparse Matrix Queue (SMQ)

To support SpDeMM operations efficiently, HyMM supports two different dataflow architectures that require different compressed formats. SMQ is designed to handle these two compressed sparse formats, CSR and CSC formats, in sparse matrices (A and X). Note that both the compressed formats share a common structure comprising pointers and indices. To accommodate this, the SMQ includes two types of buffers, a

pointer buffer and an index buffer. At the start of the operation, the controller fetches the pointer data from the off-chip memory to fill the pointer buffer. Then, the controller fetches indices and values using the pointer data. Figure 4 provides a detailed illustration of the SMQ structure.

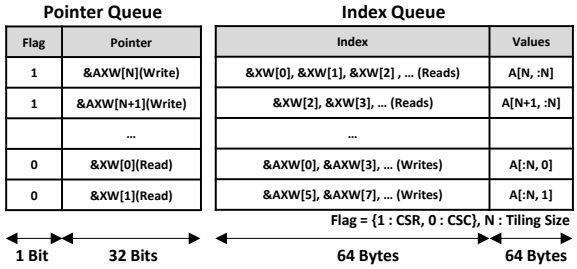


Fig. 4: Sparse matrix queue

A single entry in the pointer queue and the index queue is composed of flag, pointer, index, and values. The flag indicates the types of the compressed format (i.e. CSR or CSC). Based on the flag, the pointer represents either the row pointer of CSR or the column pointer of CSC. For the RWP dataflow, the row of the sparse matrix corresponds to the row index of the output matrix. Therefore, in a CSR entry, the pointer refers to the row index where the operation result will be written. Conversely, in the OP mode, the pointer represents the column of the sparse matrix. In this case, the pointer in a CSC entry indicates the address of the row in the dense matrix that needs to be loaded. The index specifies the load/write indexes in each CSR or CSC entry, while the values contain the data of the compressed matrix (A and X). These values serve as operands for multiplications. The data in an SMQ entry is delivered to the load/store queue and PE arrays when requested.

B. Load/Store Queue (LSQ)

Figure 5 depicts the structure of the LSQ. Each LSQ entry consists of four fields: valid, load/store, data, and address. The valid field indicates whether the entry contains valid data, and the load/store flag specifies the type of operation (i.e. load or store). HyMM minimizes the scope of irregular (XW) accesses during the RWP mode, leveraging the efficiency of the small and fast LSQ. As shown in the region ❷ of Figure 2b, read requests for narrow sections exploit spatial locality, significantly reducing the reliance on the DMB.

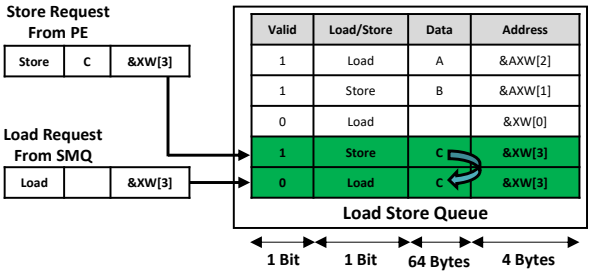


Fig. 5: Load/store queue

During the consecutive GCN inference operations (i.e. aggregation and combination), LSQ manages dependencies between load and store operations. LSQ enables efficient forwarding of data from stores (i.e. results of the combination phase) to subsequent dependent loads (i.e. loading XW during aggregation) without waiting for the store data to be written back to the off-chip memory. For example, when a store request from the PE is generated (e.g. $\&XW[3]$), it is entered into the LSQ with a store-type flag, along with its memory address and data. If a subsequent load request (e.g. $\&XW[3]$) from the SMQ is issued, it is added to the LSQ with a load-type flag and its memory address. However, if the load request's address matches an existing entry in the LSQ, the data from the corresponding store entry is immediately forwarded. In other words, since the store request from the PE already provided data for the same address, the LSQ fetches the data from that entry to fulfill the load request without accessing memory.

Additionally, LSQ can hide the latency of memory load/store instructions. While a missed load instruction waits for its data to be retrieved from the memory subsystem, subsequent load instructions targeting addresses already present in the LSQ can continue execution. This minimizes the pipeline stalls caused by the latency of missed load requests. Unlike traditional load/store units in CPUs, LSQ does not need to track the order of store instructions. It is because store instructions for a specific address, corresponding to the output matrices (XW and AXW), are unique. Such characteristics inherent to the workloads like matrix multiplications enable LSQ to maintain a simpler structure with minimal hardware overhead.

C. Processing Engine (PE)

HyMM employs PEs optimized for efficiently handling hybrid dataflow operations. The PE array is designed to perform scalar-vector multiplication and accumulation. Each PE includes a multiply-accumulate (MAC) unit and a stationary buffer that can support the RWP and OP operation modes. The stationary buffer in each PE is designed to store input elements as well as output results from the PE to support hybrid dataflow efficiently. The detailed operations of PE under the different dataflow modes are described below.

Row-wise product: LSQ reads a single scalar data from SMQ and broadcasts it to all PEs. Simultaneously, the vector data held in the corresponding entry in the LSQ is delivered to the multiple PEs. Each PE performs a MAC operation, and the result is stored in the stationary buffer in the PE. Subsequently, the next entry data (i.e. the scalar and vector data) in SMQ are transferred to PEs. Each PE again performs a MAC operation using the newly delivered data, then the result data is accumulated with the previous result held in the stationary buffer. Namely, during the RWP mode, PEs operates in an output-stationary dataflow manner to minimize data movement between the PEs and internal buffers.

Outer product: During the OP mode, the stationary buffer in each PE holds the input element to minimize the data movement between the PEs and buffers. Note that the elements in the dense matrix are repeatedly utilized in the OP dataflow architecture. PEs generate the partial outputs using the input elements stored in the stationary buffer. Note that the OP mode requires frequent read-modify-write operations to compute partial outputs and this is a significant performance burden in SpDeMM operations. HyMM applies an efficient tiling approach as shown in Figure 2b to make partial outputs

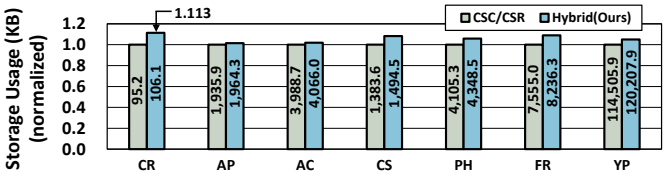


Fig. 6: Storage usage of adjacency matrix

merged. For this purpose, HyMM employs a near-memory processor that can merge partial outputs while accessing DMB.

Consequently, HyMM implements the efficient PE and internal buffer interfaces to support hybrid dataflow efficiently with minimal controls. HyMM's design aims to maximize operational efficiency while maintaining adaptability to different dataflows, making it a versatile and comprehensive solution for aggregation processing.

D. Dense Matrix Buffer (DMB) with an Accumulator

DMB comprises a buffer memory, an accumulator for near-memory processing, write/read queues, and miss status holding registers (MSHRs). Unlike the prior GCN accelerators that equip separate buffers for different data elements, DMB employs a unified buffer that can accommodate both the input (W and XW) and the output (XW and AXW) matrix elements to improve hardware efficiency significantly. We describe DMB's read and write operations below.

Read: LSQ sequentially sends read requests to the read queue and retrieves the corresponding entry from the buffer. In the event of a miss, it logs the request in the MSHRs and sends the request to the off-chip memory. If the target of the read request is already in the write queue, LSQ moves the request to the end of the read queue to prevent hazards.

Write with accumulation: DMB includes an accumulator for merging partial outputs to ensure efficient utilization of its limited buffer capacity. Figure 3 illustrates the DMB's operation. Initially, both the off-chip memory and LSQ sequentially enqueue write requests into the write queue. If the target address of a write request already exists in the buffer, the controller prioritizes its processing by moving the request to the front of the read queue. The partial output stored in the buffer is then merged with the new partial output using the accumulator and written back to the buffer. When the buffer reaches full capacity, data is evicted to the off-chip memory in the order of W and then XW, ensuring that partial outputs are retained. To further prevent the eviction of high-degree nodes, the buffer employs a least recently used (LRU) eviction policy.

E. Design Overhead

HyMM employs a tiling approach to sorted graph structure to efficiently perform aggregation using disparate dataflows. When applying the tiling approach, We consider the following design spaces of HyMM since the tiling requires extra pointer and index data to represent the position of non-zero elements.

Storage overhead: Figure 6 shows the storage usage of the original CSC/CSR-formatted matrix versus the tiled version by HyMM. For Cora the storage overhead is 10.2%. This is because smaller graphs tend to have fewer edges distributed across individual rows/columns. Consequently, as the graph size increases, the storage overhead cane decrease.

Tiling size: The tiling size is configured to ensure that both AXW (write requests in OP) and XW (read requests in RWP) required by each high-degree region fit to DMB. The maximum tiling size, referred to as the tiling threshold, is set to 20% of the total number of graph nodes. However, if the DMB is smaller than 20% of graph's nodes, the tiling is adjusted to accommodate as much XW or AXW as DMB can hold.

V. Evaluation

We implement a cycle-accurate simulator to evaluate HyMM. We study commonly used graph datasets from PyG as listed in Table II [28]. We compare the performance of HyMM with other GCN dataflow architectures. The RWP dataflow represents GROW [21], and the OP architecture represents GCNAX [19]. We assume the GCN accelerators employ the similar memory hierarchy such as sparse/dense buffers and PEs.

TABLE II: Graph datasets

Graph dataset	*# of nodes*	*# of edges*	*Adjacency sparsity*	*Feature sparsity*	*Feature length*	*Layer dimension*	*Sorting cost(ms)*
Cora (CR)	2,708	10,556	99.86%	98.73%	1,433	16	0.58
Amazon-Photo (AP)	7,650	238,162	99.59%	65.26%	745	16	2.62
Amazon-Computers (AC)	13,752	491,722	99.74%	65.16%	767	16	5.96
Computer-Science (CS)	18,333	163,788	99.95%	99.12%	6,805	16	3.42
Physics (PH)	34,493	495,924	99.96%	99.61%	8,415	16	6.80
Flickr (FR)	89,250	899,756	99.99%	53.61%	500	16	15.12
Yelp (YP)	716,847	13,954,819	99.99%	99.99%	300	16	215.93

To estimate the hardware cost of HyMM, we use Synopsys Design Compiler with ASAP 7nm library [29]. We use CACTI 7.0 to estimate the area of the memory components [30]. We apply the scaling to TSMC 40 nm library to compare with prior GCN accelerators [19], [21]. Table III lists the hardware parameter and estimated area of each hardware component of HyMM. Each PE supports single precision and has a width of 32 bits. HyMM achieve a performance of 32 GFLOPS. The number of PEs is consistent with that of GCNAX and GROW. The DMB consists of a 256KB buffer, two (read/write) queues, and an adder for accumulation. The SMQ consists of a single-ported 4KB and 12KB for the pointer buffer and index buffer, respectively. The LSQ has 128 entries, each of which is 68 bytes in size. The total area is 3.215 mm^2, which is smaller than GCNAX (6.51 mm^2) but larger than GROW (2.291 mm^2).

TABLE III: Hardware parameters and estimated area

Component	*Configuration*	*Area(mm^2)*	
		7nm	*40nm*
PE Array	16 MAC	0.006	0.21
DMB	256 KB	0.077	2.39
SMQ	16 KB	0.008	0.254
LSQ	128 Entries, 68B/Entry	0.009	0.292
Others	-	0.004	0.129
Total	-	*0.106*	*3.215*

A. Performance

Figure 7 compares the performance of the baselines and HyMM. Among the baselines, the row-wise product reduces execution time by up to 2.05× compared to the outer product on average. This improvement is due to the outer product generating a large number of partial outputs that require merging, causing disruptions in the PEs. HyMM achieves a maximum performance improvement of 4.78× over the outer product for AP. This is because HyMM effectively employs

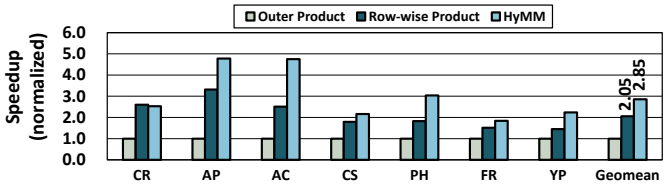


Fig. 7: Speedup of HyMM and baseline dataflows

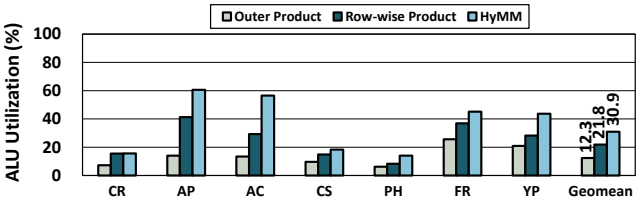


Fig. 8: Utilization of ALU

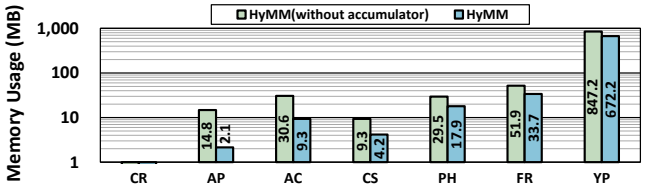


Fig. 10: Memory usage by parital outputs

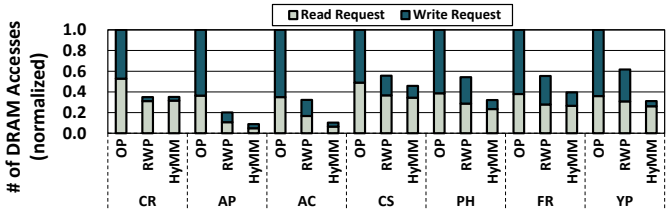


Fig. 11: DRAM access breakdown

a divide-and-conquer strategy by appropriately partitioning the graph datasets to apply the RWP and OP dataflow architectures selectively.

B. ALU Utilization

ALU utilization is a critical factor in the performance of SpDeMM dataflows. Figure 8 illustrates the percentage of ALU utilization (including both the multiplier and adder in the PE) during runtime. The outer product dataflow demonstrates the lowest ALU utilization, primarily due to wasted cycles caused by merging partial outputs and waiting for off-chip memory access. In contrast, our accelerator achieves significantly higher ALU utilization by merging partial outputs near the DMB instead of relying solely on the adder in the PE. This approach results in up to a 27% increase in ALU utilization compared to the row-wise product for AC. As shown in Figure 8, the PEs exhibit lower utilization for CR, CS, and PH workloads compared to others, which is attributed to high feature sparsity and the presence of long feature vectors.

C. Hit Rate

Figure 9 exhibits the hit rates, which indicate the proportion of requests where the target data is found in the buffers. Both dataflows, characterized by numerous irregular read and write requests, exhibit low hit rates. However, HyMM improves the hit rates by exploiting graph clustering and hybrid dataflow. This improvement can be attributed to HyMM's ability to confine the memory address range for write requests to AXW and read requests to XW. Additionally, HyMM efficiently reduces the partial outputs generated during the outer product by using the accumulator near DMB.

We analyze how HyMM's architecture and dataflow reduce the memory footprint of partial outputs. In traditional outer product implementations, partial outputs often consume substantial memory. As shown in Figure 10, without an accumulator, the memory footprint frequently exceeds the DMB's capacity, resulting in data being flushed to DRAM. However, by incorporating an accumulator near the DMB, our accelerator dramatically reduces this footprint, achieving reductions of up to 85% for AP.

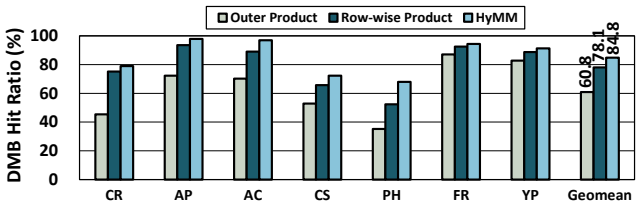


Fig. 9: Hit ratio of dense matrix buffer

D. Off-chip Memory Access

We analyze the off-chip DRAM accesses by SpDeMM dataflows and HyMM as shown in Figure 11. By exploiting data locality efficiently with the hybrid dataflow architecture, HyMM can effectively reduce off-chip accesses. Our evaluation results reveal HyMM reduces the off-chip memory accesses by 91% for AP and 89% for AC.

VI. Conclusion

In this work we propose a hybrid SpDeMM accelerator architecture for GCN inferences, called HyMM. Since graph structures exhibit power-law distributions in edge degrees, we motivate the different dataflow architectures exhibit diverged performance gains based on the connectivities of graph clusters. Based on our observations, we propose a hybrid SpDeMM architecture that employs row-wise and outer product approaches for different graph clusters. Our solution outperforms another dataflow by a maximum of $4.78\times$, indicating a substantial acceleration in performance. Moreover, we successfully mitigate the primary bottlenecks encountered in SpDeMM. This includes a reduction of up to 91% in off-chip memory accesses.These advancements serve to demonstrate the efficacy of our proposed accelerator and its potential for improving the efficiency of GCN inference.

Acknowledgment

This work was supported partly by the National Research Foundation of Korea (NRF) funded by the Korea government (MSIT) (NRF-2018R1C1B5086594) and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (IITP-2024-2020-0-01819, ICT Creative Consilience Program and RS-2021-II212068, Artificial Intelligence Innovation Hub). Gunjae Koo is the corresponding author.

REFERENCES

[1] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[2] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[3] S. Abadal, A. Jain, R. Guirado, J. López-Alonso, and E. Alarcón, "Computing graph neural networks: A survey from algorithms to accelerators," *ACM Computing Surveys (CSUR)*, vol. 54, no. 9, pp. 1–38, 2021.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[6] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[8] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[9] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[10] H. Gao, Z. Wang, and S. Ji, "Large-scale learnable graph convolutional networks," in *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, 2018, pp. 1416–1424.

[11] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," *arXiv preprint arXiv:1312.6203*, 2013.

[12] S. Yun, M. Jeong, R. Kim, J. Kang, and H. J. Kim, "Graph transformer networks," *Advances in neural information processing systems*, vol. 32, 2019.

[13] M. Yan, Z. Chen, L. Deng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Characterizing and understanding gcns on gpu," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 22–25, 2020.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[15] U. Alon and E. Yahav, "On the bottleneck of graph neural networks and its practical implications," *arXiv preprint arXiv:2006.05205*, 2020.

[16] I. Kim, J. Jeong, Y. Oh, M. K. Yoon, and G. Koo, "Analyzing gcn aggregation on gpu," *IEEE Access*, vol. 10, pp. 113 046–113 060, 2022.

[17] T. Geng, A. Li, R. Shi, C. Wu, T. Wang, Y. Li, P. Haghi, A. Tumeo, S. Che, S. Reinhardt *et al.*, "Awb-gcn: A graph convolutional network accelerator with runtime workload rebalancing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 922–936.

[18] M. Yan, L. Deng, X. Hu, L. Liang, Y. Feng, X. Ye, Z. Zhang, D. Fan, and Y. Xie, "Hygcn: A gcn accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 15–29.

[19] J. Li, A. Louri, A. Karanth, and R. Bunescu, "Gcnax: A flexible and energy-efficient accelerator for graph convolutional neural networks," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 775–788.

[20] H. You, T. Geng, Y. Zhang, A. Li, and Y. Lin, "Gcod: Graph convolutional network acceleration via dedicated algorithm and accelerator co-design," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 460–474.

[21] R. Hwang, M. Kang, J. Lee, D. Kam, Y. Lee, and M. Rhu, "Grow: A row-stationary sparse-dense gemm accelerator for memory-efficient graph convolutional neural networks," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 42–55.

[22] F. G. Gustavson, "Two fast algorithms for sparse matrices: Multiplication and permuted transposition," *ACM Transactions on Mathematical Software (TOMS)*, vol. 4, no. 3, pp. 250–269, 1978.

[23] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[24] G. Karypis and V. Kumar, "Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," 1997.

[25] V. Balaji and B. Lucia, "When is graph reordering an optimization? studying the effect of lightweight graph reordering across applications and input graphs," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2018, pp. 203–214.

[26] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 17–30.

[27] C. Xie, L. Yan, W.-J. Li, and Z. Zhang, "Distributed power-law graph computing: Theoretical and empirical analysis," *Advances in neural information processing systems*, vol. 27, 2014.

[28] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv preprint arXiv:1903.02428*, 2019.

[29] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "Asap7: A 7-nm finfet predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.

[30] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas, "Cacti 7: New tools for interconnect exploration in innovative off-chip memories," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 2, pp. 1–25, 2017.